

ECE 661 Homework 9

Andrea Goh (goh8@purdue.edu)

November 2024

1 Theory Question

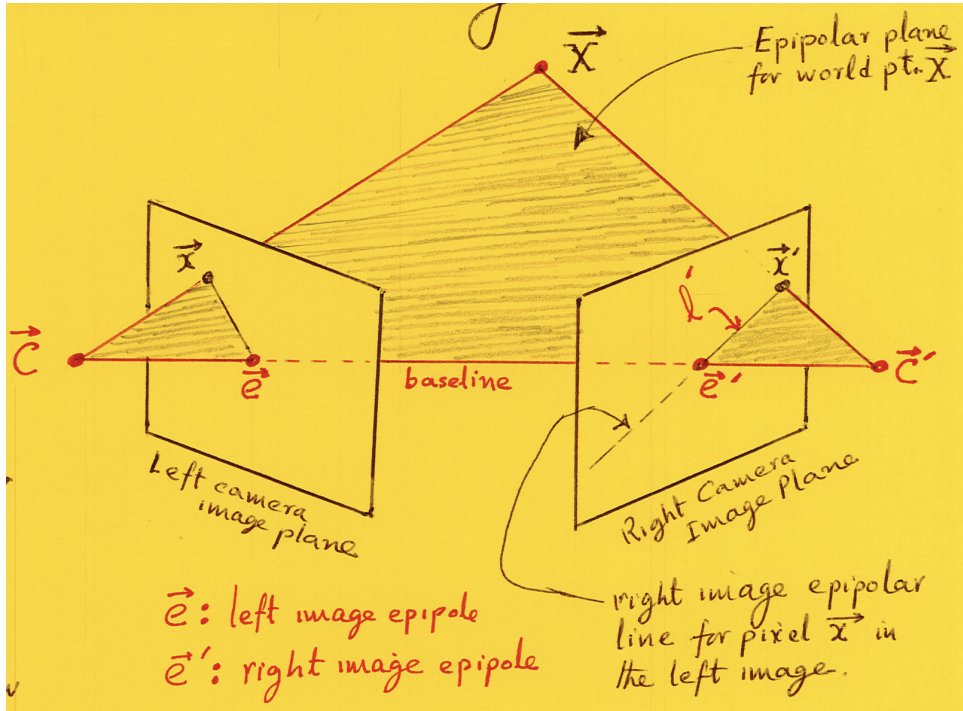


Figure 1: Caption

Each camera is modeled using the pinhole camera model, where a point world point $\mathbf{X}$ can be projected to image points $\vec{x}$ in the left image and $\vec{x}'$ in the right image by applying the projection matrix:

$$\vec{x} = P\mathbf{X} = K[I_{3 \times 3} | \vec{0}]\mathbf{X} \quad (1)$$

$$\vec{x}' = P'\mathbf{X} = K'[R | t]\mathbf{X} \quad (2)$$

For stereo images, the camera center of the left camera $\vec{C}$ is a world coordinate that can be projected onto a pixel in the right camera image plane $\vec{e}'$, and vice versa. This statement can be represented as follows:

$$\vec{e}' = P'\vec{C} \quad (3)$$

$$\vec{e} = P\vec{C}' \quad (4)$$

If we back project the pixel $\vec{x}$, the world point $P^\dagger \vec{x}$ is guaranteed to exist on the ray from $\vec{C}$ to $\vec{x}$. Projecting the world point $P^\dagger \vec{x}$ to the image plane in the right camera gives us $P'P^\dagger \vec{x}$, that gives us a pixel that lies on the epipolar line $\vec{\ell}'$. Thus, the projection of the left camera center to the right image plane and the projection of the world point $P^\dagger \vec{x}$ to the right image plane gives us two points in the right image plane, allowing us to form a line ℓ' : $\ell' = \vec{e}' \times P'P^\dagger \vec{x}$.

2 Task 1: Projective Stereo Reconstruction

The first part of this assignment is to create a 3D reconstruction from stereo images, taken from an uncalibrated camera. There are three main components to accomplishing this task: Image Rectification, Interest Point Detection, and Projective Reconstruction.

2.1 Image Rectification

The goal of image rectification is to find the homographies H and H' for the stereo images such that in the transformed image pair, the same world point appears on the same row. This can be done using the following steps:

1. Estimate initial fundamental matrix using point correspondence:

The fundamental matrix can be estimated using a minimum of 8 point correspondence. These point correspondence will first be normalized by doing: $\hat{x}_i = Tx_i$ and $\hat{x}'_i = T'x'_i$. The transformation matrix can be calculated as follows:

$$T = \begin{bmatrix} sI_2 & -s\bar{x} \\ 0^T & 1 \end{bmatrix} \quad (5)$$

In equation 5, $\bar{x}$ is the mean of the given points as $s = \frac{\sqrt{2}}{\frac{1}{n} \sum_{i=1}^n |x_i|}$. Then, the fundamental matrix can be estimated with the normalized correspondence points. Each $(\vec{x}, \vec{x}')$ correspondence gives us the equation $\vec{x}'^T \mathbf{F} \vec{x} = 0$, which expands out to

$$x'x f_{11} + x'y f_{12} + x'f_{13} + y'x f_{21} + y'y f_{22} + y'f_{23} + xf_{31} + yf_{32} + f_{33} = 0 \quad (6)$$

Equation 6 can be expressed as

$$A\vec{f} = 0 \quad (7)$$

In equation 7, $A = [x'x \ x'y \ x' \ y'x \ y'y \ y' \ x \ y \ 1]$ and $\vec{f} = (f_{11} \ f_{12} \ f_{13} \ f_{21} \ f_{22} \ f_{23} \ f_{31} \ f_{32} \ f_{33})^T$. For N number of correspondences, equation 7 can be stacked together to form $\mathbf{A}\vec{f} = \vec{0}$, where $\mathbf{A}$ is $N \times 9$, which can be solved using least-squares. The solution to $\mathbf{A}$ must be conditioned to enforce the requirement $\text{rank}(\mathbf{F}) = 2$. This can be done by doing its SVD to obtain UDV^T , and zeroing out the smallest singular value in D .

2. Estimate the left and right epipoles:

The right and left epipoles, $\mathbf{e}$ and $\mathbf{e}'$ respectively, are the left and right null vector of $\mathbf{F}$.

3. Obtain initial estimates of projection matrices in canonical form:

The left and right projection matrices, $\mathbf{P}$ and $\mathbf{P}'$ respectively, can be calculated using the following equations:

$$\begin{aligned} P &= [I_{3 \times 3} | \vec{0}] \\ P' &= [sF | \vec{e}'] \end{aligned} \quad (8)$$

4. Refine the right projection matrix using nonlinear optimization:

Given a $\mathbf{P}$ and $\mathbf{P}'$ for two cameras, use the current estimate for $\mathbf{F}$ to triangulate each corresponding pair $(\vec{x}_i, \vec{x}'_i)$ into the world point $\vec{X}_i$. Then, use the camera matrices $\mathbf{P}$ and $\mathbf{P}'$ to project $\vec{X}_i$ back into the two images, to obtain $(\hat{\vec{x}}_i, \hat{\vec{x}}'_i)$. This can be done by solving the following equation using linear least squares:

$$A\vec{X} = 0, \text{ where } A = \begin{bmatrix} x\vec{P}_3^T - \vec{P}_1^T \\ y\vec{P}_3^T - \vec{P}_2^T \\ x'\vec{P}'^T - \vec{P}'^1 \\ y'\vec{P}'^T - \vec{P}'^2 \end{bmatrix} \quad (9)$$

Then, the geometric error can be defined as:

$$d_{geom}^2 = \sum_i (||\vec{x}_i - \hat{\vec{x}}_i||^2 + ||\vec{x}'_i - \hat{\vec{x}}'_i||^2) \quad (10)$$

Using the geometric error in equation 10 as the cost function, we can then use Levenberg Marquardt optimization to obtain the refined $\mathbf{P}'$.

5. **Obtain the refined fundamental matrix $\mathbf{F}$ using the refined $\mathbf{P}'$:** With the refined $\mathbf{P}'$, the refined epipole $\vec{e}'$ is simply the last column of $\mathbf{P}'$. Then, the refined fundamental matrix can be calculated as follows:

$$\mathbf{F} = [\vec{e}']_{\times} \mathbf{P}' \mathbf{P}'^{\dagger} \quad (11)$$

6. **Estimate the right homography $\mathbf{H}'$ using refined epipoles:** The homography matrix for the right image is calculated using the following equation:

$$\mathbf{H}' = \mathbf{T}_2 \mathbf{G} \mathbf{R} \mathbf{T}_1 \quad (12)$$

$\mathbf{T}_2, \mathbf{G}, \mathbf{R}$, and $\mathbf{T}_1$ from equation 12 are the following matrices:

$$\mathbf{T}_1 = \begin{bmatrix} 1 & 0 & -w2/2 \\ 0 & 1 & -h2/2 \\ 0 & 0 & 1 \end{bmatrix} \quad (13)$$

$$\mathbf{G} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1/f & 0 & 1 \end{bmatrix} \quad (14)$$

$$\mathbf{R} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (15)$$

$$\mathbf{T}_2 = \begin{bmatrix} 1 & 0 & w2/2 \\ 0 & 1 & h2/2 \\ 0 & 0 & 1 \end{bmatrix} \quad (16)$$

7. **Estimate the left homography matrix $\mathbf{H}$:** The left homography can be estimated with the following steps:

- (a) $\mathbf{M} = \mathbf{P}' \mathbf{P}^{\dagger}$
- (b) $\mathbf{H}_0 = \mathbf{H}' \mathbf{M}$
- (c) Transform points in the left and right images, $\vec{x}$ and $\vec{x}'$, respectively to obtain: $\hat{\vec{x}} = H_0 \vec{x}$ and $\hat{\vec{x}}' = H' \vec{x}'$, where H' was obtained in step 6.
- (d) Then, we want to solve for $H_A = \begin{bmatrix} a & b & c \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ by minimizing $\min \sum_i (a\hat{x}_i + b\hat{y}_i + c - \hat{x}'_i)^2$
- (e) The right homography can then be calculated as: $H = H_A H_0$

8. **Apply $\mathbf{H}$ and $\mathbf{H}'$ to the stereo pair to rectify the images:**

2.2 Interest Point Detection

Before obtaining the projective 3-D reconstruction, interest points are obtained using OpenCV's Canny edge detection function. Then, correspondences between interest points are obtained using the SSD metric. Since the pictures used has a noisy background, cv2.medianBlur is first used to blur the image, followed by cv2.Canny to obtain edges. Then, for each edge coordinate of the left image, the SSD metric used for every edge pixel in the right image within 10 rows of the base pixel. Once these interest points and their correspondences has been found on the rectified image, the inverse homographies have to be applied to them to obtain the respective pixel coordinates of the original image.

2.3 Projective Reconstruction

From the interest points in the previous section, the world coordinates can be found using triangulation, using equation 9

3 Task 1 Results

Correspondence	Point from left image	Points from right image
1	(211, 133)	(274, 122)
2	(164, 122)	(236, 100)
3	(354, 253)	(357, 272)
4	(183, 168)	(238, 145)
5	(179, 185)	(228, 161)
6	(444, 166)	(480, 216)
7	(241, 90)	(316, 90)
8	(186, 158)	(243, 138)
9	(415, 160)	(454, 202)
10	(234, 88)	(311, 86)
11	(423, 171)	(459, 214)
12	(173, 244)	(213, 213)
13	(249, 156)	(299, 152)
14	(418, 107)	(476, 153)
15	(446, 137)	(495, 189)
16	(378, 218)	(396, 246)
17	(181, 248)	(217, 217)
18	(205, 188)	(248, 169)
19	(259, 134)	(317, 136)
20	(303, 70)	(379, 86)

Table 1: First 20 of 40 correspondences used



(a) Left Image

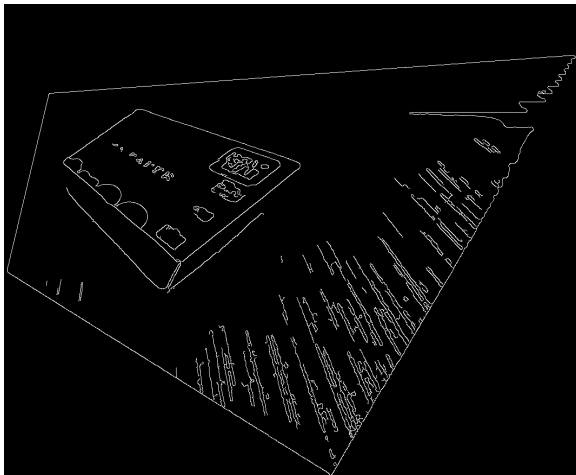


(b) Right Image

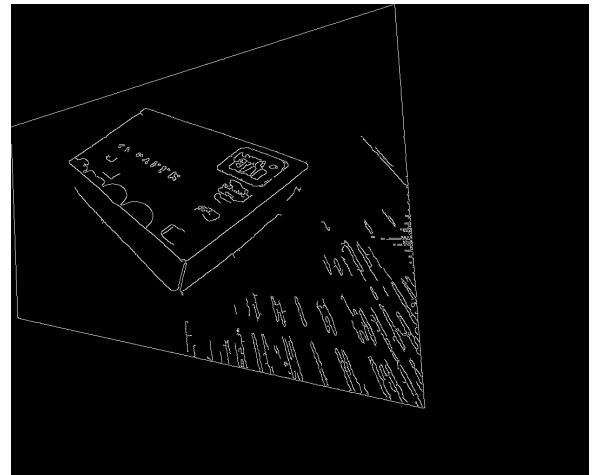
Figure 2: Images used for Task 1 and 2



Figure 3: Rectified Image

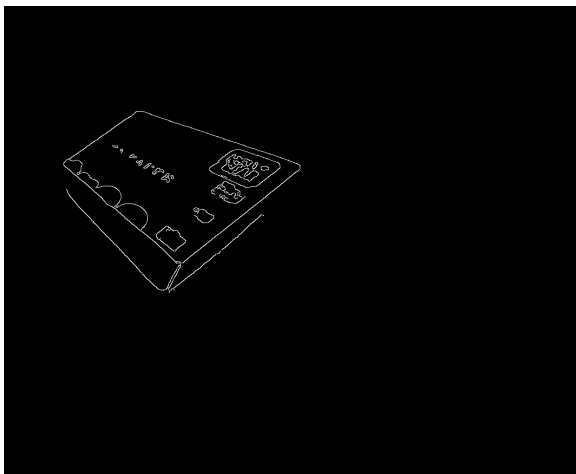


(a) Left Image

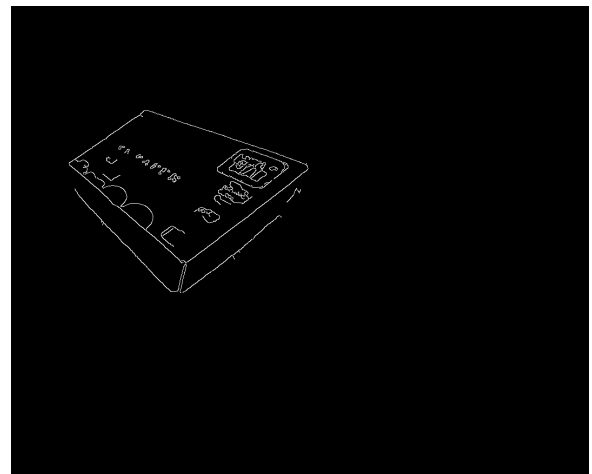


(b) Right Image

Figure 4: Edges obtained when cv2.Canny is directly applied to the rectified images.



(a) Left Image



(b) Right Image

Figure 5: Edges obtained when the rectified images were preprocessed using cv2.medianBlur and when the borders were constrained.

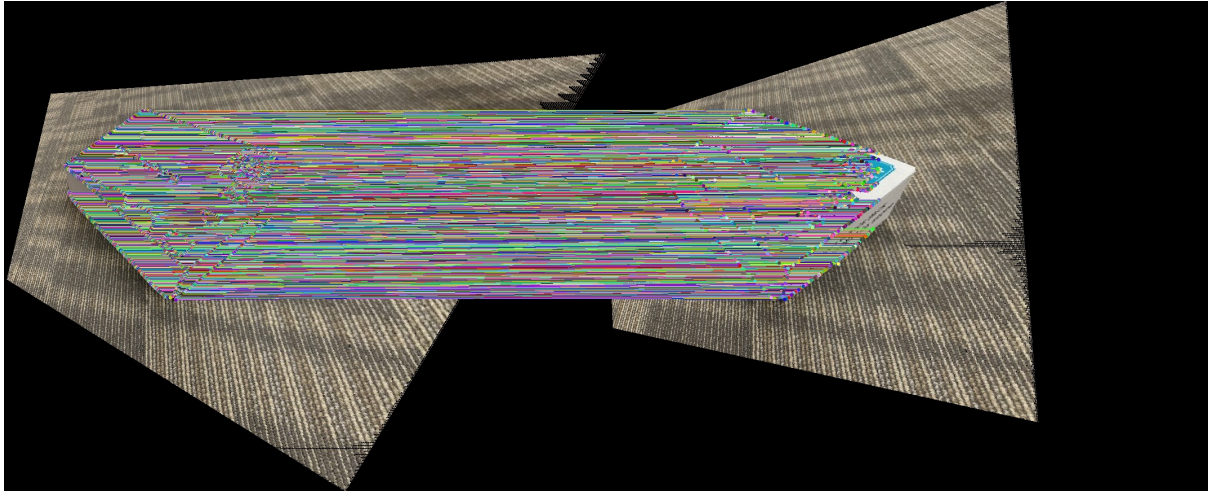


Figure 6: All interest points and correspondence using SSD metric.

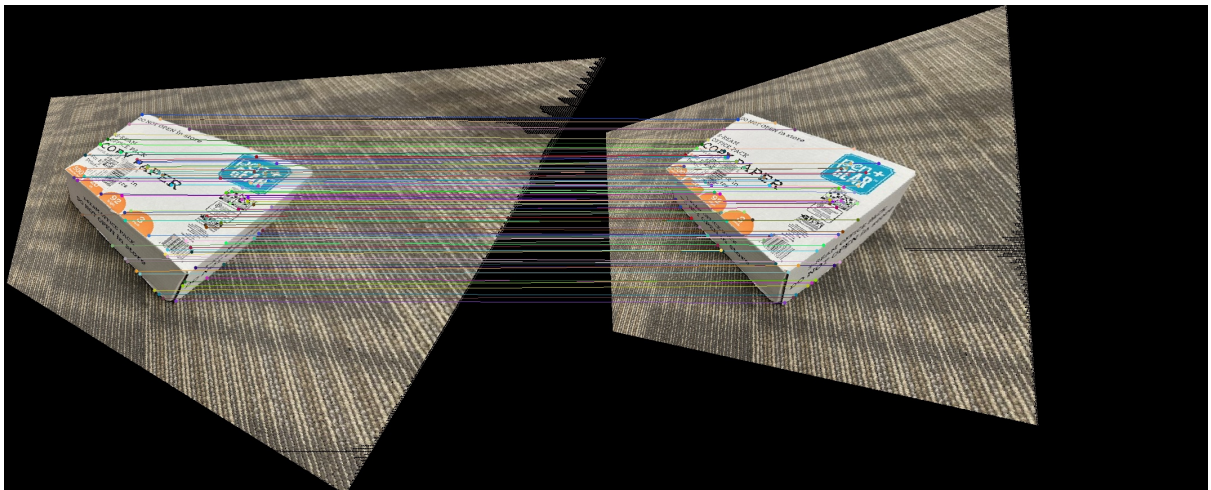


Figure 7: Downsampled version of figure 6

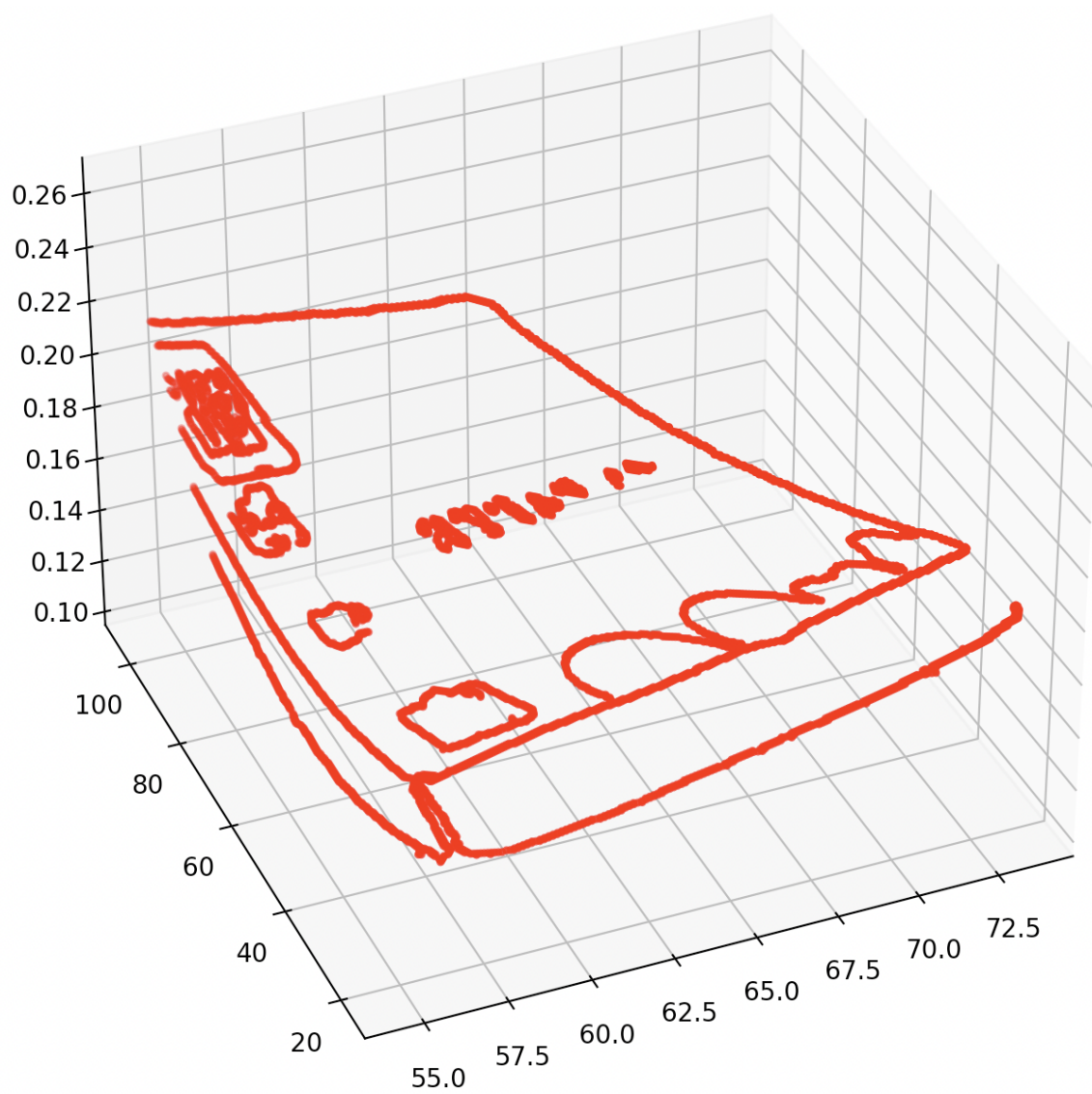


Figure 8: 3D Reconstruction using all correspondences from SSD, View 1

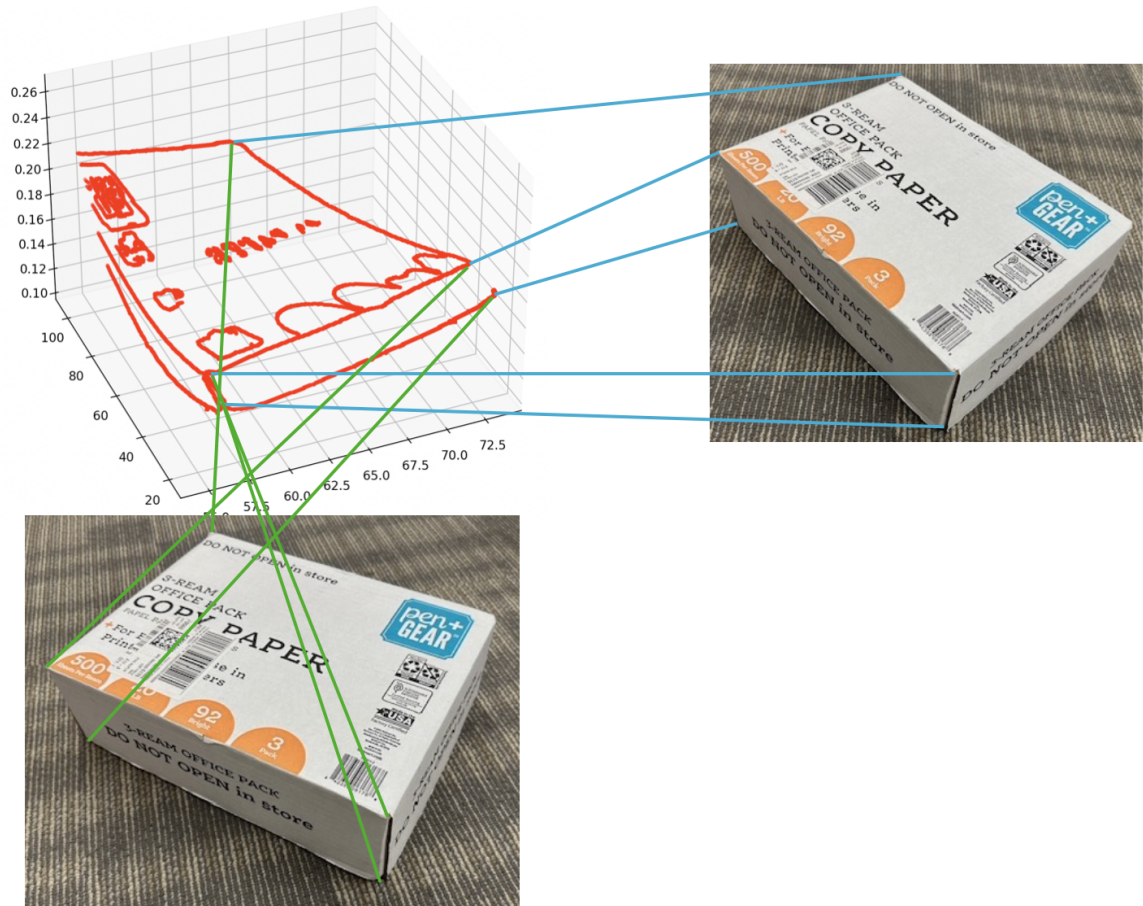


Figure 9: 3D Reconstruction using all correspondences from SSD with stereo images, view1

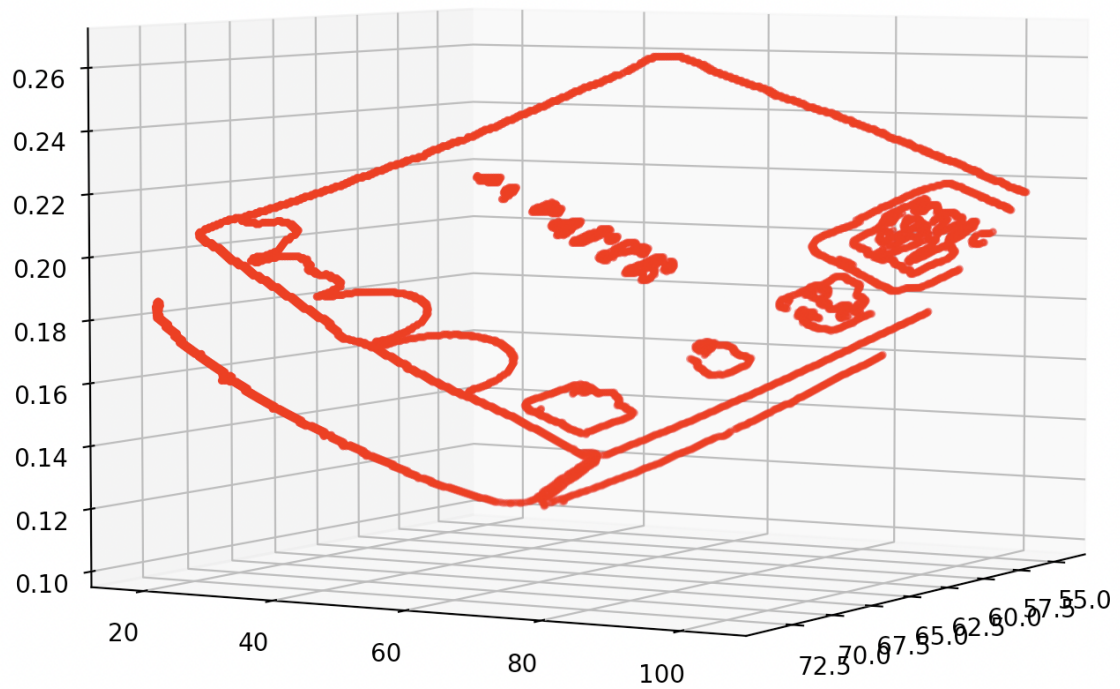


Figure 10: 3D Reconstruction using all correspondences from SSD, View 2

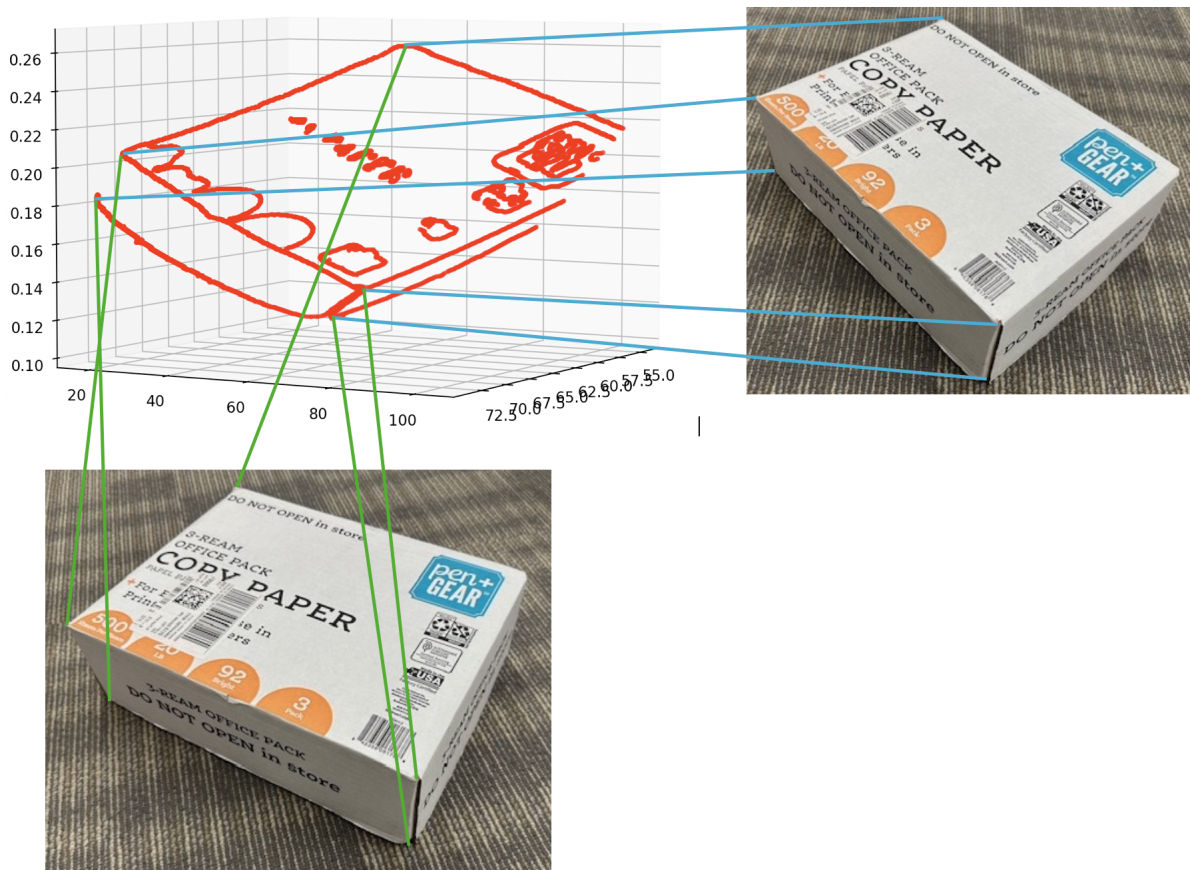


Figure 11: 3D Reconstruction using all correspondences from SSD with stereo images, view2

4 Task 2: The Loop and Zhang Algorithm

The Loop and Zhang algorithm decomposes the rectifying homographies as follows:

$$H = H_{sh}H_{sim}H_p \quad (17)$$

$$H' = H'_{sh}H'_{sim}H'_p \quad (18)$$

In equation 18, H_p and H'_p are the purely projective homographies whose purpose is to send the epipoles e and e' to infinity. H_{sim} and H'_{sim} are similarity homographies, which can only rotate, translate, and uniformly scale an image. Its purpose is to rotate the epipoles, which are at infinity, to the world-X axis. H_{sh} and H'_{sh} are shearing homographies, which introduces additional degrees of freedom to remove distortion that can't be removed by affine homographies.

5 Task 2 Results

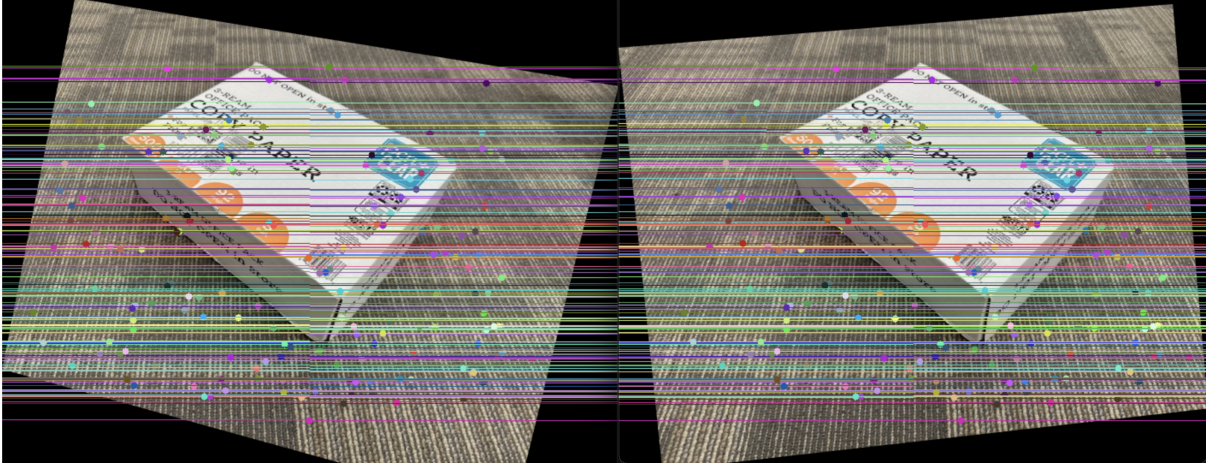


Figure 12: Caption

5.1 Comments

The Loop and Zhang algorithm was able to rectify the image pair by warping the two images to a lesser extent than my algorithm did. In addition, the rectified images from Loop and Zhang outperformed as the rows are visibly more aligned compared to my results. The correspondences are also of higher quality. In my pipeline, I used the SSD metric, which compares grayscale values in neighborhoods. This lead to a quite a bit of mismatched correspondences. However, the loop and Zhang algorithm seems to have resulted in much fewer mismatched correspondences.

6 Task 3: Dense Stereo Matching

In this task, we are trying to estimate the left disparity map. The steps for doing so are as follows:

1. For each pixel in the left image, (x,y) , the bitvector of size M^2 is calculated, by setting a bit one wherever the pixel value is strictly greater than the center pixel value.
2. For a range of corresponding pixels on the right $(x-d,y)$, where $d \in \{0 \dots d_{max}\}$, bitvectors are formed in the same manner as described in step 1.
3. After the bitwise XOR operation between the two bitvectors, the cost is computed as the number of ones in the output bitvector.
4. The disparity value d that minimizes the cost is chosen for the pixel on the left image (x,y) .

7 Task 3 Results



(a) Disparity Map



(b) Error Mask for $\delta = 2$

Figure 13: Accuracy = 20.56% when $m=3$

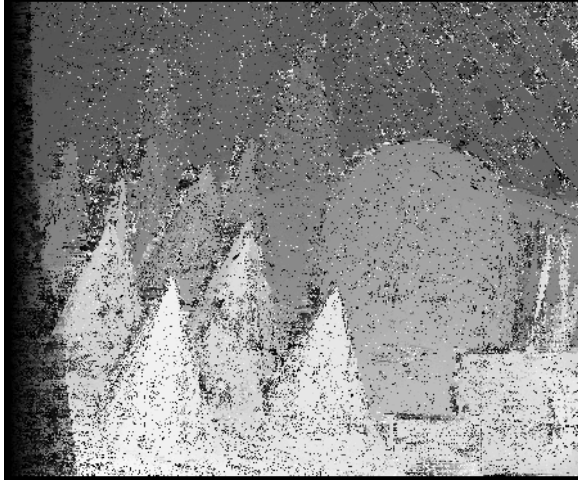


(a) Disparity Map

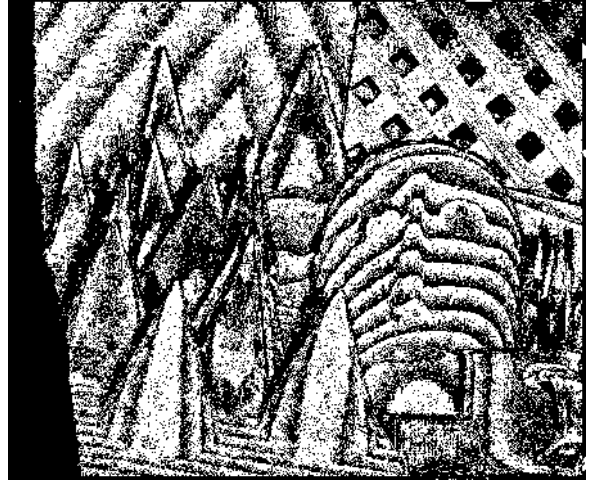


(b) Error Mask for $\delta = 2$

Figure 14: Accuracy = 40.37% when $m=5$

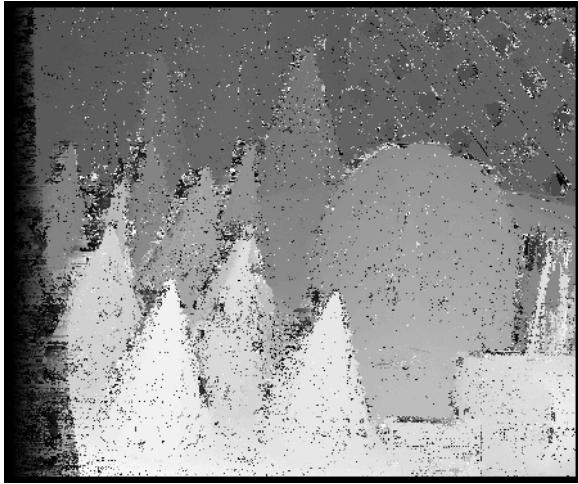


(a) Disparity Map

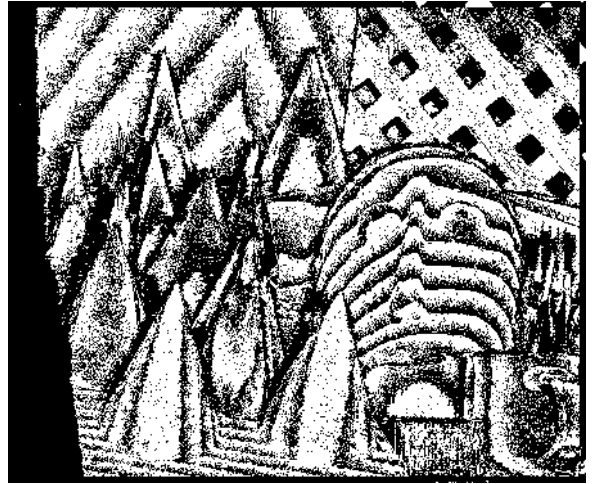


(b) Error Mask for $\delta = 2$

Figure 15: Accuracy = 48.30% when $m=7$



(a) Disparity Map



(b) Error Mask for $\delta = 2$

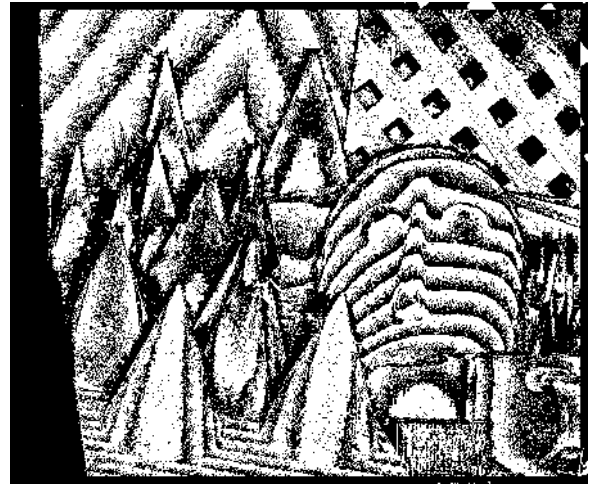
Figure 16: Accuracy = 50.85% when $m=11$

7.1 Observations

In figure 18, we can see that as the window size increases from a small initial value, the accuracy improves significantly. However, this improvement diminishes as the window size grows larger. Notably, at a window size of 17×17 , further increases in the window size result in a decline in accuracy.



(a) Disparity Map



(b) Error Mask for $\delta = 2$

Figure 17: Accuracy = 51.70% when $m=13$

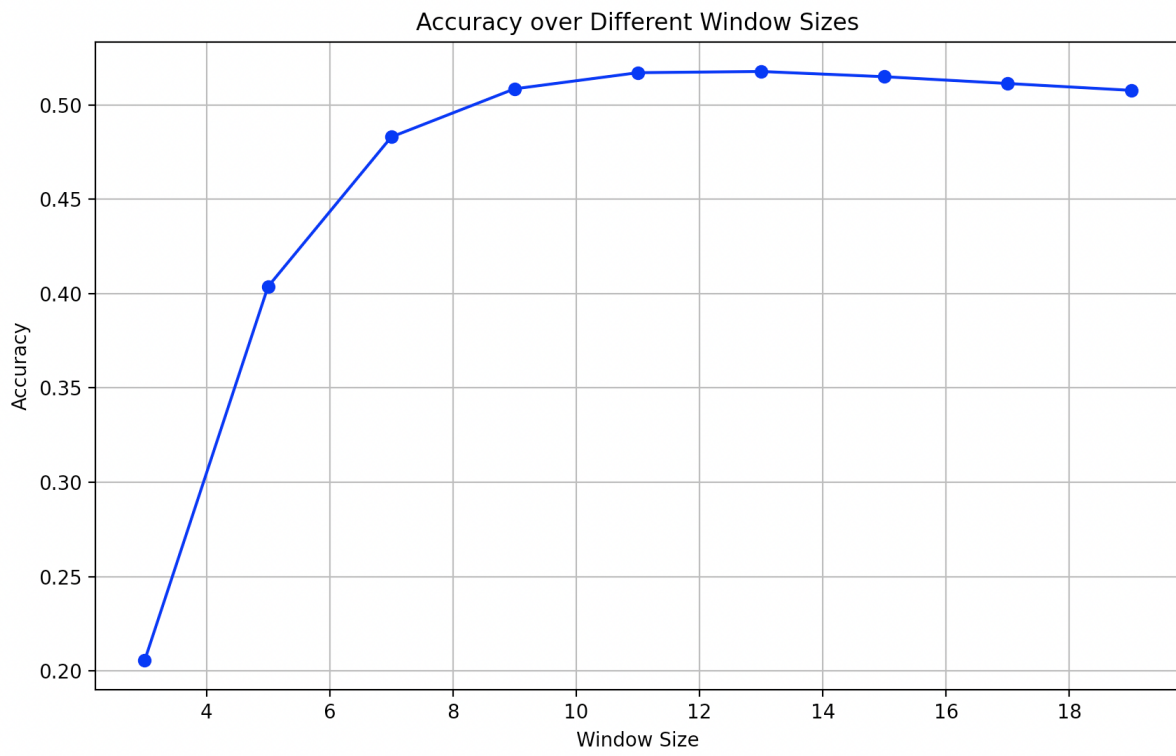


Figure 18: Accuracies at different window sizes

8 Task 4: Depth Map and Automatic Extraction of Dense Correspondences

Automatic Extraction of Dense Correspondences has many useful applications such as stereo matching, image registration, and 3D reconstruction. It is also used in deep learning applications as dense correspondences are essential for training models like SuperPoint, LoFTR, and CAPS, which rely on accurate keypoint matching.

The correspondences ($\{x_i^A \leftrightarrow x_i^B\}$) between a pair of images (I_A, I_B) can be automatically extracted using the given depth maps of the images (D_A, D_B). The general idea is to use the depth map to estimate the 3D coordinates of each pixel in the image, and project these 3D coordinates back into the image plane of the other image to array out a "Depth Check", to find correspondences. The "Depth-Check" process is outlined in the following steps:

1. Calculate the inverse of the camera matrix K_A . K_A can be calculated using the following equation:

$$\mathbf{K} = \begin{bmatrix} \alpha_x & s & x_0 \\ 0 & \alpha_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} \quad (19)$$

2. Calculate the 3D coordinate ($\mathbf{X}_{cam}^A$) of any point on the ray corresponding to the pixel in the camera coordinate system using the following equation:

$$\mathbf{X}_{cam} = \mathbf{K}^{-1}x_i \quad (20)$$

3. Get the 3D coordinate of the world point with the depth value $D_A(x_i^A)$ by multiplying equation 20 by the depth value, which will give us the 3D coordinate of the point in terms of the camera coordinate from of image I_A

$$\mathbf{X}_{cam} = [\mathbf{R}|\mathbf{t}]\mathbf{X}_{world} \quad (21)$$

4. Convert the 3D coordinate to the world coordinate frame using camera extrinsic parameters: ($[\mathbf{R}_A|\mathbf{t}_A]$):

$$\mathbf{X}_{world_{hc}} = \mathbf{T}^{-1}\mathbf{X}_{cam_{hc}} \quad (22)$$

$$, \text{ where } \mathbf{T}_{4 \times 4} = \begin{bmatrix} \mathbf{R} & \mathbf{r} \\ 0 & 1 \end{bmatrix}$$

5. Calculate the Z_{cam} coordinate of $\mathbf{X}_{world}$ with respect to the camera coordinate frame of I_B using the camera extrinsic of I_B , which can be done using $\mathbf{X}_{cam}^B = [\mathbf{R}_B|\mathbf{t}_B]\mathbf{X}_{world}$. This Z_{cam} coordinate is the estimated depth value.
6. Project the 3D world coordinate to I_B to get the corresponding pixel location x_i^B and use the depth map $\mathbf{D}_B$ to obtain the true depth $\mathbf{D}_B(x_i^B)$.
7. Calculate the absolute difference between the true and projected depth values and if it is less than a certain threshold, the two points $x_i^A \leftrightarrow x_i^B$ are matching correspondences.

8.1 Task 4 Results

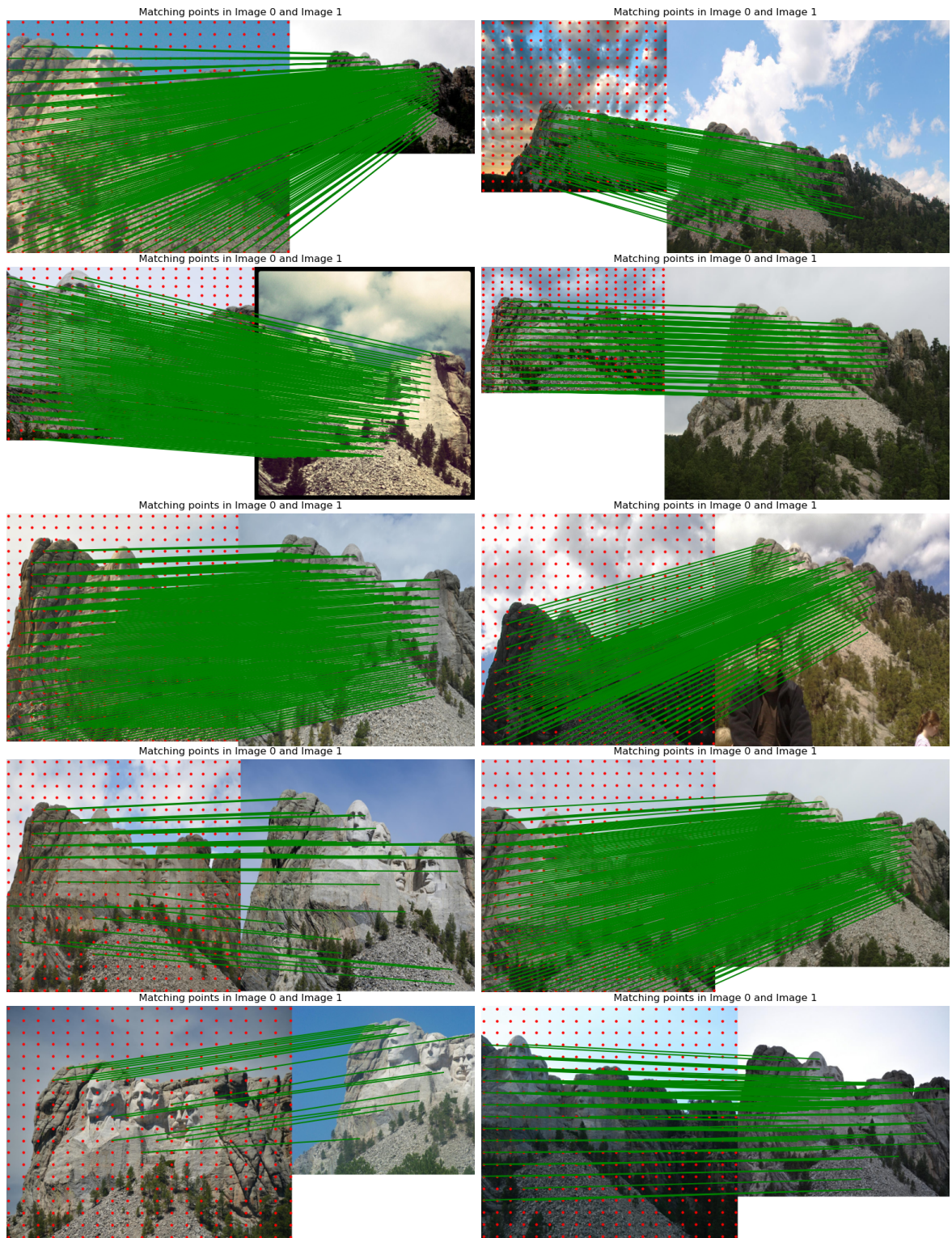


Figure 19: Depth Check Pair

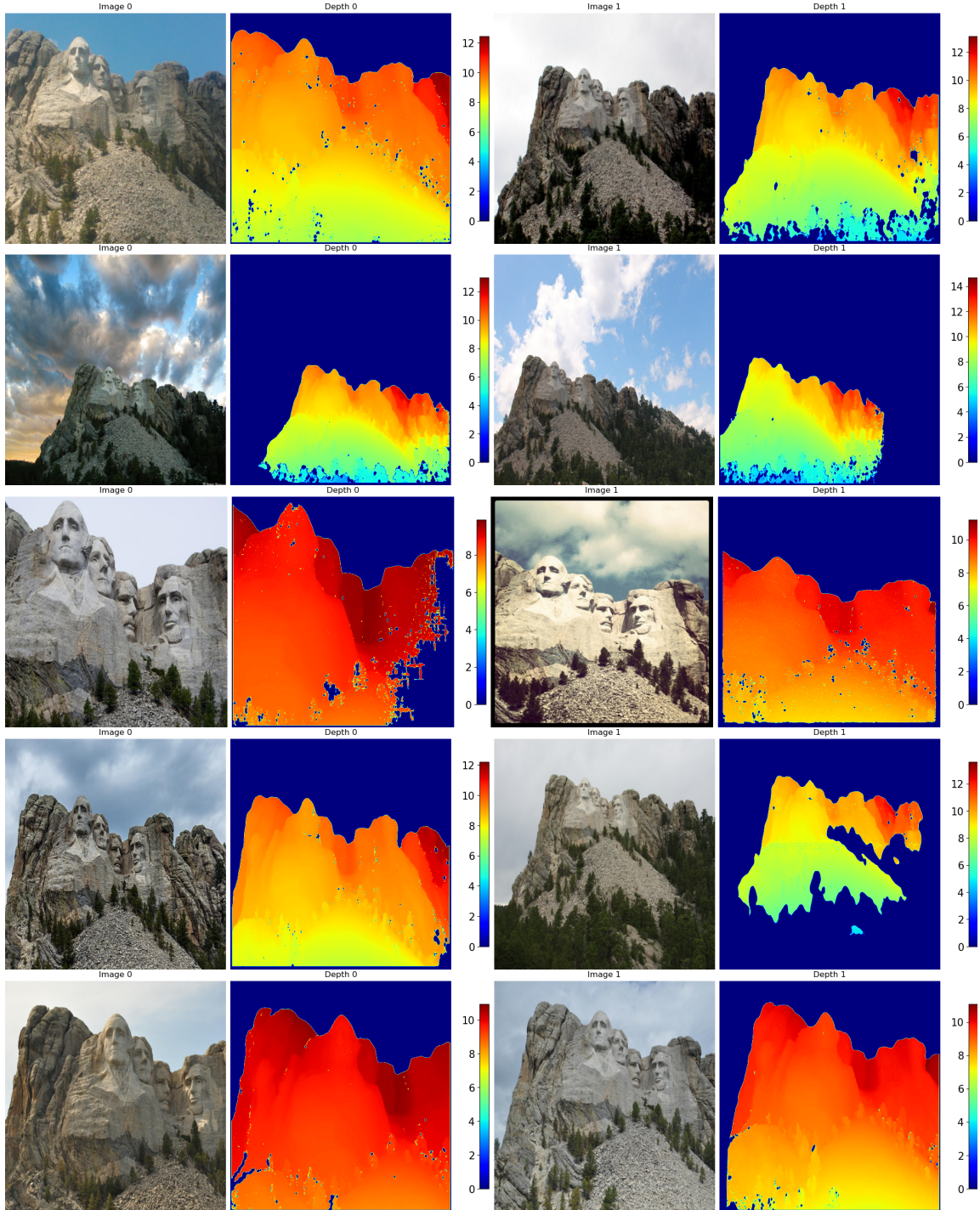


Figure 20: Image and Depth Check Pair

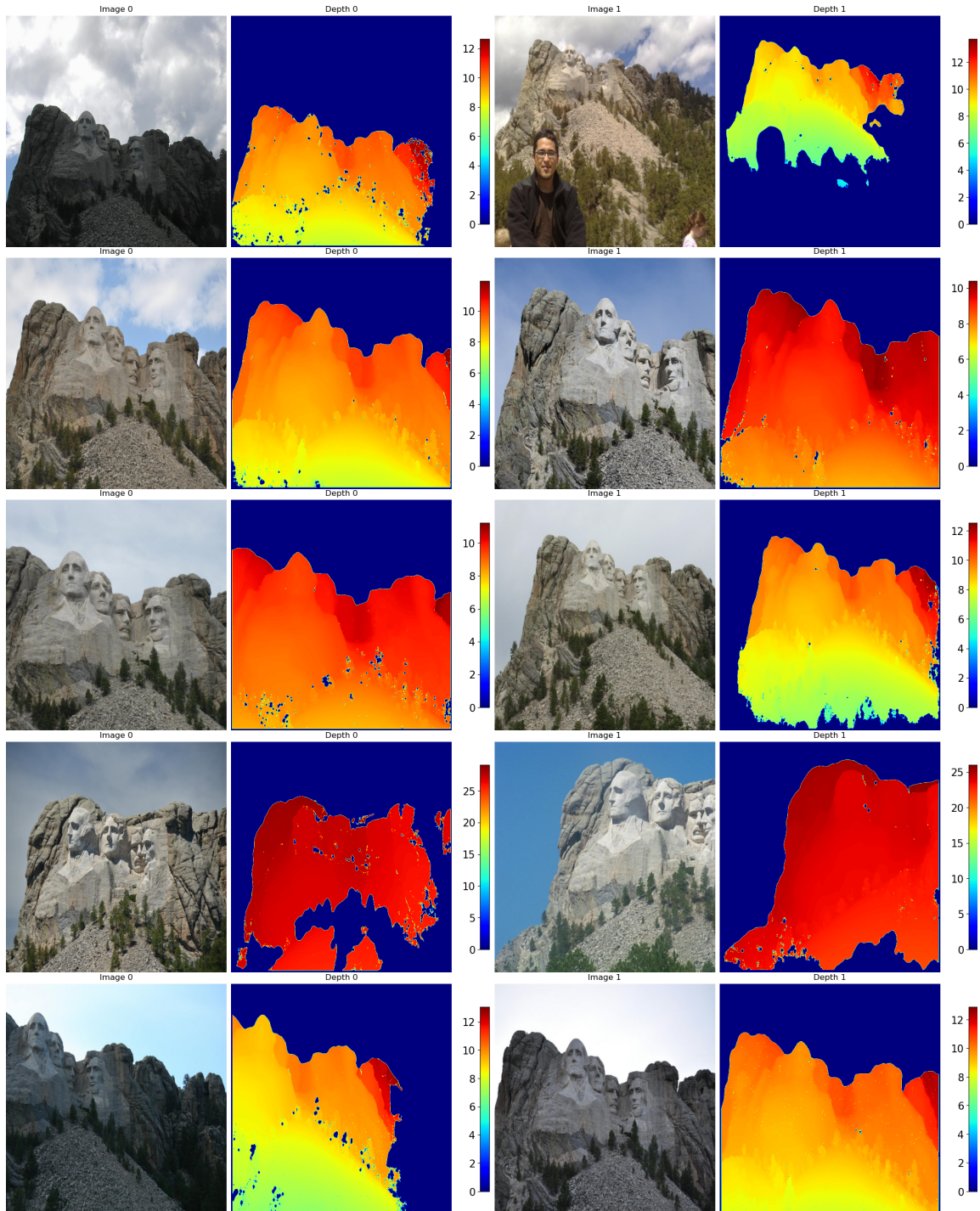


Figure 21: Image and Depth Check Pair

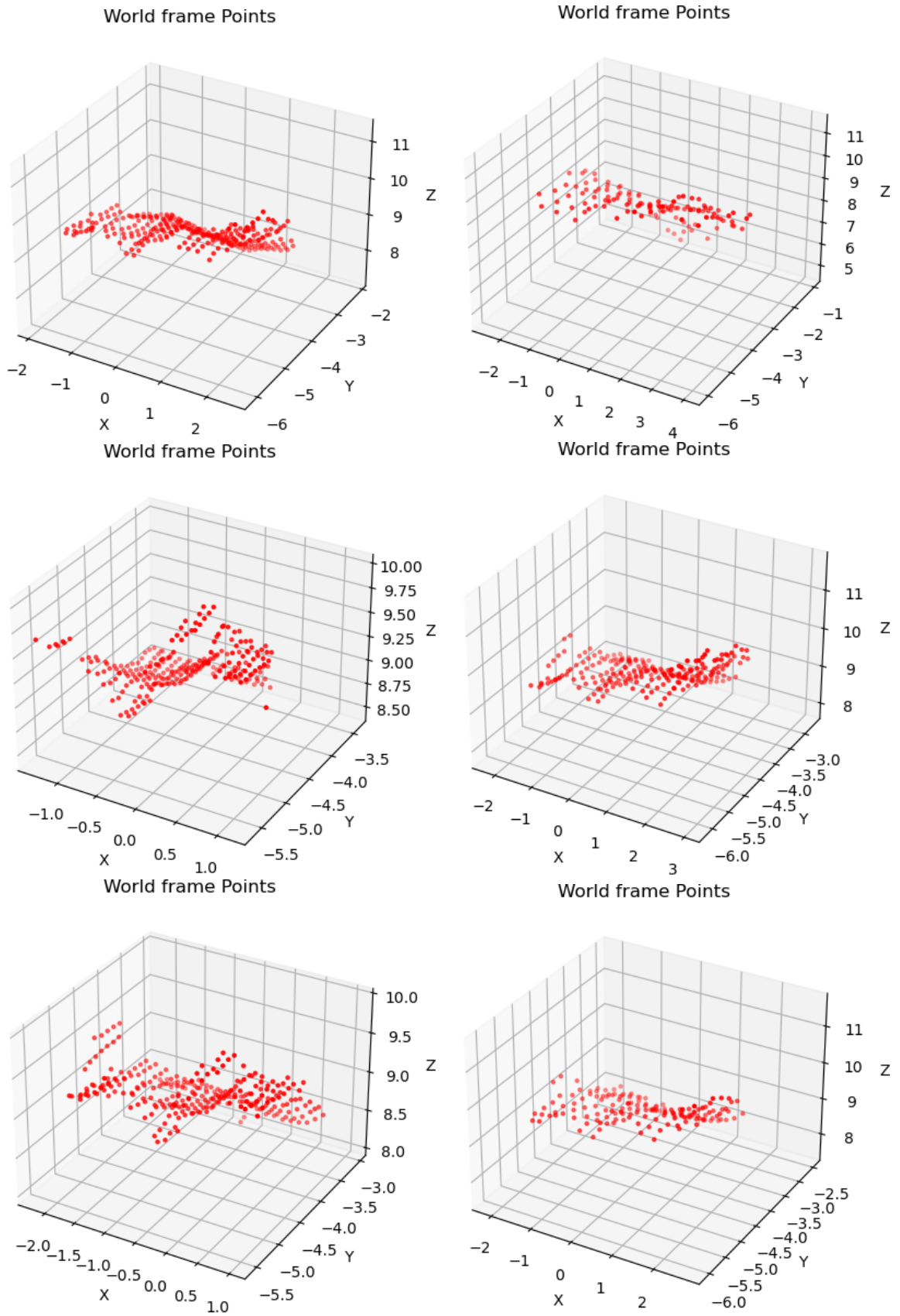


Figure 22: World Frame Points

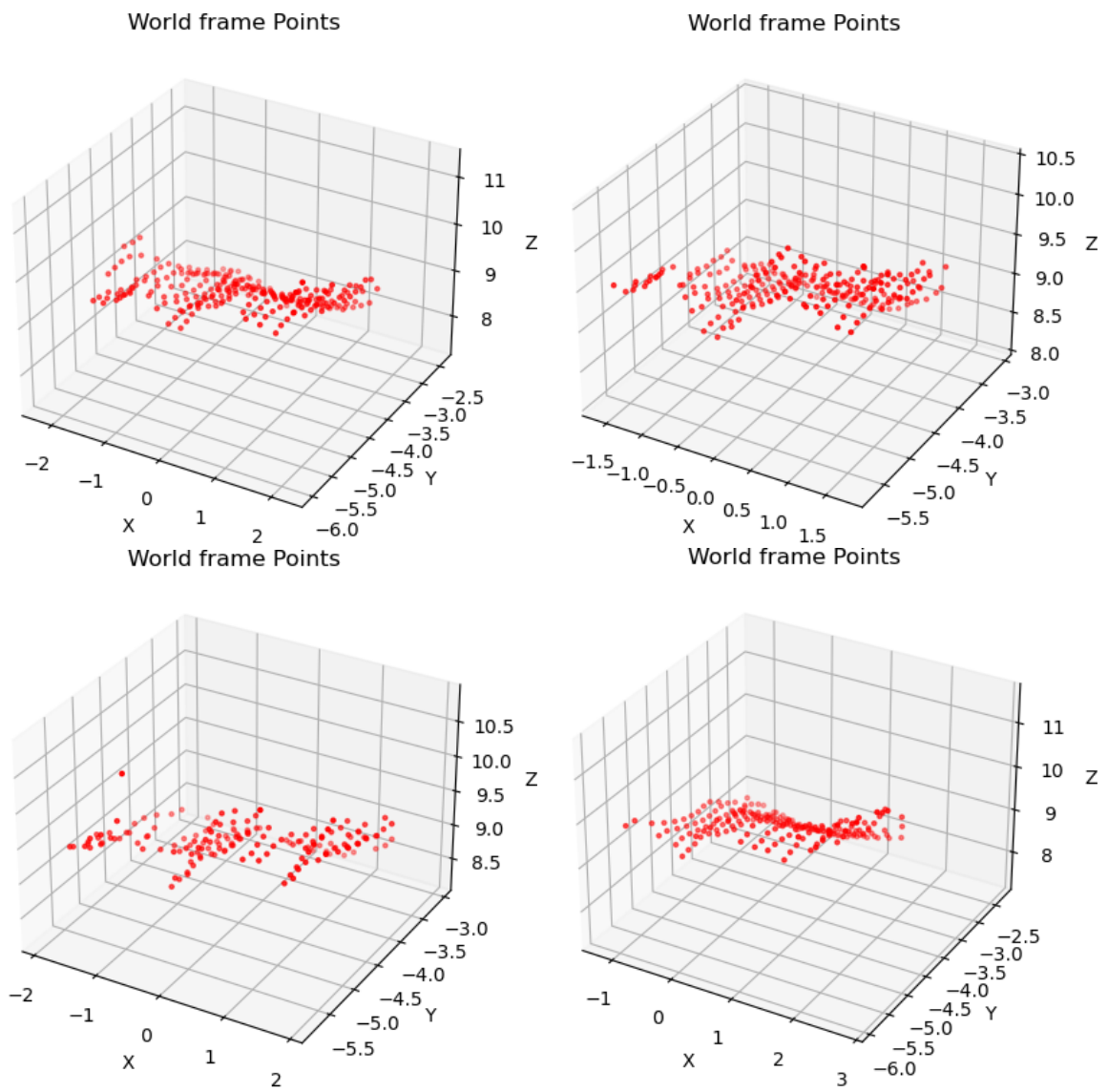


Figure 23: World Frame Points

9 Code

9.1 Task 1

Listing 1: listing copy pasted

```
import numpy as np
import cv2
from scipy.optimize import least_squares
from scipy import ndimage as nd
import random
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.patches import ConnectionPatch

def normalize_pts(points):
    # input: 3XN array of homogeneous points
    # output: array of points
    mean = np.mean(points,axis=1).reshape(3,1) # mean of x and y values
    D_m = np.mean(np.linalg.norm(points-mean,axis=0)) # mean of distances for each point
    s = np.sqrt(2)/D_m # scale factor
    T = np.array([[s,0,-s*mean[0,0]],[0,s,-s*mean[1,0]],[0,0,1]])
    pts_normalized = (points-mean)*s
    return pts_normalized,T

def get_linear_F(x1,x2):
    x1_N,T1 = normalize_pts(x1) # Normalize points
    x2_N,T2 = normalize_pts(x2)
    A = np.zeros((x1_N.shape[1],9))
    for i in range(x1_N.shape[1]):
        A[i,:] = np.array([x2_N[0,i]*x1_N[0,i],
                           x2_N[0,i]*x1_N[1,i],
                           x2_N[0,i],
                           x2_N[1,i]*x1_N[0,i],
                           x2_N[1,i]*x1_N[1,i],
                           x2_N[1,i],
                           x1_N[0,i],
                           x1_N[1,i],
                           1])
    _,_,vh = np.linalg.svd(A)
    F_temp = vh[-1].reshape(3,3) # last row
    u,s,Vh = np.linalg.svd(F_temp)
    s[-1] = 0 # Condition F_temp
    d = np.diag(s)
    F_conditioned = u@d@Vh
    F_denormalized = T2.T@F_conditioned@T1 # Denormalize F_conditioned
    return (F_denormalized/F_denormalized[-1,-1]).reshape(3,3)

def get_epipoles(F):
    u,_,vh = np.linalg.svd(F)
    e1 = (vh[-1].T).reshape(3,1) # last row
    e2 = (u[:, -1]).reshape(3,1) # last column
    return e1/e1[-1], e2/e2[-1]

def get_cross_mat(vec):
    mat_X = np.array([[0,-vec[2,0],vec[1,0]],
                       [vec[2,0],0,-vec[0,0]],
                       [-vec[1,0],vec[0,0],0]])
```

```

    return mat_X

def get_P(F,e2):
    P1 = np.hstack((np.eye(3),np.zeros((3,1))))
    e2_X = get_cross_mat(e2)
    P2 = np.hstack((e2_X@F,e2))
    return P1,P2

def img_to_world(x1,x2,P1,P2):
    P11 = P1[0,:]
    P12 = P1[1,:]
    P13 = P1[2,:]
    P21 = P2[0,:]
    P22 = P2[1,:]
    P23 = P2[2,:]
    X_all = np.zeros((4,x1.shape[1]))
    for i in range(x1.shape[1]):
        x1i = x1[:,i]
        x2i = x2[:,i]
        A = np.array([[x1i[0]*P13-P11],
                      [x1i[1]*P13-P12],
                      [x2i[0]*P23-P21],
                      [x2i[1]*P23-P22]])
        _,_,vh = np.linalg.svd(A.squeeze())
        X = vh[-1] # last row
        X_all[:,i] = X/X[-1]
    return X_all

def world_to_img(P1,P2,X_all):
    x1_hat = P1@X_all
    x2_hat = P2@X_all
    return x1_hat/x1_hat[2,:], x2_hat/x2_hat[2,:]

def get_LM_P(x1,x2,P1,P2_init,X_all_init):
    init_params = np.hstack((P2_init.flatten(),X_all_init[0:3,:].flatten()))
    def cost(params):
        P2 = params[0:12].reshape(3,4)
        X_all = params[12:].reshape(3,-1) # world points
        X_all_homogeneous = np.vstack((X_all, np.ones((1, X_all.shape[1]))))
        x1_hat,x2_hat = world_to_img(P1,P2,X_all_homogeneous)
        cost1 = (x1_hat[0:2,:].T-x1[0:2,:].T).ravel() # ravel flattens the array
        cost2 = (x2_hat[0:2,:].T-x2[0:2,:].T).ravel()
        return np.hstack((cost1,cost2))
    res = least_squares(cost, init_params, method='lm', verbose=1)
    refined_params = res.x # refined parameters
    P1_refined = refined_params[0:12].reshape(3,4)
    X_all_refined = np.vstack((refined_params[12:].reshape(3, -1), np.ones((1, X_all.shape[1]))))
    return P1_refined/P1_refined[-1,-1],X_all_refined

def get_H_prime(img,e2):
    h,w,_ = img.shape
    x0 = w/2
    y0 = h/2
    theta = np.arctan(-(e2[1]-y0)/(e2[0]-x0))
    f = np.abs((e2[0]-x0)*np.cos(theta)-(e2[1]-y0)*np.sin(theta))
    T1 = np.array([[1,0,-x0],[0,1,-y0],[0,0,1]])
    T2 = np.array([[1,0,x0],[0,1,y0],[0,0,1]])
    R = np.array([[np.cos(theta),-np.sin(theta),0],

```



```

        [np.sin(theta), np.cos(theta), 0],
        [0, 0, 1]])
G = np.array([[1, 0, 0], [0, 1, 0], [-1/f, 0, 1]])
H2 = T2@G@R@T1
return H2/H2[-1, -1]

def get_H(P1, P2, H2, x1, x2):
    P1_pinv = P1.T@(np.linalg.inv(P1@P1.T))
    M = P2@P1_pinv
    H_0 = H2@M
    x1_hat = H_0@x1
    x1_hat_homogeneous = x1_hat/x1_hat[2, :]
    x2_hat = H2@x2
    x2_hat_homogeneous = x2_hat/x2_hat[2, :]
    A = x1_hat_homogeneous.T
    b = x2_hat_homogeneous[0, :].T
    H_a = np.eye(3)
    H_a[0, :] = (np.linalg.inv(A.T@A)@A.T)@b
    H1 = H_a@H_0
    return H1/H1[-1, -1]

def apply_homography(img, H):
    h, w, _ = img.shape
    x = np.arange(w) # [W]
    y = np.arange(h) # [H]
    X, Y = np.meshgrid(x, y) # X: [H,W], Y:[H,W]
    X = X.flatten() # [H*W]
    Y = Y.flatten() # [H*W]
    X_hc = np.stack((X, Y, np.ones_like(X))) # coordinates of domain image
    X_hc_p = H @ X_hc # coordinates of range image
    X_hc_p /= (X_hc_p[2, :]) # normalize
    return X_hc, X_hc_p

def interpolate(sparse: np.ndarray):
    height, width, _ = sparse.shape
    mask = np.zeros((height, width), dtype=bool)
    collapse = np.sum(sparse, axis=2)
    for i, r in enumerate(collapse):
        nz = np.where(r != 0)[0] # find nearest neighbors
        if nz.size > 0:
            fi, la = nz[0], nz[-1]
            mask[i, fi:la] = True
    targets = mask & (collapse == 0)
    ind = nd.distance_transform_edt(targets, return_distances=False, return_indices=True)
    sparse = sparse[tuple(ind)]
    return sparse

def get_rectified_img(img1, img2, H1, H2):
    X1_hc, X1_hc_p = apply_homography(img1, H1)
    X2_hc, X2_hc_p = apply_homography(img2, H2)
    min_x, min_y, _ = np.min(np.hstack((X1_hc_p, X2_hc_p)), axis=1)
    max_x, max_y, _ = np.max(np.hstack((X1_hc_p, X2_hc_p)), axis=1)
    h = int(max_y - min_y) # height and width of rectified image
    w = int(max_x - min_x)
    X1_hc_p[0, :] -= min_x # shift so no negative values for either images
    X2_hc_p[0, :] -= min_x
    X1_hc_p[1, :] -= min_y
    X2_hc_p[1, :] -= min_y

```

```

rectified_img_1_sparce = np.zeros((h,w,3),dtype='uint8')
rectified_img_2_sparce = np.zeros((h,w,3),dtype='uint8')

num1_x , num1_y, _ = rectified_img_1_sparce.shape
for i in range(X1_hc_p.shape[1]):
    X = X1_hc[:,i]
    X_p = X1_hc_p[:,i]
    if 0<=int(X_p[1]) <num1_x and 0<=int(X_p[0]) < num1_y:
        rectified_img_1_sparce[int(X_p[1]), int(X_p[0])] = img1[int(X[1]), int(X[0])]
for i in range(X2_hc_p.shape[1]):
    X = X2_hc[:,i]
    X_p = X2_hc_p[:,i]
    if 0<=int(X_p[1]) <num1_x and 0<=int(X_p[0]) < num1_y:
        rectified_img_2_sparce[int(X_p[1]), int(X_p[0])] = img2[int(X[1]), int(X[0])]
rectified_img_1 = interpolate(rectified_img_1_sparce) # fill in the gaps
rectified_img_2 = interpolate(rectified_img_2_sparce)
both = np.hstack((rectified_img_1,rectified_img_2))
cv2.imwrite("Rectified Image 3.jpg", both)
return rectified_img_1,rectified_img_2,both

def canny_interest_points(img,name):
    gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) # convert to gray
    gray_img_blur = cv2.medianBlur(gray_img,7) # median blue to remove noise
    edges = cv2.Canny(gray_img_blur,100,300)
    cv2.imwrite(name, edges)
    return edges

def bound_edge_coords(edges_img1,edges_img2):
    edge_coords1 = []
    edge_coords2 = []
    for i in range(180,530): # hard coded for now
        for j in range(100,550): # bounds for the edges
            if not(i>450 and j<200):
                if not(i>450 and j>400):
                    if edges_img1[i,j] != 0:
                        edge_coords1.append((i,j))
    for i in range(192,530):
        for j in range(105,550):
            if not(i<210 and j<140):
                if edges_img2[i,j] != 0:
                    edge_coords2.append((i,j))
    return edge_coords1,edge_coords2

def distance_metric(edge_coords1,edge_coords2,img1,img2,N,dist_type):
    gray_img1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
    gray_img2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)
    M = N//2 # half of window size
    edge_coords1_sorted = []
    edge_coords2_sorted = []
    for edge1 in edge_coords1:
        neigh1 = gray_img1[edge1[1]-M:edge1[1]+M,edge1[0]-M:edge1[0]+M] # window ar
        err = 10 # check within 10 rows
        row1,col1 = edge1
        temp_edge2 = []
        temp_dist = []
        for edge2 in edge_coords2:
            row2,col2 = edge2
            if abs(row1-row2) <= err:

```

```

        neigh2 = gray_img2[edge2[1]-M:edge2[1]+M,edge2[0]-M:edge2[0]+M]
        if neigh2.shape==neigh1.shape and neigh1.size != 0:
            temp_edge2.append(edge2)
            if dist_type == 'NCC':
                m1 = np.mean(neigh1)
                m2 = np.mean(neigh2)
                num = np.sum((neigh1-m1)*(neigh2-m2))
                den = np.sqrt(np.sum((neigh1-m1)**2)*np.sum((neigh2-m2)**2))
                temp_dist.append(num/(den+0.0001))
            else:
                temp_dist.append(np.sum((neigh1-neigh2)**2))
        if temp_edge2:
            edge_coords2_sorted.append(temp_edge2[np.argmin(temp_dist)])
            edge_coords1_sorted.append(edge1)
    return edge_coords1_sorted,edge_coords2_sorted

def show_correspondence(img1,img2,edge_coords1,edge_coords2,name,dist_type):
    h,w,_ = img1.shape
    img_combined = np.hstack((img1,img2))
    for i,edge1 in enumerate(edge_coords1):
        edge2 = (edge_coords2[i][1]+w,edge_coords2[i][0])
        color = (random.randint(0, 255), random.randint(0, 255), random.randint(0, 255))
        cv2.circle(img_combined, (edge1[1],edge1[0]), 3, color,color,thickness=-1)
        cv2.circle(img_combined, edge2, 3, color=color,thickness=-1)
        cv2.line(img_combined, (edge1[1],edge1[0]),edge2, color=color,thickness=1)
    cv2.imwrite(dist_type+name, img_combined)
    return

def reconstruction_auto(P1,P2,x1,x2):
    X_all = img_to_world(x1,x2,P1,P2)
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(X_all[0, :], X_all[1, :], X_all[2, :], c='r', marker='.')
    plt.show()
    return

def task1():
    pts1 =[(211.3, 132.7),(163.7, 122.4),(354, 252.7),(182.7, 168),(179, 185),(444.2, 89.8),
    (185.9, 158.1),(414.5, 160.3),(234.0, 87.8),(422.5, 170.6),(173.3, 243.9),(243.9, 69.5),
    (308.0, 67.9),(350.1, 255.8),(204.1, 241.8),(199.2, 234.8),(213.4, 116.0),(243.9, 89.9),
    (243.0, 138.1),(454.3, 202.4),(310.9, 86.4),(458.5, 214.3),(213.4, 213.4),(243.9, 86.4),
    (386.2, 86.2),(354.5, 272.8),(235.8, 217.9),(232.9, 210.6),(282.0, 107.4),(308.0, 94.0),
    (451.6, 162.6),(242.2, 107.1),(235.2, 120.0),(252.7, 234.7),(436.9, 243.7)]
    x1 = np.hstack((np.array(pts1),np.ones((len(pts1),1))))*T # Homogeneous
    x2 = np.hstack((np.array(pts2),np.ones((len(pts2),1))))*T

    F_linear = get_linear_F(x1,x2) # linear solution for F estimate
    e1,e2 = get_epipoles(F_linear) # epipoles
    P1,P2 = get_P(F_linear,e2) # Projection matrices
    X_all = img_to_world(x1,x2,P1,P2) # World points
    P2_refined,X_refined = get_LM_P(x1,x2,P1,P2,X_all) # Levenberg-Marquardt optimization
    e2_refined = P2_refined[:, -1] # Refined epipole
    e2_refined_X = get_cross_mat(e2_refined.reshape(3,1))
    F_refined = e2_refined_X@P2_refined@np.linalg.pinv(P1) # Refine F

    img1 = cv2.imread('Pic_1.jpg')

```

```

img2 = cv2.imread('Pic_2.jpg')
H2 = get_H_prime(img2,e2_refined) # Right homography for rectification
H1 = get_H(P1,P2_refined,H2,x1,x2) # Left homography for rectification

rect_img_1,rect_img_2,rect_img_comb = get_rectified_img(img1,img2,H1,H2)

edges_img1 = canny_interest_points(rect_img_1,'edges from img1.jpg') # Canny edges
edges_img2 = canny_interest_points(rect_img_2,'edges from img2.jpg')

edge_coords1,edge_coords2 = bound_edge_coords(edges_img1,edges_img2) # Extract edge coordinates
dist_type = 'SSD'
edge_coords1_sorted,edge_coords2_sorted = distance_metric(edge_coords1,edge_coords2,dist_type)
show_correspondence(rect_img_1,rect_img_2,edge_coords1_sorted,edge_coords2_sorted)

edge_coords1_sorted_downsampled = edge_coords1_sorted[:50]
edge_coords2_sorted_downsampled = edge_coords2_sorted[:50]
show_correspondence(rect_img_1,rect_img_2,edge_coords1_sorted_downsampled,edge_coords2_sorted_downsampled)
np.save('edge_coords1_sorted.npy',edge_coords1_sorted)
np.save('edge_coords2_sorted.npy',edge_coords2_sorted)

edge_coords1_rect = np.load('edge_coords1_sorted.npy')
edge_coords2_rect = np.load('edge_coords2_sorted.npy')

x1_rect = np.hstack((np.array(edge_coords1_rect),np.ones((len(edge_coords1_rect),1))))
x2_rect = np.hstack((np.array(edge_coords2_rect),np.ones((len(edge_coords2_rect),1))))

x1_original = H1@x1_rect
x2_original = H2@x2_rect

x1_original = x1_original/x1_original[2,:]
x2_original = x2_original/x2_original[2,:]
reconstruction_auto(P1,P2_refined,x1_original,x2_original)
return

task1()

```

9.2 Task 3

Listing 2: listing copy pasted

```

import numpy as np
import cv2
import matplotlib.pyplot as plt
from tqdm import tqdm

def disparity_map(imgL,imgR,M,d_max):
    M_2 = M//2 # half of window size
    h,w = imgL.shape
    disparity_map = np.zeros_like(imgL)
    for i in tqdm(range(M_2,h-M_2)):
        for j in range(M_2,w-M_2):
            windowL = imgL[i-M_2:i+M_2+1,j-M_2:j+M_2+1] # window in left image
            bit_vecL = (windowL>imgL[i,j]).ravel() # bit vector for left image pixel
            bit_count = []
            for d in range(min(j-M_2,d_max)):
                windowR = imgR[i-M_2:i+M_2+1,j-M_2-d:j+M_2+1-d] # window in right image
                bit_vecR = (windowR>imgR[i,j-d]).ravel() # bit vector for right image pixel

```



```

        bit_count.append(np.sum(bit_vecL ^ bit_vecR)) # xor operation summed
    if bit_count:
        disparity_map[i,j] = np.argmin(bit_count)
return disparity_map

def show_disparity_results(disparity_map,name):
    dis_map_normalized = cv2.normalize(disparity_map, None, alpha=0, beta=255, norm_type=cv2.NORM_MINMAX)
    dis_map_uint8 = dis_map_normalized.astype(np.uint8)
    cv2.imwrite(name, dis_map_uint8)
    return

def get_binary_error(ground_truth,disparity_map,name):
    h,w = ground_truth.shape
    error_map = np.zeros_like(ground_truth)
    count = 0
    N = cv2.countNonZero(ground_truth)
    for i in range(h):
        for j in range(w):
            err = abs(ground_truth[i,j]-disparity_map[i,j])
            if err <= 2:
                error_map[i,j] = 255
                count+=1
    accuracy = count/N
    cv2.imwrite(name, error_map)
    return error_map,accuracy

def plot_accuracies(window_sizes, accuracies):
    plt.figure(figsize=(10, 6))
    plt.plot(window_sizes, accuracies, marker='o', linestyle='-', color='b')
    plt.xlabel('Window Size')
    plt.ylabel('Accuracy')
    plt.title('Accuracy over Different Window Sizes')
    plt.grid(True)
    plt.show()

imgL = cv2.cvtColor(cv2.imread('Task3Images/im2.png'),cv2.COLOR_BGR2GRAY)
imgR = cv2.cvtColor(cv2.imread('Task3Images/im6.png'),cv2.COLOR_BGR2GRAY)
ground_truth_32 = cv2.cvtColor(cv2.imread('Task3Images/disp2.png'),cv2.COLOR_BGR2GRAY)
ground_truth = (ground_truth_32/4).astype(np.uint8)
d_max = np.max(ground_truth)

window_sizes = [3,5,7,9,11,13,15,17,19]
for size in window_sizes:
    disparity_map = disparity_map(imgL,imgR,size,d_max)
    np.save(f'Disparity map M={size}.npy',disparity_map)
    disparity_map = np.load(f'Disparity map M={size}.npy')
    show_disparity_results(disparity_map,f'Disparity map M={size}.png')
    error_map,accuracy=get_binary_error(ground_truth,disparity_map,f'error_map M={size}.png')
    print(f'Accuracy for M={size} = {accuracy}')

# Accuracies extracted from previous results
accuracies = [0.20563185383386093,0.40367129762859644,0.4829813679808475,0.508507785]
plot_accuracies(window_sizes, accuracies)

```

9.3 Task 4

Listing 3: listing copy pasted

```

import numpy as np
import pickle as pkl
import matplotlib.pyplot as plt
import h5py # for reading depth maps
"""
A few notes on the scene_info dictionary:
- depth maps are stored as h5 files. Depth is the distance of the object from the camera.
- The intrinsics are stored as a 3x3 matrix.
- The poses [R,t] are stored as a 4x4 matrix to allow for easy transformation of points.
  T = [[R, t]
        [0, 1]] where R is a 3x3 rotation matrix and t is a 3x1 translation vector.
"""

DEPTH_THR = 0.1

def plot_image_and_depth(img0, depth0, img1, depth1, plot_name):
    # Enable constrained layout for uniform subplot sizes
    fig, ax = plt.subplots(1, 4, figsize=(20, 5), constrained_layout=True)

    # Image 0
    ax[0].imshow(img0, aspect='auto')
    ax[0].set_title('Image 0')
    ax[0].axis('off')

    # Depth 0
    im1 = ax[1].imshow(depth0, cmap='jet', aspect='auto')
    ax[1].set_title('Depth 0')
    ax[1].axis('off')
    cbar1 = fig.colorbar(im1, ax=ax[1], shrink=0.8, aspect=20)
    cbar1.ax.yaxis.set_ticks_position('left')
    cbar1.ax.yaxis.set_label_position('left')
    cbar1.ax.tick_params(labelsize=15)

    # Image 1
    ax[2].imshow(img1, aspect='auto')
    ax[2].set_title('Image 1')
    ax[2].axis('off')

    # Depth 1
    im2 = ax[3].imshow(depth1, cmap='jet', aspect='auto')
    ax[3].set_title('Depth 1')
    ax[3].axis('off')
    cbar2 = fig.colorbar(im2, ax=ax[3], shrink=0.8, aspect=20)
    cbar2.ax.yaxis.set_ticks_position('left')
    cbar2.ax.yaxis.set_label_position('left')
    cbar2.ax.tick_params(labelsize=15)

    plt.savefig(plot_name, bbox_inches='tight', pad_inches=0)
    plt.close()

if __name__ == "__main__":
    scene_info = pkl.load(open('./data/scene_info/1589_subset.pkl', 'rb'))

    for i_pair in range(len(scene_info)):
        # print(scene_info[i_pair].keys())
        # ['image0', 'image1', 'depth0', 'depth1', 'K0', 'K1', 'T0', 'T1', 'overlap_score']

```

```

# print(scene_info[i_pair]['image0']) # path to image0
# print(scene_info[i_pair]['image1']) # path to image1
# print(scene_info[i_pair]['depth0']) # path to depth0
# print(scene_info[i_pair]['depth1']) # path to depth1
# print(scene_info[i_pair]['K0']) # intrinsic matrix of camera 0 [3,3]
# print(scene_info[i_pair]['K1']) # intrinsic matrix of camera 1 [3,3]
# print(scene_info[i_pair]['T0']) # pose matrix of camera 0 [4,4]
# print(scene_info[i_pair]['T1']) # pose matrix of camera 1 [4,4]
# print('-----')

# read images
img0 = plt.imread(scene_info[i_pair]['image0'])
img1 = plt.imread(scene_info[i_pair]['image1'])

# read depth
with h5py.File(scene_info[i_pair]['depth0'], 'r') as f:
    depth0 = f['depth'][:]
with h5py.File(scene_info[i_pair]['depth1'], 'r') as f:
    depth1 = f['depth'][:]

# check shapes
h0, w0 = img0.shape[: -1]
h1, w1 = img1.shape[: -1]
assert img0.shape[: -1] == depth0.shape, f"depth and image shapes do not match"
assert img1.shape[: -1] == depth1.shape, f"depth and image shapes do not match"

# plot image and depth
plot_name = f'./pics/image_and_depth_pair_{i_pair}.png'
plot_image_and_depth(img0, depth0, img1, depth1, plot_name)

#(1) make meshgrid of points in image 0
density = 20
x = np.linspace(10, img0.shape[1]-10, density)

#<Student code>
# meshgrid of x and y coordinates
y = np.linspace(10, img0.shape[0] - 10, density)
xx, yy = np.meshgrid(x, y)

# make homogeneous coordinates for points0 #[3, N]

# <Student code>
xxF, yyF = xx.flatten(), yy.flatten()
points0 = np.vstack((xxF, yyF, np.ones((xxF.shape[0],))))

#(2) get depth values at points0
"""
<Student code>
depth_values0 = ...
"""
depth_values0 = depth0[yyF.astype(int), xxF.astype(int)]
# remove points with depth 0 (invalid points)
valid_points = np.where(depth_values0 != 0)[0]
points0 = points0[:, valid_points]
depth_values0 = depth_values0[valid_points]

"""
<Student code>

```

```

valid_points = ...
# mask points0 and depth_values0
"""

# (3) Find the 3D coordinates of these points in camera 0 frame
K0 = scene_info[i_pair]['K0'] # [3,3]
T0 = scene_info[i_pair]['T0'] # [4,4]

# inverse of K0
K0_inv = np.linalg.inv(K0)
# convert points0 to camera coordinates
xyz_cam0 = K0_inv @ points0
# normalize xyz_cam0 to set z = 1 (sanity check)
xyz_cam0 /= xyz_cam0[2]
# get the point at depth
xyz_cam0 = xyz_cam0 * depth_values0

# make homogeneous coordinates [4,N]
xyz_cam0_hc = np.vstack((xyz_cam0, np.ones_like(xyz_cam0[0])))
# convert to world frame [4,N]
xyz_world_hc = np.linalg.inv(T0) @ xyz_cam0_hc

# (4) Transform these points to camera 1 frame
T1 = scene_info[i_pair]['T1']
# <Student code>
# transform points to camera 1 frame
xyz_cam1_hc = T1 @ xyz_world_hc
# convert to camera 1 coordinates
xyz_cam1 = xyz_cam1_hc[:3] / xyz_cam1_hc[3]
# get z coordinates for depth check
estimated_depth_values1 = xyz_cam1[2]

# project to image 1

# <Student code>
points1 = scene_info[i_pair]['K1'] @ xyz_cam1
# normalize by dividing by last row
points1 = points1 / points1[2]
# check if points1 are within image bounds
mask = ((points1[0] > 0) & (points1[0] < w1)) & ((points1[1] > 0) & (points1[1] < h1))
points1 = points1[:, mask]
# get the depth values at these points using the depth map
true_depth_values1 = estimated_depth_values1[mask]
true_depth_values1 = depth1[points1[1].astype(int), points1[0].astype(int)]
# (5) plot matching points in image 0 and image 1 with depth check such that
fig, ax = plt.subplots(1, 1, figsize=(10, 5))

# Horizontally stack the images
combined_img = np.ones((max(img0.shape[0], img1.shape[0]), img0.shape[1] + img1.shape[1]), dtype=img0.dtype)
combined_img[:img0.shape[0], :img0.shape[1]] = img0
combined_img[:img1.shape[0], img0.shape[1]:] = img1

ax.imshow(combined_img, aspect='auto')
ax.scatter(xx, yy, c='r', s=5)
ax.set_title('Matching points in Image 0 and Image 1')
ax.axis('off')

# draw lines between matching points

```



```

for i in range(points1.shape[1]):
    # if depth values match
    if np.abs(estimated_depth_values1[i] - true_depth_values1[i]) < DEPTH_TH:
        ax.plot([points0[0,i], points1[0, i] + img0.shape[1]], [points0[1,i], points1[1,i]])

plt.savefig(f'./pics/depth_check_pair_{i_pair}.png', bbox_inches='tight', pad_inches=0.5)
plt.close()
print(f"Done with pair {i_pair}")

# (6) Plot all 3D points for the pair
"""
<Student code>
...
"""
plt.figure()
ax = plt.axes(projection='3d')
ax.scatter(xyz_world_hc[0], xyz_world_hc[1], xyz_world_hc[2], c='r', s=5)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.set_title('World frame Points')
plt.savefig(f'./pics/world_frame_points{i_pair}.png', bbox_inches='tight', pad_inches=0.5)

```