ECE 661 Homework 9

Michael Goldberg

November 25 2024

1 Theoretical Question

1.1 Question 1

Let $\vec{l}$ be the epipolar line in the left camera and $\vec{l'}$ be the epipolar line in the right camera. Let's explore how an image line backprojects into world coordinates. We know that

$$\vec{x^T}\vec{l} = 0 \tag{1}$$

and

$$\vec{x} = P\vec{X} = \vec{X^T}P^T \tag{2}$$

We can therefore rewrite Equation 1 as:

$$\vec{X^T}P^T\vec{l} = 0 \tag{3}$$

This can be further examined to see that $P^T\vec{l}$ forms a plane $\vec{\pi}$ that point $\vec{x}$ must lie on.

However, we know that the cameras are imaging the same point and their center of projections also exist in world3D. This describes this exact plane π that we just derived, but this plane also pierces camera P'. Therefore, we know that both cameras must share the same plane, and by extension $\vec{x'}$ must lie on its corresponding line $\vec{l'}$, as that line and point must also backproject to the same plane.

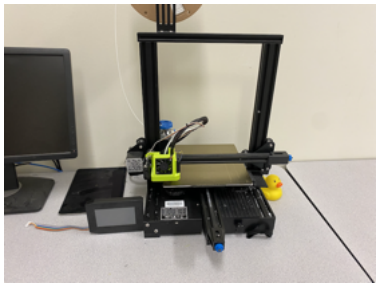
2 Programming Tasks

The following task was to perform stereo rectification on a pair of images to result in an eventual pointcloud of the object being imaged. This was broken down into the following steps:

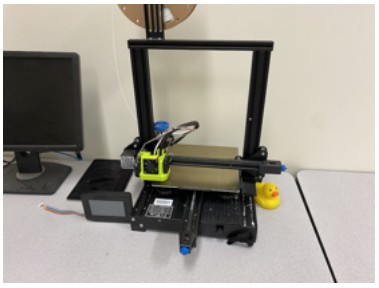
1. 8-Point Correspondence Estimation of F
2. Epipole Estimation
3. Camera Matrix Estimation

4. Refinement of P and P'
5. Refinement of F
6. Estimation of H and H'
7. Interest Point Detection
8. Projective Reconstruction

The following shows the two images used:



(a) Left Stereo Image

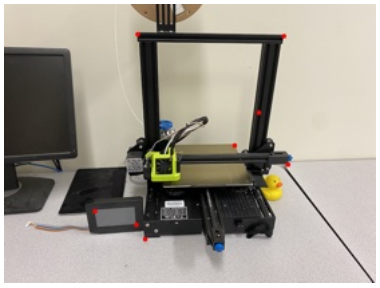


(b) Right Stereo Image

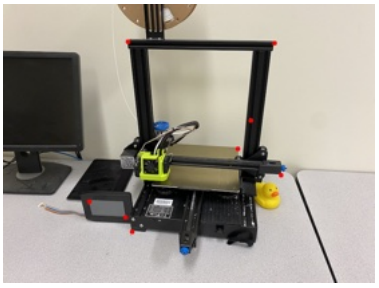
I will now go over how I addressed each task.

2.1 8-Point Correspondence Estimation of F

I first established 8 point-to-point correspondences between the left and right images, which can be seen here:



(a) Left Stereo Image Points



(b) Right Stereo Image Points

The first step is to normalize the image coordinates. From here on, I will refer to the left image points as x and the right image points as x'. This normalization matrix is:

$$T = \begin{bmatrix} s & 0 & -s\bar{x} \\ 0 & s & -s\bar{y} \\ 0 & 0 & 1 \end{bmatrix} \tag{4}$$

and

$$T' = \begin{bmatrix} s' & 0 & -s'\bar{x'} \\ 0 & s' & -s'\bar{y'} \\ 0 & 0 & 1 \end{bmatrix} \tag{5}$$

, where $\bar{x}$ and $\bar{y}$ are the means of the x and y coordinates of the left points and $\bar{x'}$ and $\bar{y'}$ are the means of the x' and y' coordinates of the right points, and s and s' are $\sqrt{2}/\sigma$ and $\sqrt{2}/\sigma'$ respectively.

These normalization matrices can be applied to each set of image points to form x_n and x'_n. We can now construct the equation to solve for F. It must be that $AF = 0$, where A is constructed in the following manner. For each point correspondence $x_{n,i} <-> x'_{n,i}$:

$$\begin{bmatrix} x'_{n,i,x}x_{n,i,x} & x'_{n,i,x}x_{n,i,y} & x'_{n,i,x} & x'_{n,i,y}x_{n,i,x} & x'_{n,i,y}x_{n,i,y} & x'_{n,i,y} & x_{n,i,x} & x_{n,i,y} & 1 \end{bmatrix} \begin{bmatrix} F_{11} \\ F_{12} \\ F_{13} \\ F_{21} \\ F_{22} \\ F_{23} \\ F_{31} \\ F_{32} \\ F_{33} \end{bmatrix} = 0 \tag{6}$$

Since we have 8 unknowns (due to ratios) we must have 8 correspondences added to A. The solution to this is therefore the SVD decomposition of A to get the elements of F. However, this needs to be conditioned to be of rank 2. This can be done by performing SVD on F and setting its smallest singular value to 0.

2.2 Epipole Estimation

Once F is estimated, we can calculate the epipoles. We know that these must satisfy the following conditions:

$$Fe = 0 \tag{7}$$

and

$$e'F = 0 \tag{8}$$

This can again be done through SVD with F for e and F^T for e'.

2.3 Camera Matrix Estimation

Once the fundamental matrix and the epipoles have been estimated, the camera matrices P and P' can be estimated. We assume canonical form, thereby setting

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{9}$$

P' can be found through the construction of a skew-symmetric matrix of rank 3 derived from e':

$$s = \begin{bmatrix} 0 & -e'_2 & e'_1 \\ e'_2 & 0 & -e'_0 \\ -e'_1 & e'_0 & 0 \end{bmatrix} \tag{10}$$

P' can then be found through: $P' = (sF|e')$

2.4 Refinement of P and P'

We now want to refine our estimation for P'. This can be done using Levenberg-Marqardt (LM) with the cost function being defined as the squared euclidean distance between the reprojected point and the original point correspondence. This is achieved through triangulation in the following manner: Suppose

$$A = \begin{bmatrix} x[0]P[2] - P[0] \\ x[1]P[2] - P[1] \\ x'[0]P'[2] - P'[0] \\ x'[1]P'[2] - P'[1] \end{bmatrix} \tag{11}$$

The SVD of A results in the triangulated world3D coordinate X of x and x'.

Then,

$$x_{reprojected} = PX \tag{12}$$

and

$$x'_{reprojected} = P'X \tag{13}$$

LM optimizes P' to minimize the reprojection error.

2.5 Refinement of F

Once P' is refined, the refined version of F can be calculated again through the help of the skew-symmetric matrix s. F is simply defined as:

$$F = sP'_{refined}P^{+} \tag{14}$$

This F needs to again be conditioned to be rank 2, and the refined epipoles can be calculated through finding the left and right null vectors of the refined F as seen from before.

2.6 Estimation of H and H'

H and H' can now be found. The first step is to transform the image center to the origin through the following homography:

$$T_1 = \begin{bmatrix} 1 & 0 & -w/2 \\ 0 & 1 & -h/2 \\ 0 & 0 & 1 \end{bmatrix} \tag{15}$$

We then want to send the right epipole to $\begin{bmatrix} f & 0 & 1 \end{bmatrix}$. This can be done through the following rotation matrix:

$$R = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{16}$$

where $\theta = -\arctan e'_y/e'_x$

Upon performing Re', we can extract f.

Then, we can send this epipole to infinity along the x-axis through the homography:

$$G = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1/f & 0 & 1 \end{bmatrix} \tag{17}$$

Finally, we want to transform the image center back using

$$T_2 = \begin{bmatrix} 1 & 0 & w/2 \\ 0 & 1 & h/2 \\ 0 & 0 & 1 \end{bmatrix} \tag{18}$$

Finally, $H' = T_2GRT_1$.

Computation of H follows a similar process, however now we perform the above steps using the left epipole and set its result to $\hat{H}$. We then construct A and b matrices in the following manner:

$$A_i = \begin{bmatrix} \hat{H}x_i[0] & \hat{H}x_i[1] & 1 \end{bmatrix} \tag{19}$$

and

$$b_i = \begin{bmatrix} H'x'_i[0] \end{bmatrix} \tag{20}$$

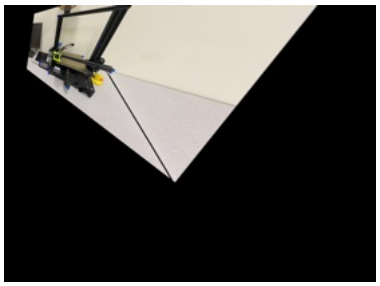
Solving for the minimization, we get:

$$m = A^+ b \tag{21}$$

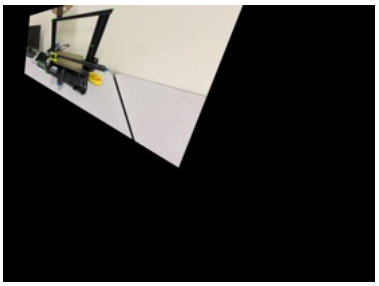
This allows us to set up:

$$H_a = \begin{bmatrix} m[0] & m[1] & m[2] \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{22}$$

Finally, $H = H_a\hat{H}$ Applying these homographies to rectify the images, we are left with the following rectified images:



(a) Left Rectified Image



(b) Right Rectified Image

Additionally, here are the correspondence lines drawn between the two:

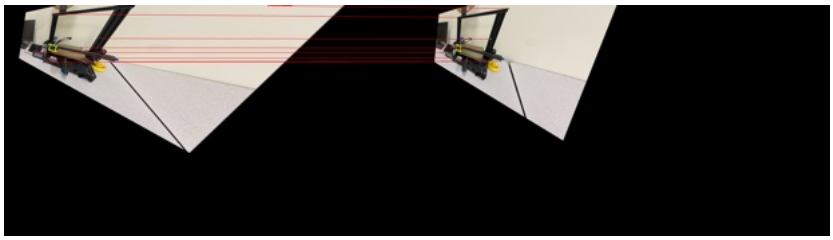
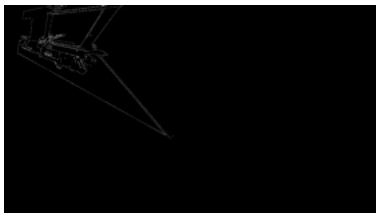


Figure 4: Rectified Images with Correspondence Lines

2.7 Interest Point Detection

Now that the images are rectified, we can continue with interest point detection. First, a canny edge detector was applied with threshold values of 600 and 800. The results of this can be seen below (please zoom in):



(a) Left Canny Detection

(b) Right Canny Detection

The pixel locations of true values could then be extracted and used as point-to-point correspondences to further refine F. This process entailed sweeping a window across each pixel in the left canny image and sweeping a corresponding window across a small region around the same row in the right canny image. The sum of squared differences (SSD) across both windows were computed, and the corresponding center pixel in the right image with the lowest SSD score was deemed to be the left image pixel's corresponding match. Lines were drawn between each correspondence, and can be seen in the dense set below:

Figure 6: Left Rectified Image

2.8 Projective Reconstruction

Finally, these dense correspondences were then used to refine P' once more using the process from above. With the final refined P', the world coordinates of the object could be extracted using the triangulation technique from above as well, resulting in the following 3D pointcloud with projective distortion:

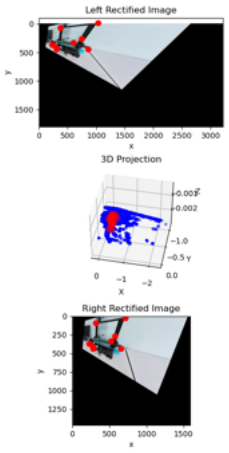
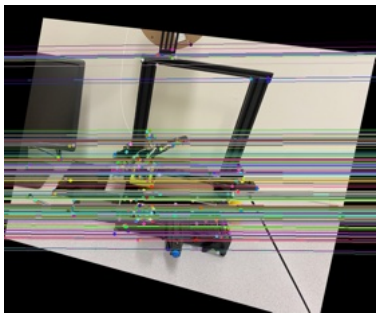


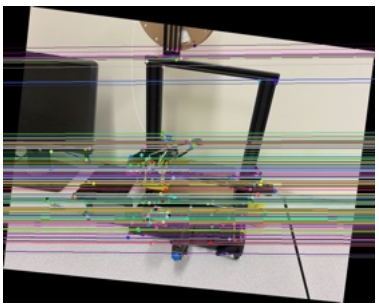
Figure 7: Rectification result, with original 8-point correspondences in Red

3 Loop and Zhang

Loop and Zhang is an image rectification algorithm that removes projective and affine distortion from images. It first uses the vanishing points of orthogonal lines to estimate the projective transformation, and aligns them to be parallel. Then, it enforces perpendicularity between the orthogonal lines, resulting in a rectified image true to the original proportions. Here is the result of Loop and Zhang on my stereo image pair:



(a) Left Image Rectification Loop and Zhang



(b) Right Image Rectification Loop and Zhang

Loop and Zhang has a lot less distortion than my implementation, because that is what it is designed to circumvent. However, this algorithm produces much less correspondences than my implementation does. This may be a good thing, as I did not perform any outlier rejection like RANSAC on my dense correspondences which may lead to inaccuracies. I do like how my implementation combines the two rectified images into a single image so you can really see which points correspond to which other points, something Loop and Zhang does not do.

4 Disparity Map Using Census Transform

This task focused on calculation of disparity between two rectified images. Disparity essentially describes how much any given shared point between two images is shifted along a row in one image compared to another. Large disparity means large shifts and indicates an object in the foreground, while small disparity indicates objects in the background. The census transform is an algorithm to calculate this disparity, and works in the following manner:

- A parent pixel is determined through raster order of the left image and its bitvector is calculated
- In the right image, a window is swept along the image in raster order and its bitvector is calculated (The bitvector is calculated by looking inside of the window and counting how many pixels are strictly larger in intensity than the center pixel)
- The two bitvectors are then XORed together, and the cost is the number of 1s are in the resultant word
- The window is then shifted in the x-direction from 0 to d_{max} and the bitvector and cost are then recalculated. The disparity at the parent point is the pixel shift that results in the least cost between the two bitvectors

I wrote my own implementation of this census transform algorithm and compared it with the ground truth disparity maps given to us. Disparity was deemed accurate if the difference between calculated and ground truth disparity was less than $\delta = 2$. I tested this with a sliding window size of 5x5 and 11x11. The 5x5 sliding window resulted in 40 accuracy, while the 11x11 window resulted in 51 accuracy.

Below are my calculated disparity maps along with the ground truth disparity maps:

(a) 5x5 Left Disparity Map

(b) 11x11 Left Disparity Map

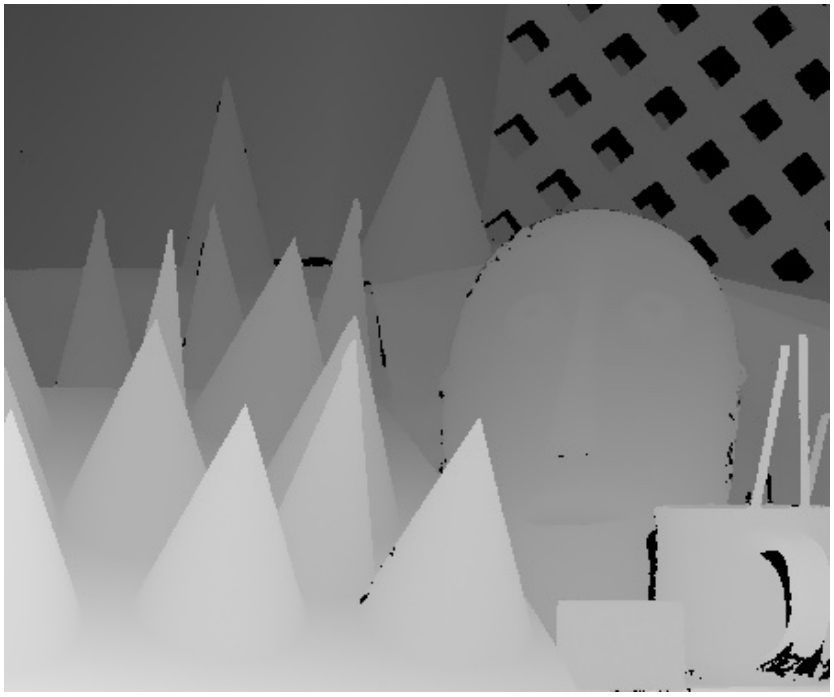
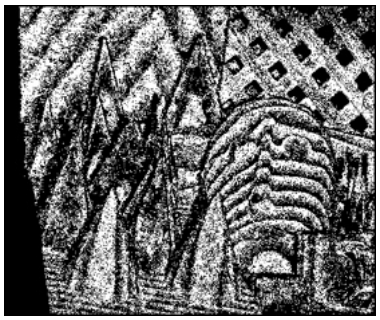
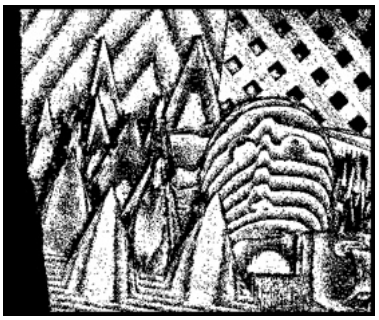


Figure 10: Ground Truth Left Disparity Map

Below are the error masks for these two tests:



(a) 5x5 Left Error Mask



(b) 11x11 Left Error Mask

It seems that increasing the window size makes the disparity map denser, while also increasing the accuracy. However, this increases the processing time as this increases the size of the bitvectors that are calculated.

5 Automatic Extraction of Dense Correspondences Using Depth Maps

This refers to finding corresponding points between two images using just the shared depth data between their depth maps. Essentially, if you image the same object, you can determine the camera's pose from the 3D geometry and determine the corresponding homographies between each image using the 3D geometry as a middle ground to project to. This allows for much more accurate correspondences, as it uses the known 3D geometry to register points to and backproject back to their own camera coordinates. In addition, depth maps are very commonly used in robotics, so this has many useful applications for figuring out which 2 points describe the same object in a stereo pair. This is particularly useful for identifying similar points across varying contexts, for example with images taken at different times or perhaps under inclement weather.

This process works in the following manner:

- Find the inverse of the left camera intrinsic matrix, and apply to each point you want to find a correspondence to $X_{cam,left} = K^{-1}x_{left}$ - this is simply the projection of the pixel along a ray stemming from the camera
- Multiply $X_{cam,left}$ by its corresponding depth value, resulting in $X_{cam,left}$ being transformed to be a 3D point in the camera's coordinate frame
- Convert $X_{cam,left}$ into the world3D coordinate frame using extrinsics
- Using the right camera's extrinsics, we can estimate this world coordinate's location $X_{cam,right}$, which includes its estimated depth $Z_{cam,right}$

- Backproject $X_{cam,right}$ to x_{right} and find the true depth using the right image's depth map
- Compare estimated depth and true depth, see if lie below threshold. If yes, x_{left} and x_{right} are correspondences.

The following shows the successful execution of this process for a variety of images with their corresponding depth maps. Below are the image and depth pairs:

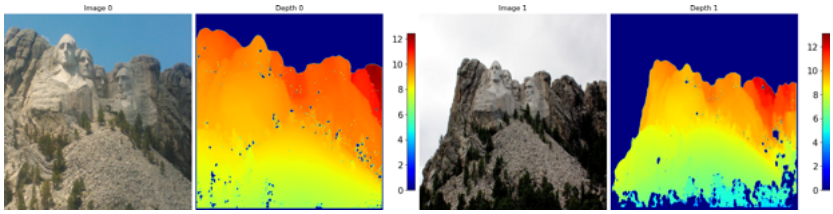


Figure 12: Image and Depth Pair 0

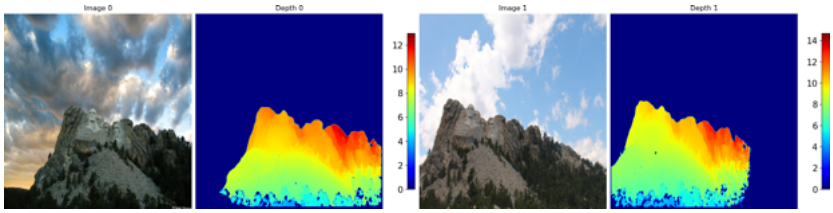


Figure 13: Image and Depth Pair 1

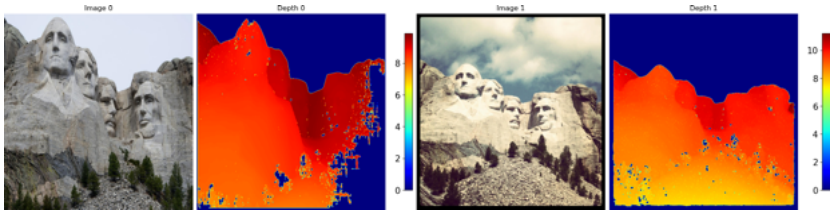


Figure 14: Image and Depth Pair 2

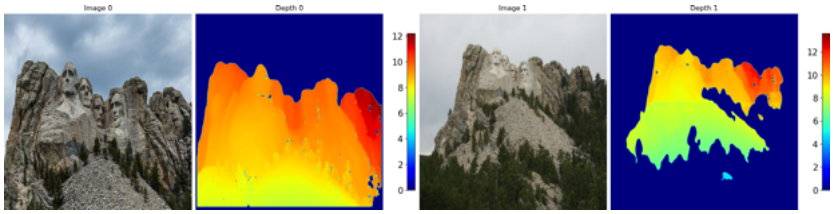


Figure 15: Image and Depth Pair 3

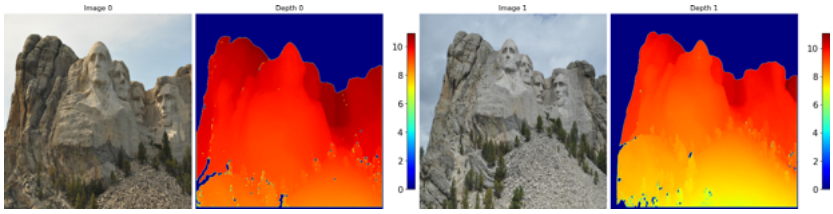


Figure 16: Image and Depth Pair 4

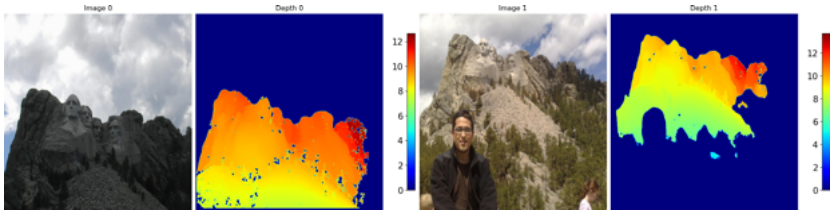


Figure 17: Image and Depth Pair 5

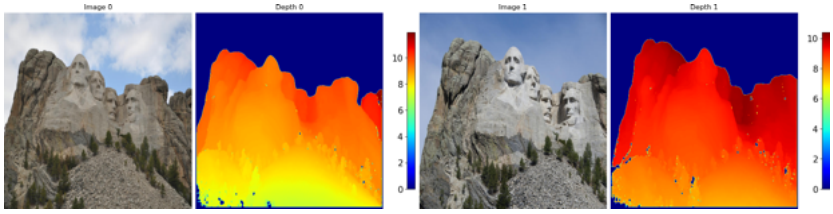


Figure 18: Image and Depth Pair 6

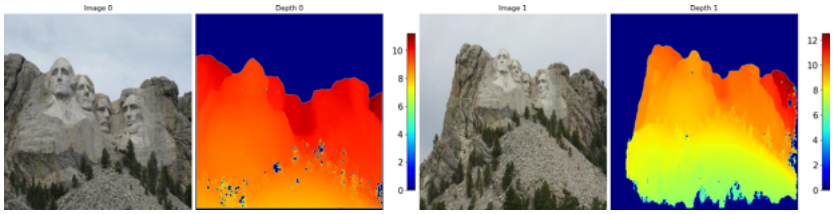


Figure 19: Image and Depth Pair 7

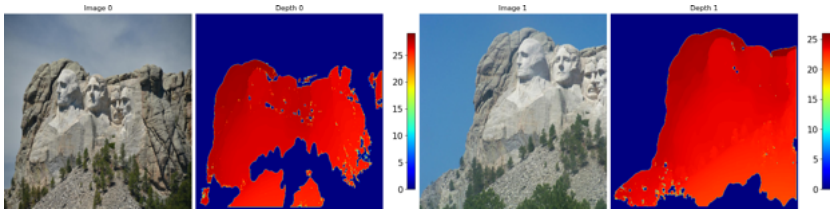


Figure 20: Image and Depth Pair 8

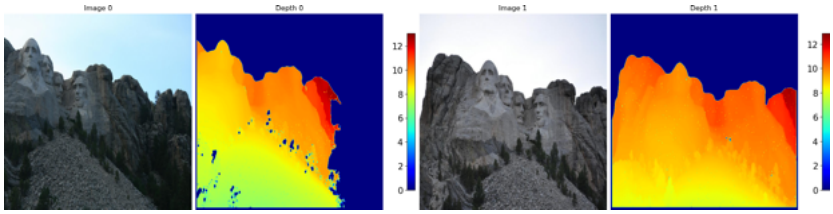


Figure 21: Image and Depth Pair 9

Below are the successfully found correspondences:

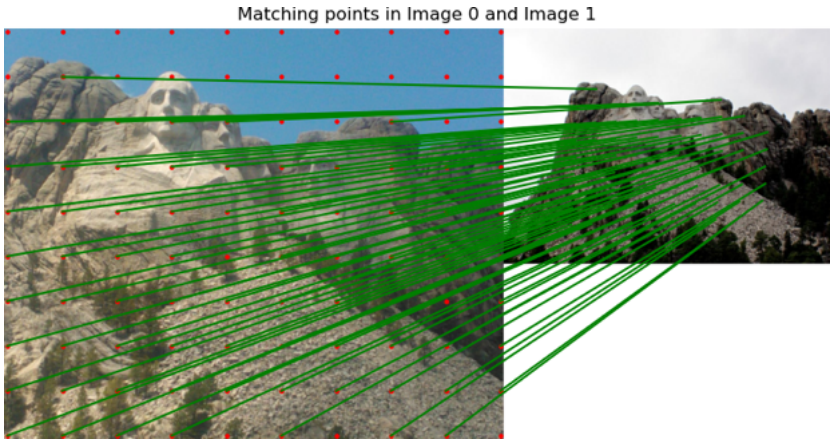


Figure 22: Depth Check Pair 0

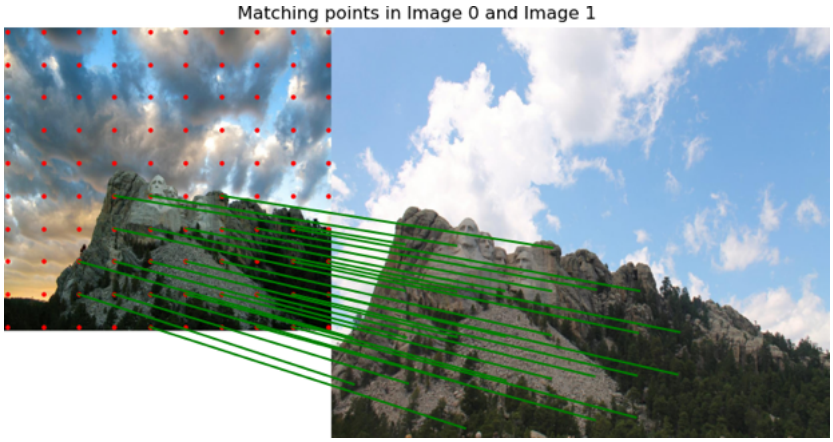


Figure 23: Depth Check Pair 1

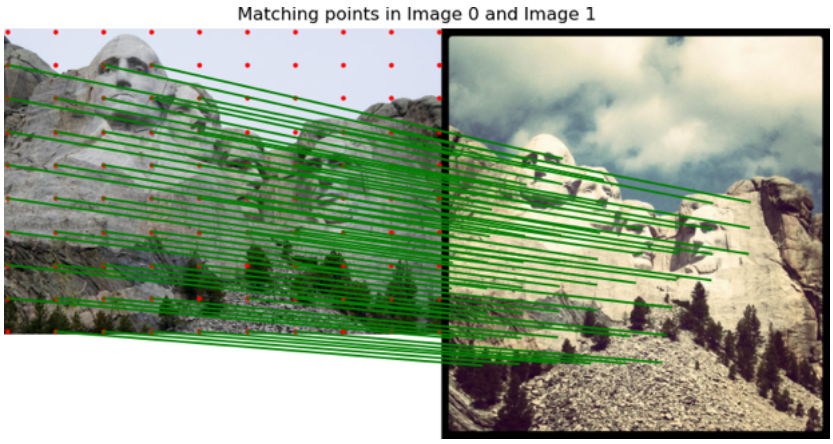


Figure 24: Depth Check Pair 2

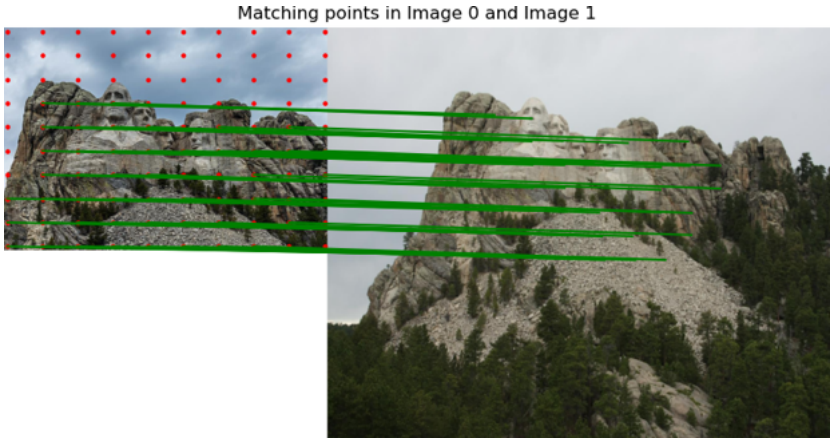


Figure 25: Depth Check Pair 3

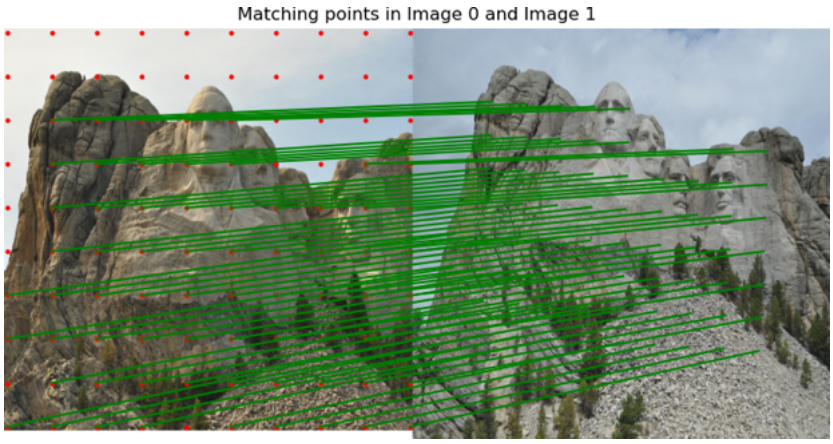


Figure 26: Depth Check Pair 4

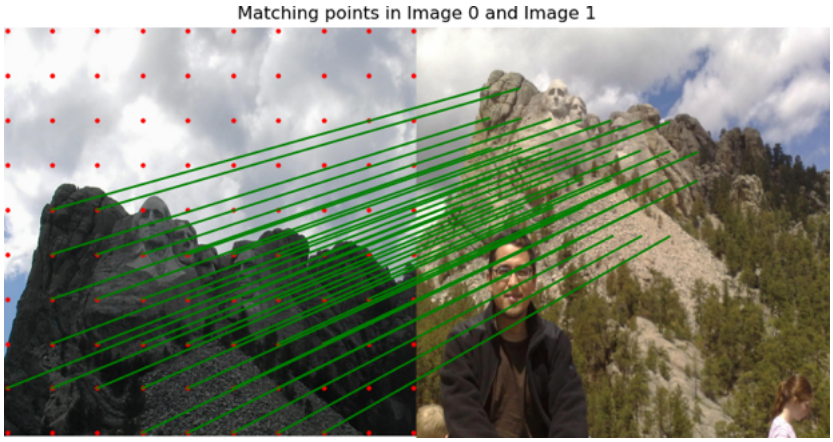


Figure 27: Depth Check Pair 5

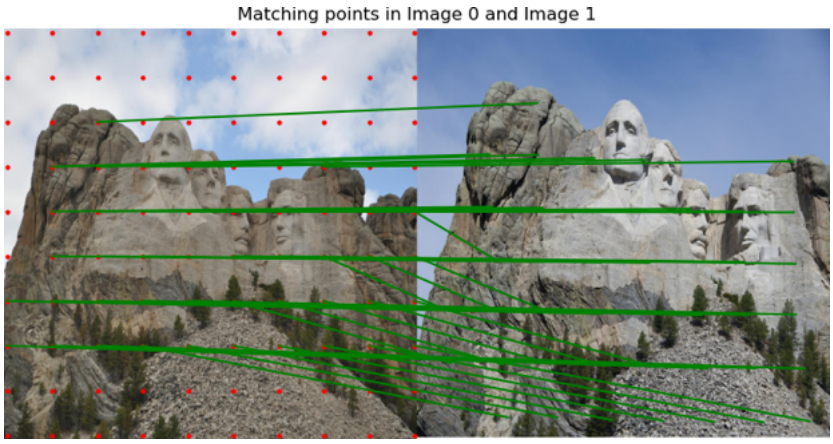


Figure 28: Depth Check Pair 6

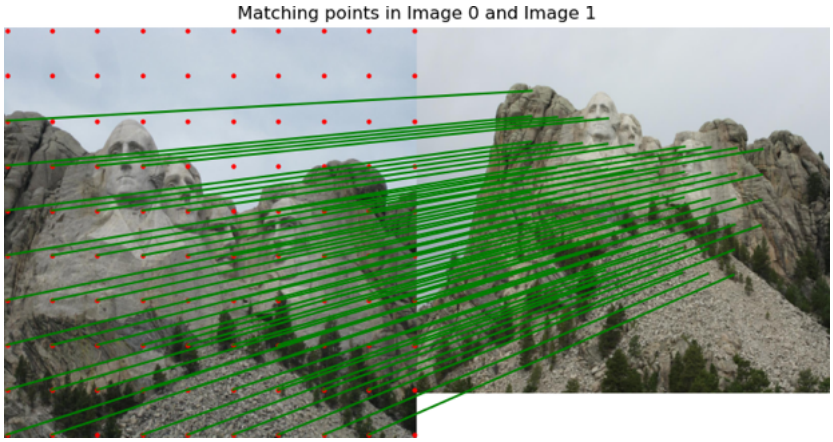


Figure 29: Depth Check Pair 7

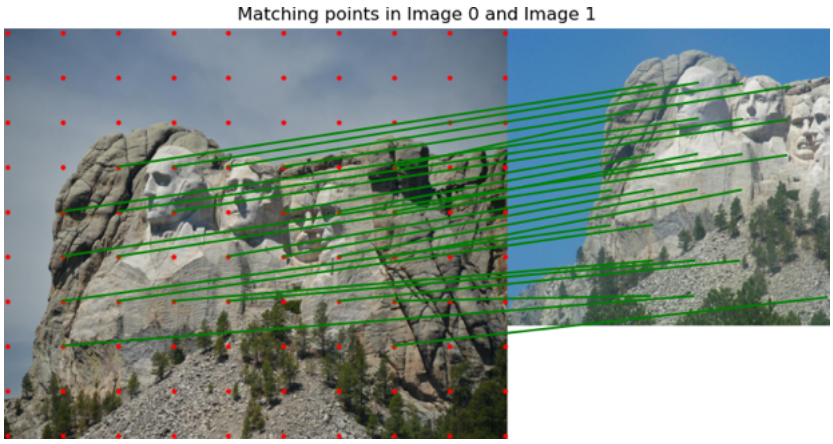


Figure 30: Depth Check Pair 8

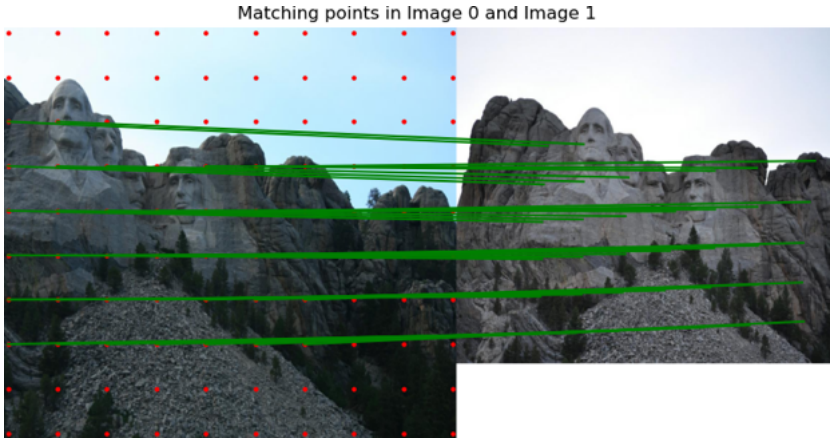


Figure 31: Depth Check Pair 9

Finally, we can generate an even denser set of correspondences (Here, I generated 250,000 correspondences) to form a pointcloud of the world 3D coordinates of the object in all 10 scenarios:

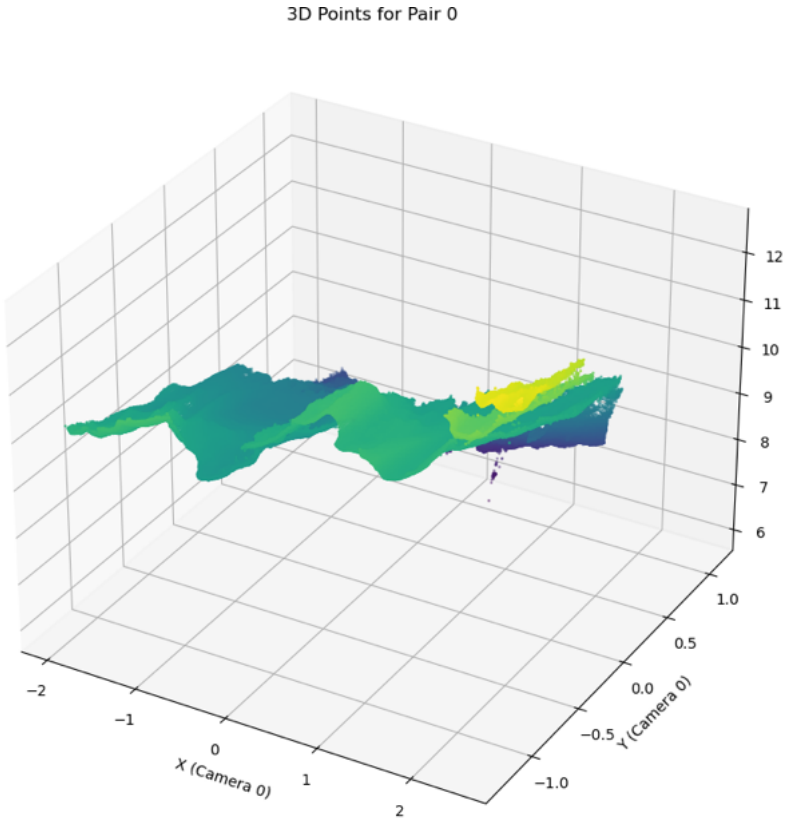


Figure 32: 3D Scatter Plot for Pair 0

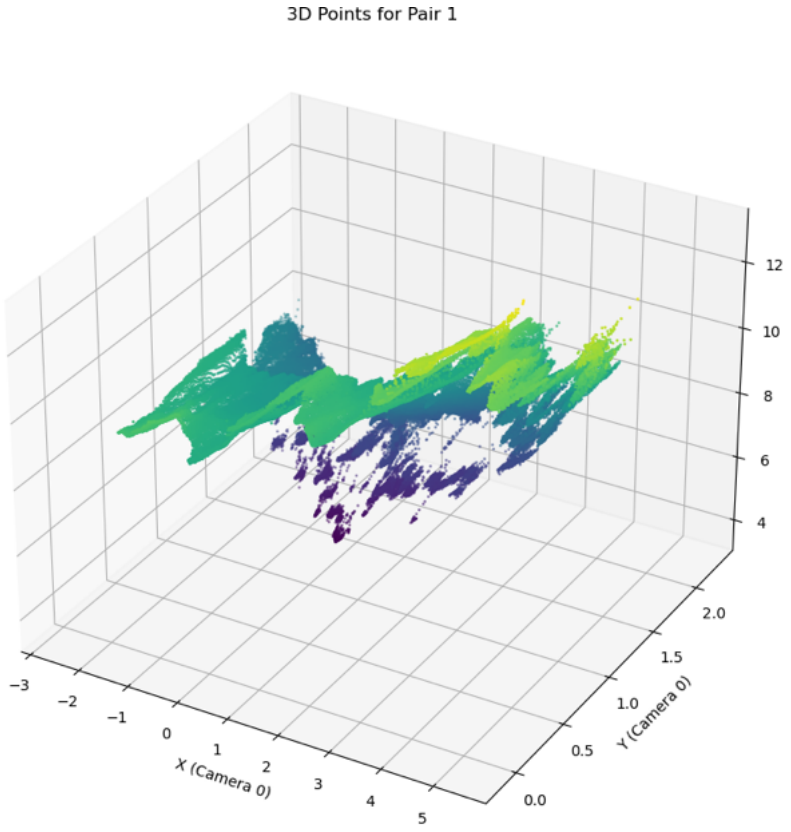


Figure 33: 3D Scatter Plot for Pair 1

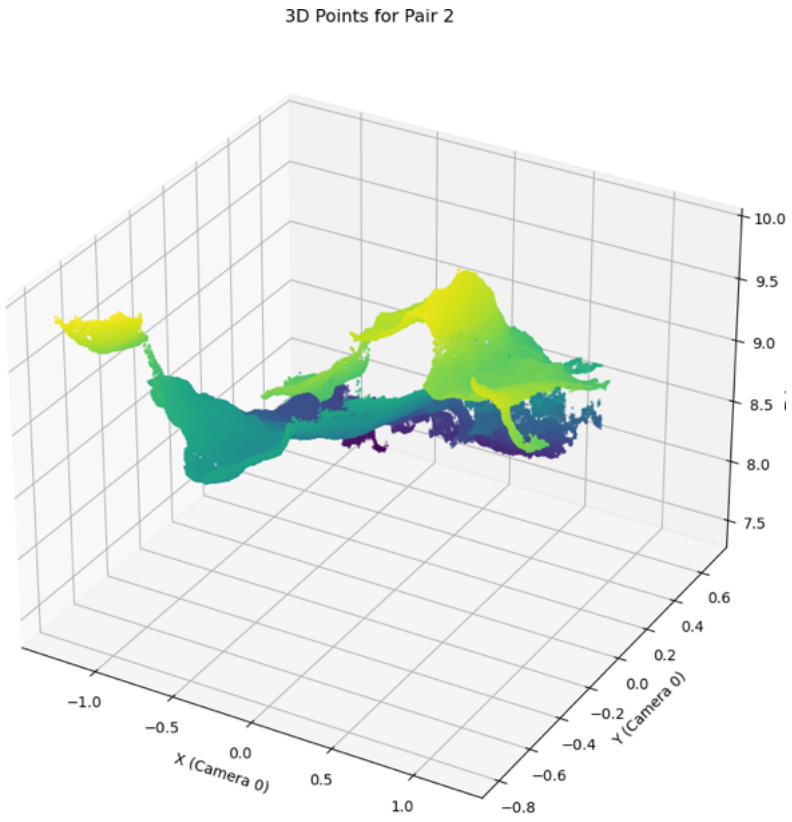


Figure 34: 3D Scatter Plot for Pair 2

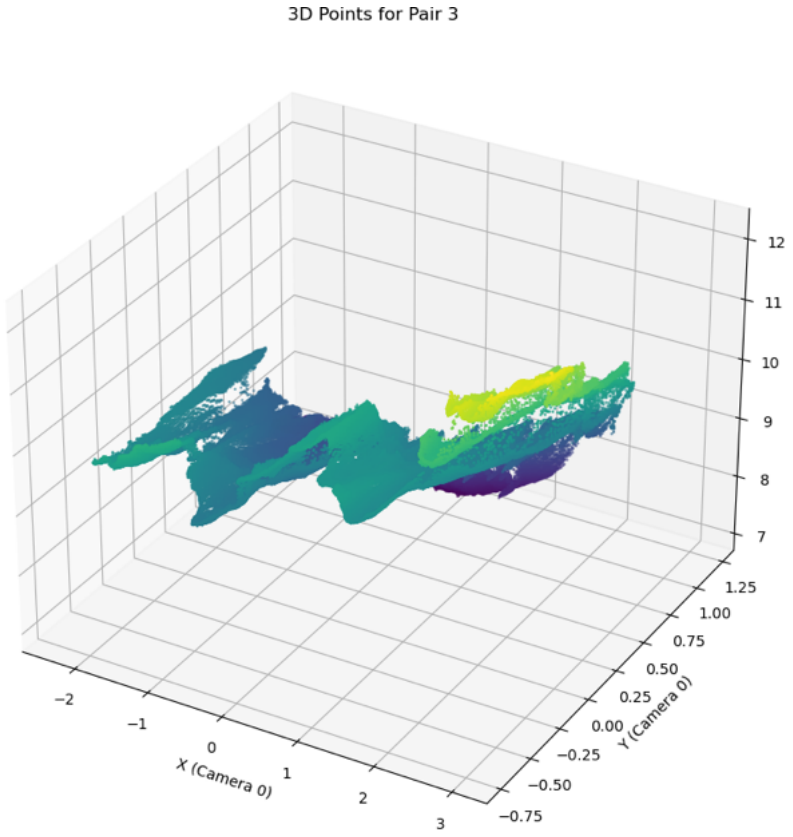


Figure 35: 3D Scatter Plot for Pair 3

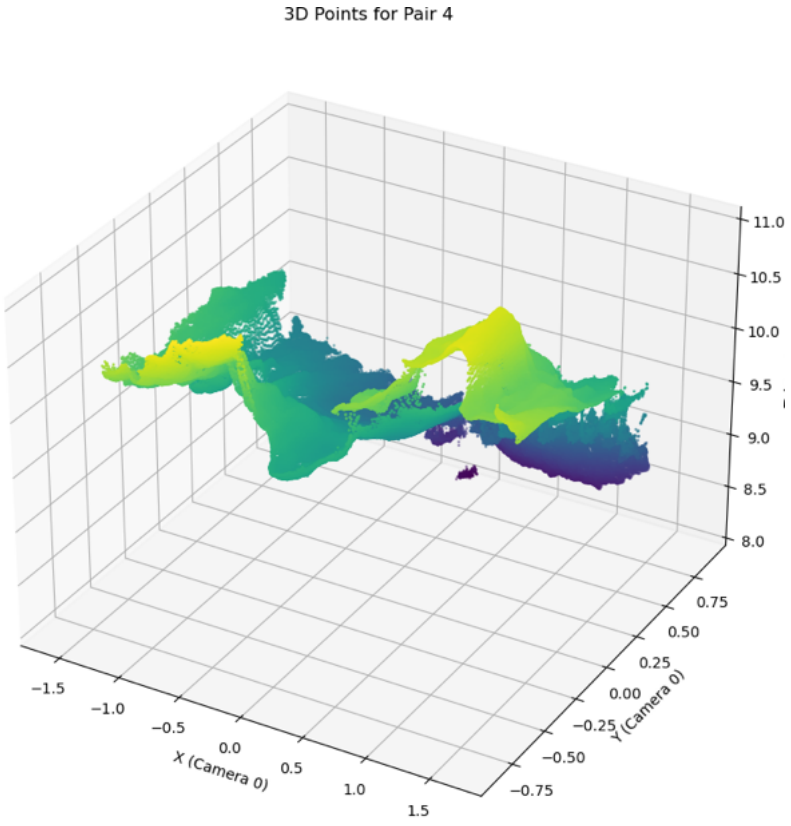


Figure 36: 3D Scatter Plot for Pair 4

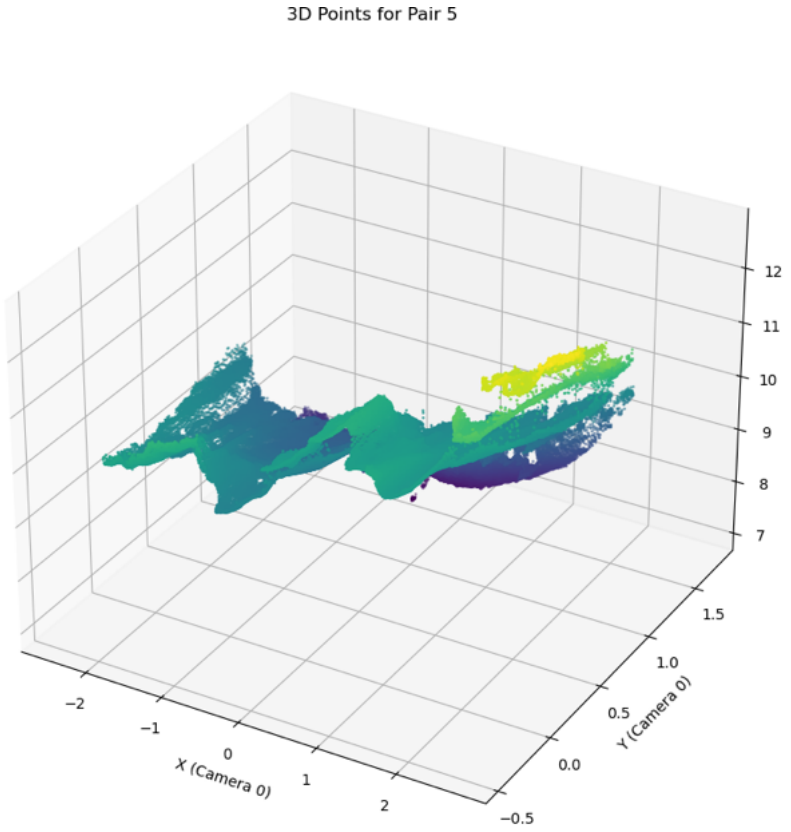


Figure 37: 3D Scatter Plot for Pair 5

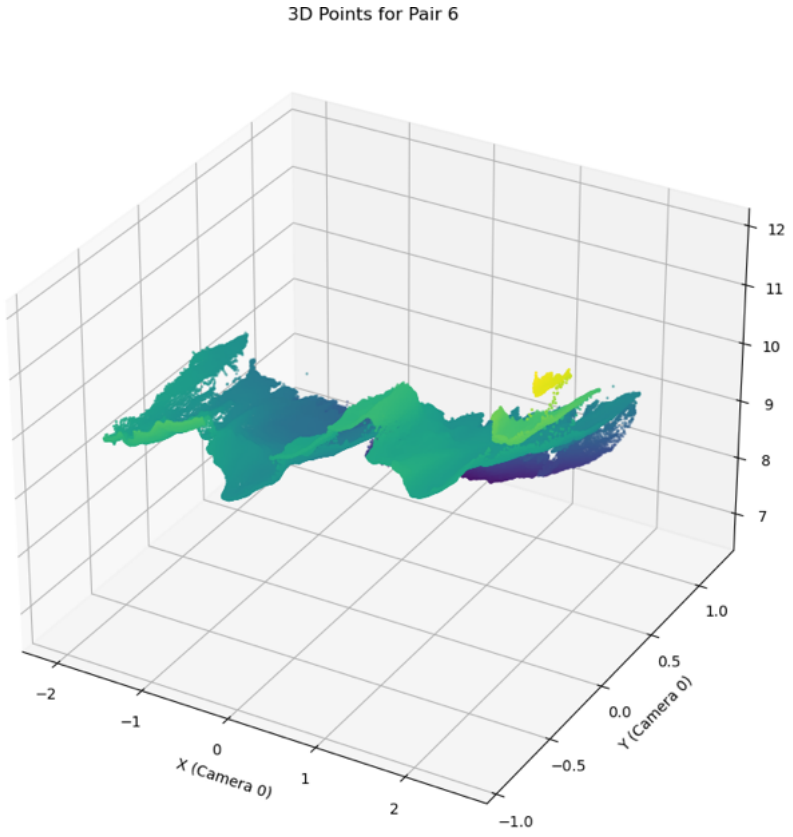


Figure 38: 3D Scatter Plot for Pair 6

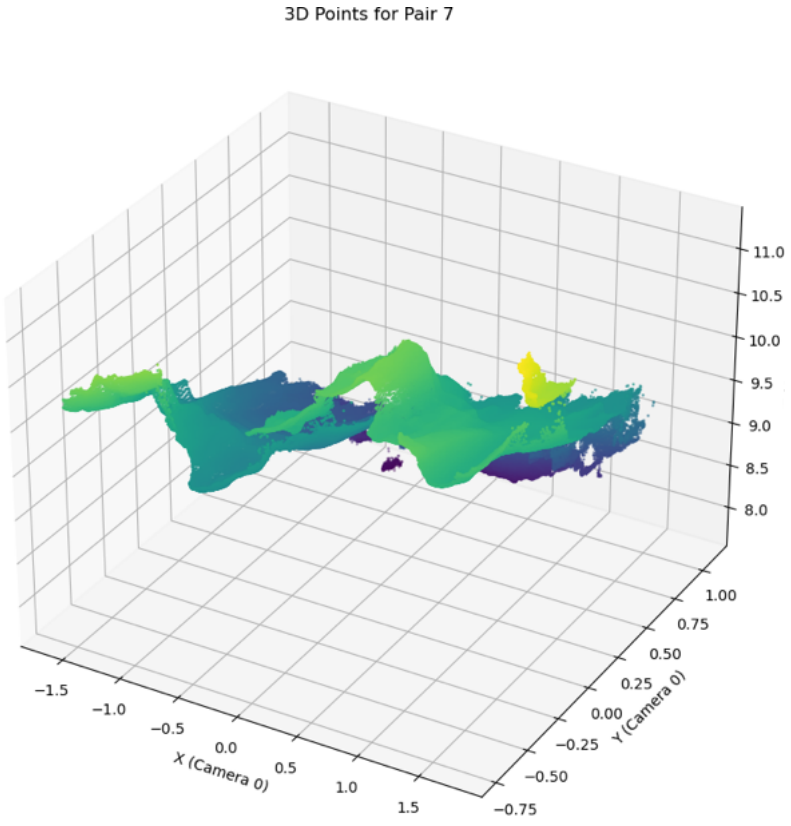


Figure 39: 3D Scatter Plot for Pair 7

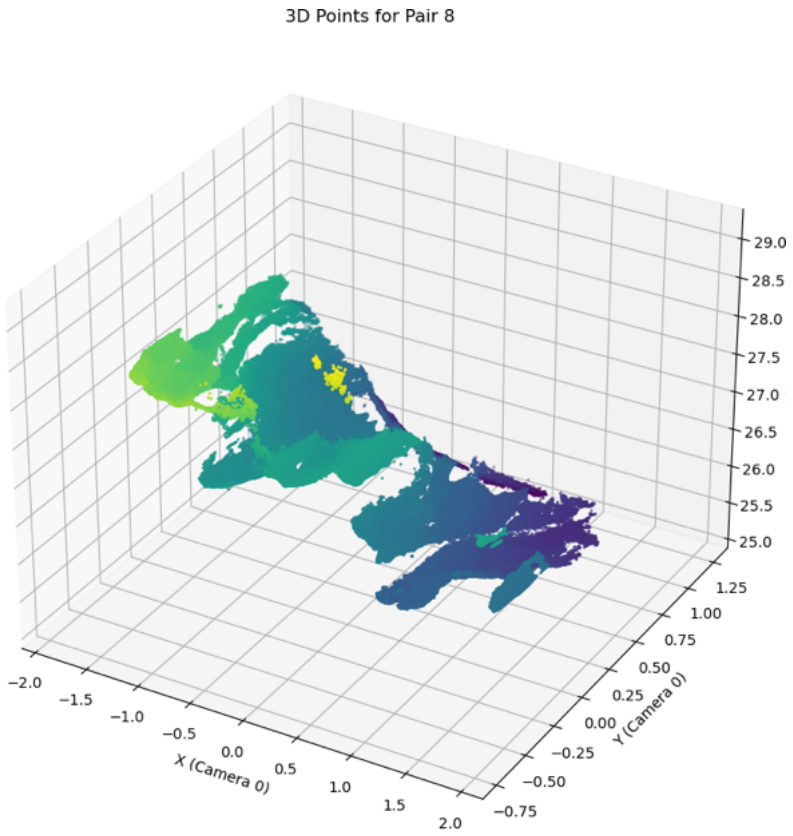


Figure 40: 3D Scatter Plot for Pair 8

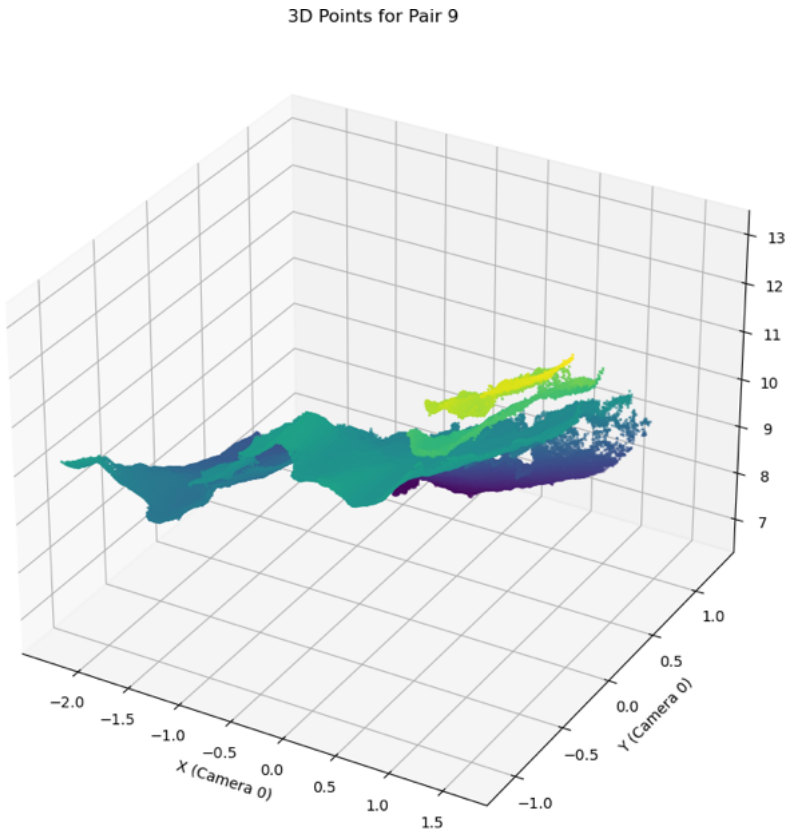


Figure 41: 3D Scatter Plot for Pair 9

6 Code

```
import cv2
import numpy as np
import random
import time
import matplotlib.pyplot as plt

from matplotlib.patches import ConnectionPatch
from scipy.optimize import least_squares
from scipy.linalg import null_space
from mpl_toolkits.mplot3d import Axes3D


image1 = cv2.imread("img1_alt.png")
image2 = cv2.imread("img2_alt.png")

image1_height, image1_width, _ = image1.shape
```

```
image2_height, image2_width, _ = image2.shape

# cv2.imshow("image1", image1)
# cv2.imshow("image2", image2)
# cv2.waitKey(0)
# cv2.destroyAllWindows()

visualize = True

# Since we are using canonical form this is constant
P = np.array([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0]])


def HC(point): # Helper function to turn pixels into HC
    return np.array([point[0], point[1], 1]).reshape((3, 1))

def dist(p1, p2):
    return np.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)

def condition_F(F):
    # Conditioning F by performing SVD and setting its smallest
    singular value in D to 0
    U, D, Vt = np.linalg.svd(F)
    D[-1] = 0

    D = np.diag(D)

    F_conditioned = np.dot(U, np.dot(D, Vt))

    return F_conditioned

def calculate_raster(image, H):
    height, width, _ = image.shape

    # Transforming corners through homography and choosing max
    bounds
    corners = np.array([
        [0, 0, 1],
        [width, 0, 1],
        [width, height, 1],
        [0, height, 1]
    ]).T

    transformed_corners = H @ corners
    transformed_corners /= (transformed_corners[-1, :] + 1e-6)

    min_x = min(transformed_corners[0, :])
    max_x = max(transformed_corners[0, :])
    min_y = min(transformed_corners[1, :])
    max_y = max(transformed_corners[1, :])

    w_out = int(max_x - min_x)
    h_out = int(max_y - min_y)

    return w_out, h_out


```

```
def visualize_correspondences():

    for (x1, y1), (x2, y2) in zip(image1_points, image2_points):
        cv2.circle(combined_image, (x1, y1), radius=5, color=(0, 0, 255), thickness=-1)
        cv2.circle(combined_image, (x2 + image1_width, y2), radius=5, color=(0, 0, 255), thickness=-1)

        cv2.line(combined_image, (x1, y1), (x2 + image1_width, y2), color=(0, 255, 0), thickness=2)

    plt.imshow(combined_image)
    plt.show()

def normalize_points(points):
    # Creation of normalization homography for 8-point method
    mean = np.mean(points, axis=0)
    std_dev = np.std(points, axis=0)
    scale = np.sqrt(2) / std_dev

    T = np.array([
        [scale[0], 0, -scale[0] * mean[0]],
        [0, scale[1], -scale[1] * mean[1]],
        [0, 0, 1]
    ])

    normalized_points = np.dot(T, np.column_stack((points, np.ones(len(points))))).T

    return normalized_points[:, :2], T

def estimate_F(image1_points, image2_points):
    # Normalize points
    points1_normalized, T1 = normalize_points(image1_points)
    points2_normalized, T2 = normalize_points(image2_points)

    # Construct A matrix
    A = np.array([
        [x2 * x1, x2 * y1, x2, y2 * x1, y2 * y1, y2, x1, y1, 1]
        for (x1, y1), (x2, y2) in zip(points1_normalized, points2_normalized)
    ])

    # SVD to solve for fundamental matrix parameters
    _, _, Vt = np.linalg.svd(A) # Using SVD to solve for homography terms
    F_normalized = Vt[-1].reshape((3, 3))

    # Condition F to be rank 2
    F_conditioned = condition_F(F_normalized)

    F = T2.T @ F_conditioned @ T1

    F /= F[2, 2]

    return F

```

```
123
124 def estimate_epipoles(F):
125     # Epipoles are left and right null vectors of F
126     e = null_space(F)
127     e_p = null_space(F.T)
128
129     e /= e[-1]
130     e_p /= e_p[-1]
131
132     return e, e_p
133
134
135 def calculate_camera_matrices(F, e_p):
136     e_p = e_p.flatten()
137
138     # Skew symmetric matrix for construction of P'
139     s = np.array([[0, -e_p[2], e_p[1]], [e_p[2], 0, -e_p[0]], [-e_p
        [1], e_p[0], 0]])
140
141     P_p = np.hstack((s @ F, e_p.reshape(-1, 1)))
142
143     return P_p
144
145 def compute_rectification_homography_right(right_epipole):
146     # Translational homographies to bring epipole to image origin
        and back
147     T1 = np.array([[1, 0, -image2_width / 2], [0, 1, -image2_height
         / 2], [0, 0, 1]])
148     T2 = np.array([[1, 0, image2_width / 2], [0, 1, image2_height /
         2], [0, 0, 1]])
149
150     right_epipole = (T1 @ right_epipole).ravel()
151
152     right_epipole = right_epipole / right_epipole[2]
153
154     # Calculate angle to bring epipole to infinity
155     theta = -np.arctan2(right_epipole[1], right_epipole[0])
156
157     R = np.array([[np.cos(theta), -np.sin(theta), 0], [np.sin(theta
        ), np.cos(theta), 0], [0, 0, 1]])
158
159     transformed_epipole = (R @ right_epipole.reshape((3, 1))).ravel
        ()
160
161     transformed_epipole = transformed_epipole / transformed_epipole
        [2]
162
163     f = transformed_epipole[0]
164
165     G = np.array([[1, 0, 0], [0, 1, 0], [-1 / f, 0, 1]])
166
167     H_p = T2 @ G @ R @ T1
168
169     H_p /= H_p[2, 2]
170
171     return H_p
172
```

```
def compute_rectification_homography_left(H_p, left_epipole, image1_points, image2_points):
    # Initial estimate for left homography
    H_hat = compute_rectification_homography_right(left_epipole)

    A = []
    b = []

    # Use minimization technique to find left homography
    for point_left, point_right in zip(image1_points, image2_points):
        point_left_hc = HC(point_left)
        point_right_hc = HC(point_right)

        transformed_point_left = (H_hat @ point_left_hc).flatten()
        transformed_point_right = (H_p @ point_right_hc).flatten()

        A.append([transformed_point_left[0], transformed_point_left[1], 1])
        b.append(transformed_point_right[0])


    A = np.array(A)
    b = np.array(b)

    min_solution = np.linalg.pinv(A) @ b

    Ha = np.array([[min_solution[0], min_solution[1], min_solution[2]], [0, 1, 0], [0, 0, 1]])

    H = Ha @ H_hat

    H /= H[2, 2]

    return H

# def verify_F(F):
#   for point1, point2 in zip(image1_points, image2_points):
#       point1_hc = HC(point1)
#       point2_hc = HC(point2)

#       print("x'^T * F * x: ", point2_hc.T @ F @ point1_hc)

def triangulate_point(P_p, point1, point2):
    # Standard triangulation using P' and P
    A = [
        point1[0] * P[2] - P[0],
        point1[1] * P[2] - P[1],
        point2[0] * P_p[2] - P_p[0],
        point2[1] * P_p[2] - P_p[1]
    ]
    A = np.array(A)
    _, _, Vt = np.linalg.svd(A.T @ A)
    X = Vt[-1]

    return X / X[3]

```

```
226 def backprojection(P_p, image1_points, image2_points):
227     errors = []
228     world_points = []
229 
230     for point1, point2 in zip(image1_points, image2_points):
231         X = triangulate_point(P_p, HC(point1), HC(point2))  #
    Triangulate the 3D point
232         world_points.append(X.flatten()[:3])
233 
234         # Apply corresponding camera matrix to world point
235         reprojected_point1 = P @ X
236         reprojected_point2 = P_p @ X
237 
238         # Normalize the projected points
239         reprojected_point1 /= reprojected_point1[2]
240         reprojected_point2 /= reprojected_point2[2]
241 
242         # Calculate the Euclidean distance (geometric error)
243         error1 = np.linalg.norm(reprojected_point1[:2] - point1)**2
244         error2 = np.linalg.norm(reprojected_point2[:2] - point2)**2
245 
246         errors.extend([error1, error2])
247 
248     return np.array(errors), np.array(world_points)
249 
250 def geometric_error(P_p_vector, image1_points, image2_points):
251 
252     P_p = P_p_vector.reshape(3, 4)  # Reshape the flattened P'
    vector
253 
254     errors, _ = backprojection(P_p, image1_points, image2_points)
255 
256     return errors
257 
258 def refine_P_p(P_p, image1_points, image2_points):
259 
260     P_p_vector = P_p.flatten()  # Flatten P' for optimization
261 
262     # Perform LM on P_p to reduce euclidean distance error
263     result = least_squares(
264         geometric_error,
265         P_p_vector,
266         method='lm',
267         args=(image1_points, image2_points)
268     )
269 
270     refined_P_p = result.x.reshape(3, 4)  # Reshape optimized
    vector back to matrix
271 
272     return refined_P_p
273 
274 def rectify_images(image1, image2, image1_points, image2_points,
    collect_world_points):
275 
276     # Estimate F from point correspondences
277     F = estimate_F(image1_points, image2_points)
278 
```

```
    # Calculate initial estimates for epipoles
    initial_e, initial_e_p = estimate_epipoles(F)
    initial_e = initial_e.reshape((3, 1))
    initial_e_p = initial_e_p.reshape((3, 1))

    # Use F and epipole estimates to construct P'
    initial_P_p = calculate_camera_matrices(F, initial_e_p)

    # Use LM to refine P' estimate
    refined_P_p = refine_P_p(initial_P_p, image1_points,
    image2_points)

    # Extract new epipole from P', construct skew-symmetric matrix
    for refined F calculation
    refined_e_p = refined_P_p[:, 3]
    s = np.array([[0, -refined_e_p[2], refined_e_p[1]], [
    refined_e_p[2], 0, -refined_e_p[0]], [-refined_e_p[1],
    refined_e_p[0], 0]])

    # Calculate refined F and condition to enforce rank 2
    refined_F = s @ refined_P_p @ np.linalg.pinv(P)

    refined_F /= refined_F[2, 2]
    refined_F = condition_F(refined_F)

    # Extract refined epipoles from refined F
    refined_e, refined_e_p = estimate_epipoles(refined_F)

    # Construct rectification homographies from
    H_p = compute_rectification_homography_right(refined_e_p)
    H = compute_rectification_homography_left(H_p, refined_e,
    image1_points, image2_points)

    if collect_world_points:
        _, world_points = backprojection(refined_P_p, image1_points
    , image2_points)
        return H, H_p, refined_P_p, world_points

    return H, H_p, refined_P_p

def create_visualization(image1_rectified, image2_rectified, H, H_p
    ):
    combined_rectified_image = np.hstack((image1_rectified,
    image2_rectified))

    for (x1, y1), (x2, y2) in zip(image1_points, image2_points):
        point_left_hc = HC((x1, y1))
        point_right_hc = HC((x2, y2))

        transformed_point_left = (H @ point_left_hc).flatten()
        transformed_point_right = (H_p @ point_right_hc).flatten()

        pixel_left = (int(transformed_point_left[0] /
    transformed_point_left[2]), int(transformed_point_left[1] /
    transformed_point_left[2]))
        pixel_right = (int(transformed_point_right[0] /
    transformed_point_right[2]), int(transformed_point_right[1] /
```

```
    transformed_point_right[2]))
325
326
327         cv2.circle(image1_rectified, pixel_left, radius=5, color
    =(255, 0, 0), thickness=-1)
328         cv2.circle(image2_rectified, pixel_right, radius=5, color
    =(0, 255, 0), thickness=-1)
329
330         cv2.circle(combined_rectified_image, pixel_left, radius=5,
    color=(255, 0, 0), thickness=-1)
331         cv2.circle(combined_rectified_image, (pixel_right[0] +
    new_width, pixel_right[1]), radius=5, color=(0, 255, 0),
    thickness=-1)
332
333         cv2.line(combined_rectified_image, pixel_left, (pixel_right
    [0] + new_width, pixel_right[1]), color=(0, 0, 255), thickness
    =2)
334
335     return combined_rectified_image
336
337 def extract_interest_points(rectified_image, canny_threshold1,
    canny_threshold2):
338     # Extract edges using canny detector, collect points where mask
     is true
339     edges = cv2.Canny(rectified_image, canny_threshold1,
    canny_threshold2)
340     points = np.column_stack(np.where(edges > 0))
341
342     # Sort by row
343     points = points[np.lexsort((points[:, 1], points[:, 0]))]
344
345     if visualize:
346         cv2.imshow("Interest Points", edges)
347         cv2.waitKey(0)
348         cv2.destroyAllWindows()
349
350     return points
351
352 def get_candidate_points(point, interest_points2, row_window=3):
353     row_range = range(point[0] - row_window, point[0] + row_window
    + 1)
354     return interest_points2[np.isin(interest_points2[:, 0],
    row_range)]
355
356 def extract_patch(image, point, patch_size=5):
357     half_size = patch_size // 2
358     x, y = point
359     h, w = image.shape[:2]
360
361     # Out of bounds checking
362     if(x - half_size < 0 or x + half_size >= w or y - half_size < 0
     or y > half_size >= h):
363         return None
364
365     # Collect all points within patch window
366     return image[x - half_size:x + half_size + 1, y - half_size:y +
     half_size + 1]
```

```
367
368 def SSD(patch1, patch2):
369     # Sum of squared differences if patches are both all full
370     if patch1 is None or patch2 is None:
371         return float("inf")
372     return np.sum((patch1 - patch2)**2)
373
374 def compute_dense_correspondences(image1_rectified,
    image2_rectified, interest_points1, interest_points2):
375     point1_correspondences = []
376     point2_correspondences = []
377
378     for point1 in interest_points1:
379         # Find potential candidates through buffered row
380         candidates = get_candidate_points(point1, interest_points2)
381         if len(candidates) == 0:
382             continue
383
384         # Go through each candidate and find lowest SSD between
    parent patch and potential patch
385         patch1 = extract_patch(image1_rectified, point1)
386         best_candidate, best_score = None, float("inf")
387
388         for candidate in candidates:
389             patch2 = extract_patch(image2_rectified, candidate)
390             score = SSD(patch1, patch2)
391
392             if score < best_score:
393                 best_score = score
394                 best_candidate = candidate
395
396         # Add best candidate to correspondence list
397         if best_candidate is not None:
398             point1_correspondences.append([point1[1], point1[0]])
399             point2_correspondences.append([best_candidate[1],
    best_candidate[0]])
400
401     return np.array(point1_correspondences), np.array(
    point2_correspondences)
402
403 if __name__ == "__main__":
404     image1_points = np.array([[602, 68], [287, 64], [496, 304],
    [303, 506], [611, 345], [195, 446], [284, 475], [548, 233]])
405     image2_points = np.array([[584, 83], [268, 80], [505, 311],
    [277, 490], [598, 368], [185, 425], [263, 459], [533, 250]])
406
407     # Calculate rectification homographies
408     H, H_p, _ = rectify_images(image1, image2, image1_points,
    image2_points, False)
409
410     # Calculate updated buffers from homographies
411     width_new1, height_new1 = calculate_raster(image1, H)
412     width_new2, height_new2 = calculate_raster(image2, H_p)
413
414
415     if visualize:
416         new_width = max(width_new1, width_new2)
```

```
        new_height = max(height_new1, height_new2)

        image1_rectified = cv2.warpPerspective(image1, H, (new_width, new_height))
        image2_rectified = cv2.warpPerspective(image2, H_p, (new_width, new_height))
    else:
        image1_rectified = cv2.warpPerspective(image1, H, (width_new1, height_new1))
        image2_rectified = cv2.warpPerspective(image2, H_p, (width_new2, height_new2))

    # if visualize:
    #   combined_image = create_visualization(image1_rectified, image2_rectified, H, H_p)

    #   cv2.imshow("rectified left image", image1_rectified)
    #   cv2.imshow("rectified right image", image2_rectified)
    #   cv2.imshow("combined_image", combined_image)
    #   cv2.waitKey(0)
    #   cv2.destroyAllWindows()


    # Canny edge detection parameters
    canny_threshold1 = 600
    canny_threshold2 = 800

    # Extract interest points
    interest_points1 = extract_interest_points(image1_rectified, canny_threshold1, canny_threshold2)
    interest_points2 = extract_interest_points(image2_rectified, canny_threshold1, canny_threshold2)

    # Find correspondences using SSD patch scoring
    new_image1_points, new_image2_points = compute_dense_correspondences(image1_rectified, image2_rectified, interest_points1, interest_points2)

    # Refine P' from dense correspondences
    _, _, refined_P_p, world_points = rectify_images(image1_rectified, image2_rectified, new_image1_points, new_image2_points, True)

    if visualize:
        combined_rectified_image = np.hstack((image1_rectified, image2_rectified))

        for point1, point2 in zip(new_image1_points, new_image2_points):
            color = (random.randint(0, 255), random.randint(0, 255), random.randint(0, 255))

            offset = image1_rectified.shape[1]
            point2 = (point2[0] + offset, point2[1])

            cv2.circle(combined_rectified_image, point1, radius=2, color=(0, 255, 0), thickness=-1)
```

```
            cv2.circle(combined_rectified_image, point2, radius=2, color=(0, 255, 0), thickness=-1)

            cv2.line(combined_rectified_image, point1, point2, thickness=2, color=color)

        cv2.imshow("combined_image", combined_rectified_image)
        cv2.waitKey(0)
        cv2.destroyAllWindows()

    # Populate 2D and 3D points to form pointcloud
    visualization_points_3d = []
    visualization_points_2d_left = []
    visualization_points_2d_right = []

    for (x1, y1), (x2, y2) in zip(image1_points, image2_points):
        point_left_hc = HC((x1, y1))
        point_right_hc = HC((x2, y2))

        transformed_point_left = (H @ point_left_hc).flatten()
        transformed_point_right = (H_p @ point_right_hc).flatten()

        visualization_points_2d_left.append([transformed_point_left[0] / transformed_point_left[2], transformed_point_left[1] / transformed_point_left[2]])
        visualization_points_2d_right.append([transformed_point_right[0] / transformed_point_right[2], transformed_point_right[1] / transformed_point_right[2]])

        world_visualization_point = triangulate_point(refined_P_p, transformed_point_left / transformed_point_left[2], transformed_point_right / transformed_point_right[2])
        visualization_points_3d.append(world_visualization_point)


    visualization_points_3d = np.array(visualization_points_3d)

    print("world_points shape: ", world_points.shape)
    print("visualization_points shape: ", visualization_points_3d.shape)

    # Plotting
    X = world_points[:, 0]
    Y = world_points[:, 1]
    Z = world_points[:, 2]

    X_v = visualization_points_3d[:, 0]
    Y_v = visualization_points_3d[:, 1]
    Z_v = visualization_points_3d[:, 2]

    fig = plt.figure(figsize=(6, 9))

    ax1 = fig.add_subplot(312, projection='3d')
    ax2 = fig.add_subplot(311)
    ax3 = fig.add_subplot(313)

    ax1.scatter(X, Y, Z, c='b', s=10)
```

```
    ax1.scatter(X_v, Y_v, Z_v, c='r', s=100)
    ax1.set_xlabel("X")
    ax1.set_ylabel("Y")
    ax1.set_zlabel("Z")
    ax1.set_title("3D Projection")


    ax2.imshow(image1_rectified)
    ax3.imshow(image2_rectified)

    for i in range(len(visualization_points_3d)):
        left_point = visualization_points_2d_left[i]
        right_point = visualization_points_2d_right[i]

        ax2.scatter(left_point[0], left_point[1], color='r', s=50)
        ax3.scatter(right_point[0], right_point[1], color='r', s
    =50)


    ax2.set_xlabel('x')
    ax2.set_ylabel('y')
    ax2.set_title("Left Rectified Image")

    ax3.set_xlabel('x')
    ax3.set_ylabel('y')
    ax3.set_title("Right Rectified Image")


    plt.tight_layout()
    plt.show()
```

Listing 1: Stereo Rectification

```
import cv2
import numpy as np
import matplotlib.pyplot as plt


def compute_census_transform(image, window_size):
    rows, cols = image.shape
    half_window = window_size // 2
    census = np.zeros((rows, cols, window_size**2), dtype=np.uint8)

    for i in range(half_window, rows - half_window):
        for j in range(half_window, cols - half_window):
            center_pixel = image[i, j]
            bit_vector = []

            for m in range(-half_window, half_window + 1):
                for n in range(-half_window, half_window + 1):
                    neighbor_pixel = image[i + m, j + n]
                    if neighbor_pixel > center_pixel:
                        bit_vector.append(1)
                    else:
                        bit_vector.append(0)

            census[i, j] = bit_vector
```

```
25
26      return census
27
28  def census_disparity_map(image1, image2, window_size, dmax):
29      rows, cols = image1.shape
30
31      disparity_map = np.zeros((rows, cols), dtype=np.uint8)
32      half_window = window_size // 2
33
34      census1 = compute_census_transform(image1, window_size)
35      census2 = compute_census_transform(image2, window_size)
36
37      for i in range(half_window, rows - half_window):
38          for j in range(half_window, cols - half_window):
39              min_cost = float("inf")
40              best_disparity = 0
41
42              for d in range(dmax + 1):
43                  if j - d < half_window:
44                      # Skipping disparities that go out of bounds
45                      continue
46
47                  bitvector1 = census1[i, j]
48                  bitvector2 = census2[i, j - d]
49
50                  xor = np.bitwise_xor(bitvector1, bitvector2)
51                  data_cost = np.sum(xor)
52
53                  if data_cost < min_cost:
54                      min_cost = data_cost
55                      best_disparity = d
56
57              disparity_map[i, j] = best_disparity
58
59      return disparity_map
60
61
62  if __name__ == "__main__":
63      image1 = cv2.imread("Task3Images/im2.png", cv2.IMREAD_GRAYSCALE
        )
64      image2 = cv2.imread("Task3Images/im6.png", cv2.IMREAD_GRAYSCALE
        )
65
66      ground_truth = cv2.imread("Task3Images/disp2.png", cv2.
        IMREAD_GRAYSCALE)
67      ground_truth = np.array(np.array(ground_truth, dtype=np.float32
        ) / 4, dtype=np.uint8)
68
69      window_size = 11
70      dmax = 64
71      disparity_threshold = 2
72      num_valid_points = 0
73      num_accurate_points = 0
74
75      disparity_map = census_disparity_map(image1, image2,
        window_size, dmax)
76      error_mask = np.zeros_like(disparity_map)
```

```
77
78      for i in range(disparity_map.shape[0]):
79          for j in range(disparity_map.shape[1]):
80              ground_truth_value = ground_truth[i, j]
81
82              if ground_truth_value > 0:
83                  num_valid_points += 1
84
85              if(ground_truth_value == 0):
86                  continue
87
88              if(abs(ground_truth_value - disparity_map[i, j]) <=
    disparity_threshold):
89                  error_mask[i, j] = 255
90                  num_accurate_points += 1
91
92
93      print("accuracy: ", num_accurate_points / num_valid_points)
94
95      cv2.imshow("Disparity Map", disparity_map)
96      cv2.imshow("Ground Truth", ground_truth)
97      cv2.imshow("Error Mask", error_mask)
98      cv2.waitKey(0)
99      cv2.destroyAllWindows()
100
101 # 5x5 40% accuracy
102 # 11x11
```

Listing 2: Disparity Census Transform

```
1  import numpy as np
2  import pickle as pkl
3  import matplotlib.pyplot as plt
4  import h5py # for reading depth maps
5  import time
6
7  """
8  A few notes on the scene_info dictionary:
9  - depth maps are stored as h5 files. Depth is the distance of the
       object from the camera (ie Z coordinate in camera coordinates).
        The depth map can contain invalid points (depth = 0) which
       correspond to points where the depth could not be estimated.
10 - The intrinsics are stored as a 3x3 matrix.
11 - The poses [R,t] are stored as a 4x4 matrix to allow for easy
       transformation of points from one camera to the other. The
       resulting transformation matrix is a 4x4 matrix is of the form:
12     T = [[R, t]
13         [0, 1]] where R is a 3x3 rotation matrix and t is a 3x1
       translation vector.
14 """
15
16 DEPTH_THR = 0.1
17 dense = False
18 if dense:
19     num_points_in_mesh = 500
20 else:
21     num_points_in_mesh = 10
```

```python


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: {img0}, {depth0}"
        assert img1.shape[:-1] ==  depth1.shape, f"depth and image shapes do not match: {img1}, {depth1}"


        # 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
        x = np.linspace(10, img0.shape[1] - 10, num_points_in_mesh)  # ignore a border of 10 pxls
        y = np.linspace(10, img0.shape[0] - 10, num_points_in_mesh)

        # meshgrid of x and y coordinates
        xx, yy = np.meshgrid(x, y)
        xx, yy = xx.flatten(), yy.flatten()

        # Make homogeneous coordinates for points0 [3, N]
        points0 = np.vstack((xx, yy, np.ones_like(xx)))

        # Get depth values at points0
        depth_values0 = depth0[yy.astype(int), xx.astype(int)]

        # Remove points with invalid depth (depth == 0)
        valid_points = depth_values0 > 0
        points0 = points0[:, valid_points]
        depth_values0 = depth_values0[valid_points]


        # (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 *= depth_values0
        # make homogeneous coordinates [4,N]
        xyz_cam0_hc = np.vstack((xyz_cam0, np.ones((1, xyz_cam0.shape[1]))))
        # convert to world frame [4,N]
        xyz_world_hc = np.linalg.inv(T0) @ xyz_cam0_hc

        # (4) Transform these points to camera 1 frame
        K1 = scene_info[i_pair]['K1']
        T1 = scene_info[i_pair]['T1']

        # transform points to camera 1 frame
        xyz_cam1_hc = T1 @ xyz_world_hc
        # convert to camera 1 coordinates
        xyz_cam1 = xyz_cam1_hc[:3, :]
        # get z coordinates for depth check
        estimated_depth_values1 = xyz_cam1[2, :]


        # project to image 1
        points1 = K1 @ xyz_cam1 # [3, N]
        # normalize by dividing by last row
        points1 = points1 / points1[2, :] # [3, N]
        # check if points1 are within image bounds
        # Create mask
        valid_projections = (
            (points1[0, :] >= 0) & (points1[0, :] < img1.shape[1]) & (points1[1, :] >= 0) & (points1[1, :] < img1.shape[0])
        )
        # Apply mask
        points1 = points1[:, valid_projections]
        estimated_depth_values1 = estimated_depth_values1[valid_projections]
        # get the depth values at these points using the depth map
        true_depth_values1 = depth1[points1[1, :].astype(int), points1[0, :].astype(int)]


        if(not dense):
            # (5) plot matching points in image 0 and image 1 with depth check such that the depth values match
            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], 3), dtype=np.uint8) * 255
            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_THR and true_depth_values1[i] != 0:
                    ax.plot([points0[0,i], points1[0, i] + img0.shape[1]], [points0[1,i], points1[1, i]], 'g')

            plt.savefig(f'./pics/depth_check_pair_{i_pair}.png', bbox_inches='tight', pad_inches=0)
            plt.close()
            print(f"Done with pair {i_pair}")

        if dense:
            # (6) Plot all 3D points for the pair

            fig = plt.figure(figsize=(10, 10))
            ax = fig.add_subplot(111, projection='3d')

            X, Y, Z = xyz_cam0[0, :], xyz_cam0[1, :], xyz_cam0[2, :]

            ax.scatter(X, Y, Z, c=Z, cmap='viridis', s=1)
            ax.set_xlabel('X (Camera 0)')
            ax.set_ylabel('Y (Camera 0)')
            ax.set_zlabel('Z (Depth)')
            plt.title(f'3D Points for Pair {i_pair}')
            plt.savefig(f'./pics/3d_points_pair_{i_pair}.png', bbox_inches='tight', pad_inches=0)
            plt.close()
            print(f"Done with pair {i_pair}")
```

Listing 3: Depth Check