ECE 661 - Computer Vision
Homework - 9

Arnav Singh

Contents

1 Theory Question

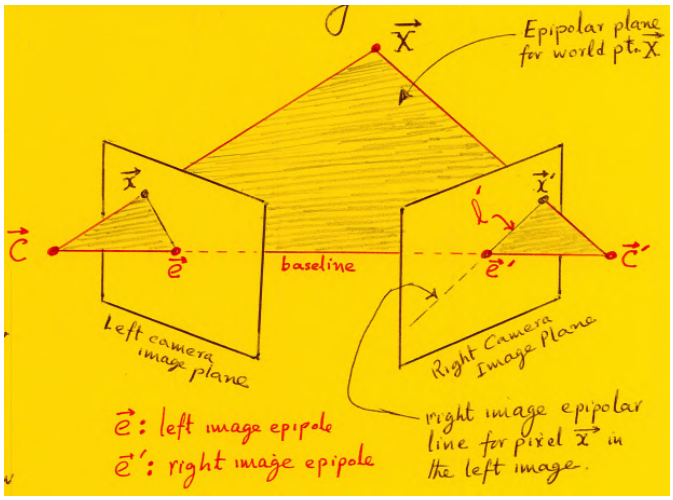


Figure 1: Epipolar geometry

The equation of the line that joins the world point $\boldsymbol{X}$ and center of projection of the left camera $\boldsymbol{C}$ is

$$\boldsymbol{L} \equiv \boldsymbol{C} + \lambda \boldsymbol{X}$$

where, λ is any scalar quantity.
The image of $\boldsymbol{L}$ in the right image is

$$\boldsymbol{l}' = \boldsymbol{P}' \ \boldsymbol{L} = \boldsymbol{P}' \ \boldsymbol{C} + \lambda \boldsymbol{P}' \ \boldsymbol{X} = \boldsymbol{e}' + \lambda \boldsymbol{x}'$$

The line $\boldsymbol{l}' = \boldsymbol{e}' + \lambda \boldsymbol{x}'$ represents a line that passes through the points $\boldsymbol{e}'$ and $\boldsymbol{x}'$, which is the exact definition of the right epipolar line.
This proves that the correspondence ($\boldsymbol{x}'$ in the right image) of $\boldsymbol{x}$ in the left image lies on the right epipolar line.

2 Task - 1: Projective Stereo Reconstruction

2.1 Methodology

2.1.1 Image Rectification

Image rectification is the process of applying appropriate homographies to the left and right image so their epipolar lines lie parallel to the x-axis. This implies that the epipoles (images of the camera center of projections in the other camera) corresponding to both cameras lie at the ideal point along x-axis i.e., $\begin{bmatrix}1 & 0 & 0\end{bmatrix}^\top$.

- To find the epipoles and epipolar lines we must find and initial estimate of the rank 2, 3×3 fundamental matrix $\boldsymbol{F}$ that contains all the information about the pair of cameras. In order to calculate $\boldsymbol{F}$ we manually select a set (minimum 8) of correspondences ($\boldsymbol{x}_i \leftrightarrow \boldsymbol{x}'_i$) in images I, I' and are constrained to $\boldsymbol{F}$ by

$$\boldsymbol{x}'^\top \boldsymbol{F} \boldsymbol{x} = 0.$$

  This can be solved using linear least squares (preceded by normalization of the coordinates $\boldsymbol{x}, \boldsymbol{x}'$). The condition of $|\boldsymbol{F}| = 0$ can be forced by setting its smallest singular value to 0.

  The normalization of the points $\boldsymbol{x}_i = \begin{bmatrix}x_i & y_i & 1\end{bmatrix}^\top$ can be done using a 3×3 transformation matrix $\boldsymbol{T}$

$$\tilde{\boldsymbol{x}}_i = \boldsymbol{T}\boldsymbol{x}_i \qquad \boldsymbol{T} = \begin{bmatrix} c & 0 & -c\bar{x} \\ 0 & c & -c\bar{y} \\ 0 & 0 & 1 \end{bmatrix}$$

  where $(\bar{x}, \bar{y})$ is the centroid of the set of correspondences, c is defined as $\frac{2}{\sqrt{\bar{D}}}$ and $\bar{D}$ is the mean Euclidean distance of the centroid from the correspondence points.

  The set of homogeneous equations to solve for $\tilde{\boldsymbol{F}}$ are

$$\boldsymbol{A}\tilde{\boldsymbol{f}} = \boldsymbol{0}$$

  where

$$\boldsymbol{A} = \begin{bmatrix} \tilde{x}'_1\tilde{x}_1 & \tilde{x}'_1\tilde{y}_1 & \tilde{x}'_1 & \tilde{y}'_1\tilde{x}_1 & \tilde{y}'_1\tilde{y}_1 & \tilde{y}'_1 & \tilde{x}_1 & \tilde{y}_1 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \tilde{x}'_N\tilde{x}_N & \tilde{x}'_N\tilde{y}_N & \tilde{x}'_N & \tilde{y}'_N\tilde{x}_N & \tilde{y}'_N\tilde{y}_N & \tilde{y}'_N & \tilde{x}_N & \tilde{y}_N & 1 \end{bmatrix} \qquad \tilde{\boldsymbol{f}} = \begin{bmatrix} f_{11} \\ f_{12} \\ f_{13} \\ f_{21} \\ f_{22} \\ f_{23} \\ f_{31} \\ f_{32} \\ f_{33} \end{bmatrix}$$

  The final $\boldsymbol{F}$ is obtained by

$$\boldsymbol{F} = \boldsymbol{T}'^\top \, \tilde{\boldsymbol{F}} \, \boldsymbol{T} \tag{1}$$

  followed by setting its smallest singular value to 0.

- The left and right epipoles $\boldsymbol{e}, \boldsymbol{e}'$ are the null vectors of $\boldsymbol{F}$ and $\boldsymbol{F}^\top$, respectively.
- We assume that the cameras are in their canonical configuration, i.e., the world co-ordinate frame coincides with co-ordinate frame of the left camera, and, the right camera is at a rotation, translation w.r.t the left camera. This simplifies the form of the projection matrices and left camera's center of projection

$$\boldsymbol{P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \qquad \boldsymbol{C} = \begin{bmatrix}0 & 0 & 0 & 1\end{bmatrix}^\top \tag{2}$$

$$\boldsymbol{P}' = [[\boldsymbol{e}']_\times \boldsymbol{F} | \boldsymbol{e}'] \tag{3}$$

  where $[\boldsymbol{a}]_\times$ is the 3×3 cross-product representation of any vector $\boldsymbol{a}$.

- This initial estimate is refined using non-linear least squares (for example Levenberg-Marquardt). The parameters over which the cost minimization are the twelve matrix elements of $\boldsymbol{P}'$ and the triangulated world points using the N manually selected correspondences (total $12 + 3N$ parameters). The cost function $\mathcal{C}$ here is the reprojection error calculated by projecting the triangulated point back on the image sensor planes.

$$\mathcal{C} = ||\boldsymbol{X} - \boldsymbol{f}(\boldsymbol{p})||^2$$

The triangulation for the world point $\boldsymbol{Q}_i$ using correspondence $(\boldsymbol{x}_i \leftrightarrow \boldsymbol{x}'_i)$ is done by solving the following homogeneous set of equations

$$\boldsymbol{A}_i \boldsymbol{Q}_i = \boldsymbol{0}$$

where

$$\boldsymbol{A}_i = \begin{bmatrix} x_i \boldsymbol{P}^{3\top} - \boldsymbol{P}^{1\top} \\ y_i \boldsymbol{P}^{3\top} - \boldsymbol{P}^{2\top} \\ x'_i \boldsymbol{P}'^{3\top} - \boldsymbol{P}'^{1\top} \\ y'_i \boldsymbol{P}'^{3\top} - \boldsymbol{P}'^{2\top} \end{bmatrix}$$

and $\boldsymbol{P}^{k\top}$ is the $k-$th row of projection matrix $\boldsymbol{P}$ (similar for primed counter-parts).

- The rectification homography for the right image $\boldsymbol{H}'$ can be calculated using a series of homographies:
  - Translate the origin to the center of the image by

$$\boldsymbol{T} = \begin{bmatrix} 1 & 0 & -w/2 \\ 0 & 1 & -h/2 \\ 0 & 0 & 1 \end{bmatrix}$$

    where w, h are the width and height of the image.
  - Rotate the image to align the right epipole with the x-axis

$$\theta = -\arctan \frac{e'_y - h/2}{e'_x - w/2}$$

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

  - Send the right epipole to the ideal point along the x-axis

$$f = \cos\theta(e'_x - w/2) - \sin\theta(e'_y - h/2)$$

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

  The final $\boldsymbol{H}'$ is given by

$$\boldsymbol{H}' = \boldsymbol{T}^{-1}\, \boldsymbol{G}\, \boldsymbol{R}\, \boldsymbol{T}$$

- The rectification homography for the left image $\boldsymbol{H}$ can be calculated using a series of steps:
  - Define homographies $\boldsymbol{H}_0$ and $\boldsymbol{H}_A$

$$\boldsymbol{H}_0 = \boldsymbol{H}'\, \boldsymbol{P}'\, \boldsymbol{P}^{\dagger}$$

$$\boldsymbol{H}_A = \begin{bmatrix} a & b & c \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

  - Let $\hat{\boldsymbol{x}}_i = \boldsymbol{H}_0 \boldsymbol{x}_i$ and $\hat{\boldsymbol{x}}'_i = \boldsymbol{H}' \boldsymbol{x}'_i$
  - Solve for a, b, c using

$$\arg\min \sum_i (a\hat{x}_i + b\hat{y}_i + c - \hat{x}'_i)^2$$

  The final $\boldsymbol{H}$ is given by

$$\boldsymbol{H} = \boldsymbol{H}_A\, \boldsymbol{H}_0$$

2.1.2 Interest Point Detection

Now that we have rectified the left and right images using rectification homographies $\boldsymbol{H}, \boldsymbol{H'}$; we can find a larger set of correspondences ($\boldsymbol{x}_i \leftrightarrow \boldsymbol{x}'_i$) because the correspondences now lie in the same row (or adjoining rows for robustness). We can use the Canny edges of the rectified images as potential interest points, and then proceed to find matches using any metric (for example SSD). Now the search space for matching is significantly smaller. These interest points in the rectified images are then transformed back to the original images using the inverse of the rectification homographies.

2.1.3 Projective Reconstruction

We can re-calculate a refined fundamental matrix $\boldsymbol{F}$ using the interest points detected using canny edges (1). Using this refined $\boldsymbol{F}$, we can calculate the projection matrices $\boldsymbol{P}, \boldsymbol{P'}$ of the canonical cameras from (2), (3). The world points of the correspondences are calculated using non-linear least squares.

2.2 Results

(a) left.jpg (b) right.jpg

Figure 2: Images for this task

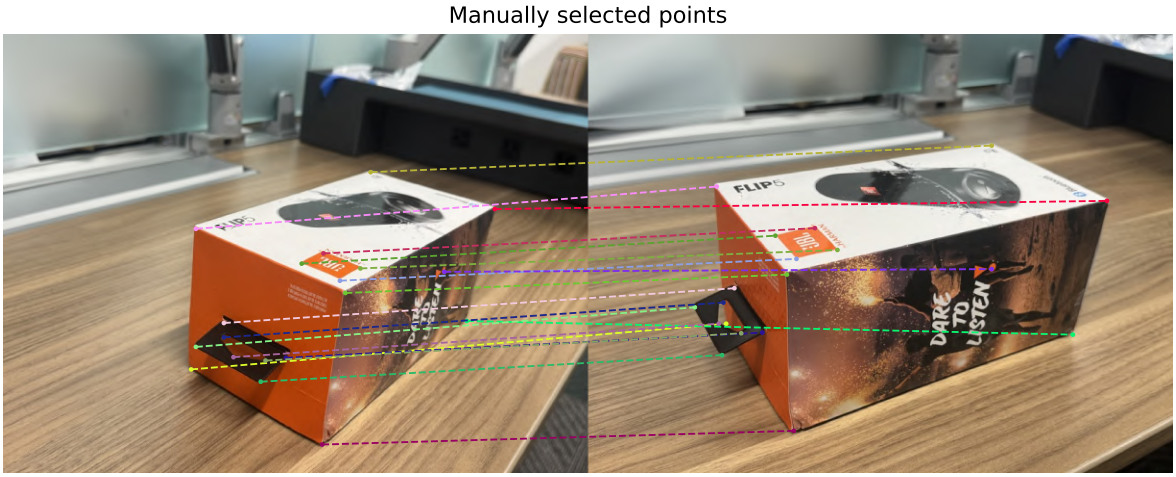


Figure 3: Manually selected correspondences

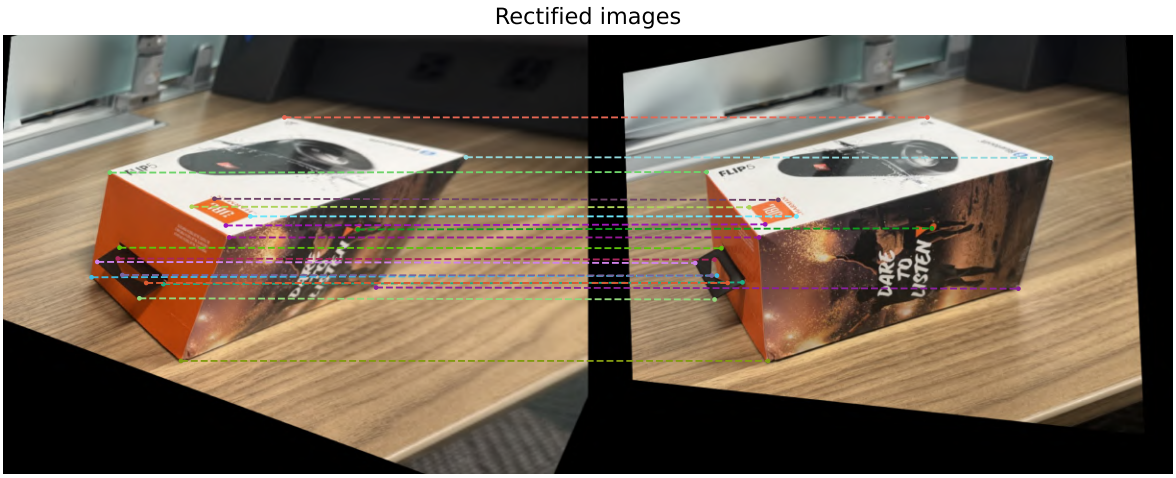


Figure 4: Rectified images

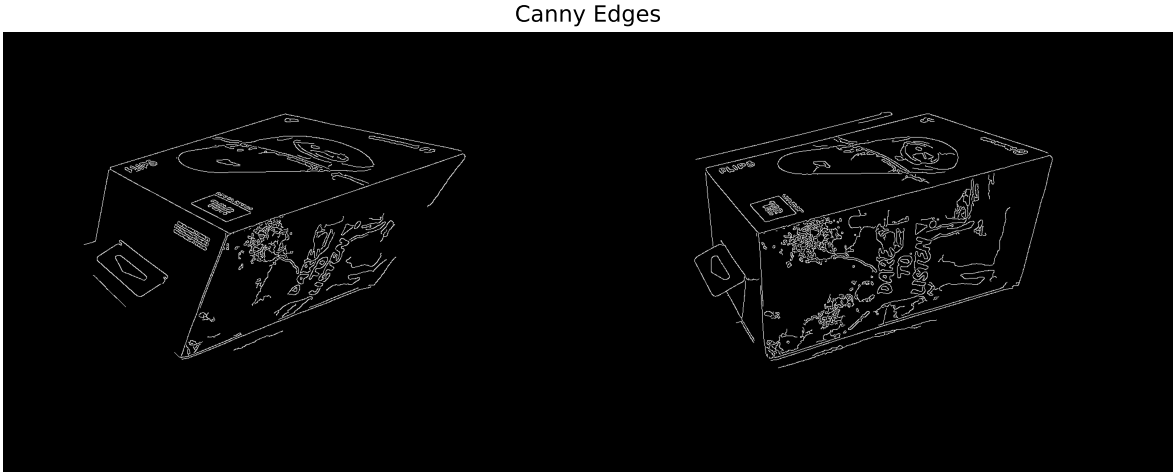


Figure 5: Canny edges of the rectified images

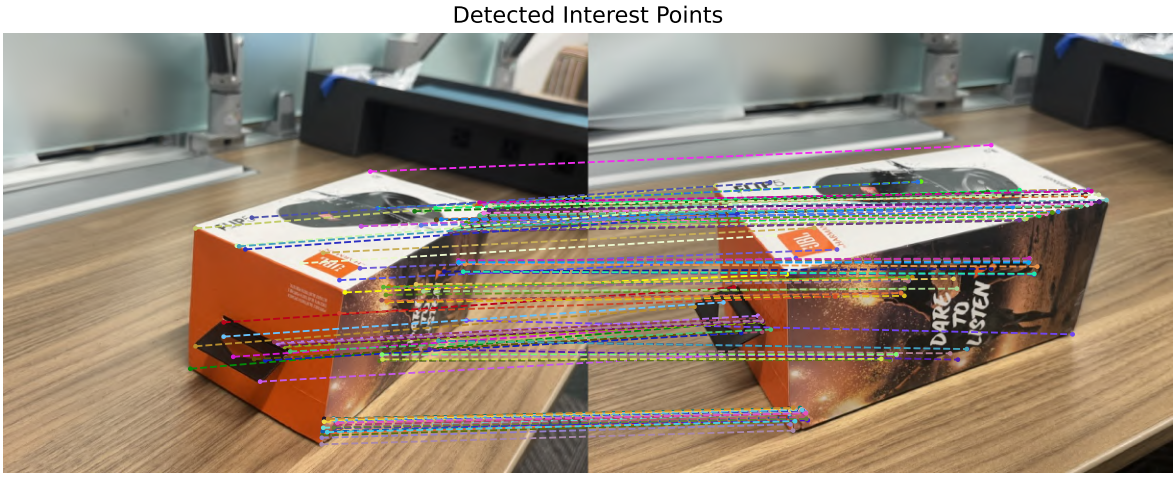


Figure 6: Detected interest points using Canny and SSD

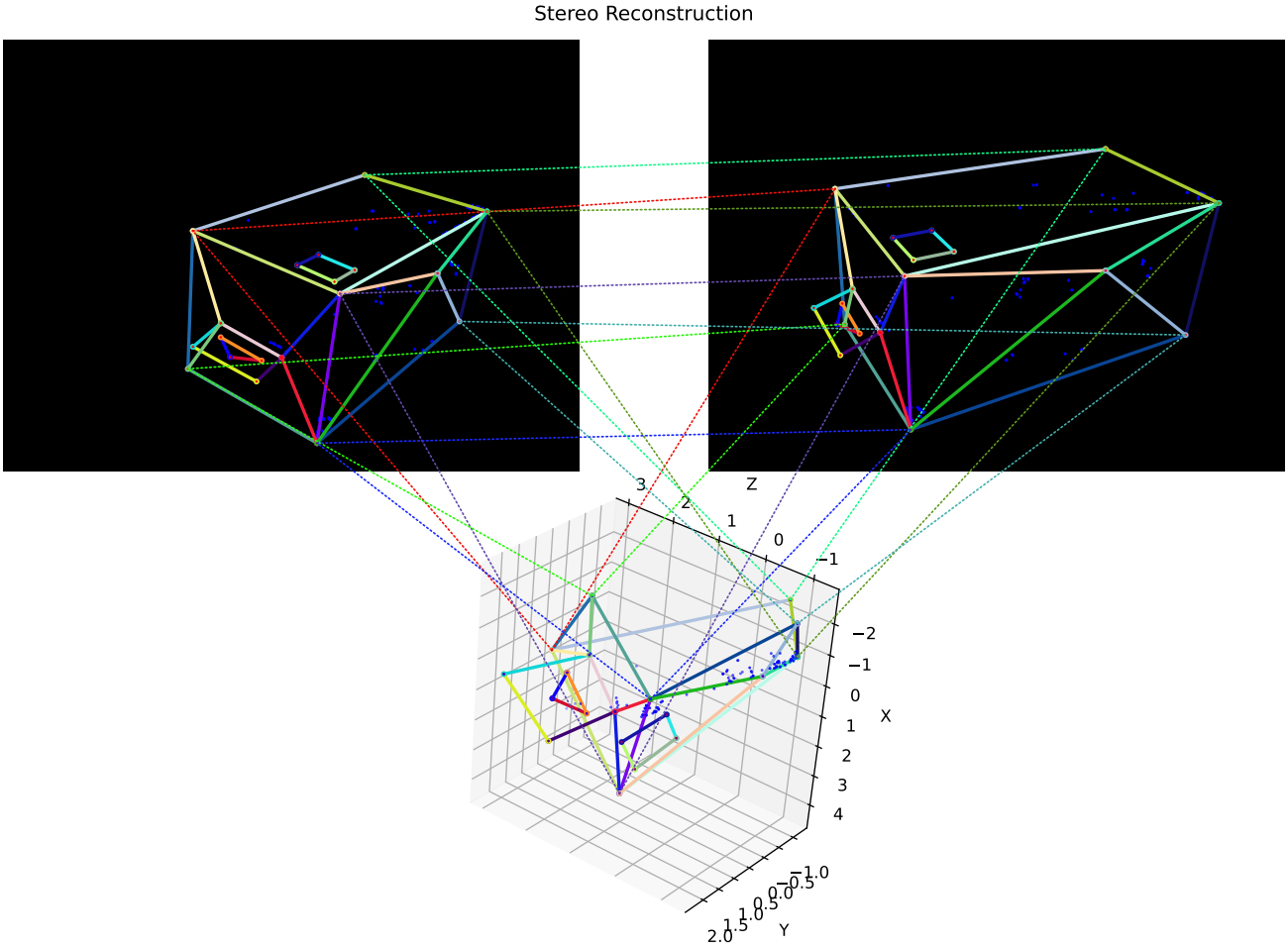


Figure 7: View 1 of the stereo reconstruction. Source images not shown; just outlines shown for better visibility.

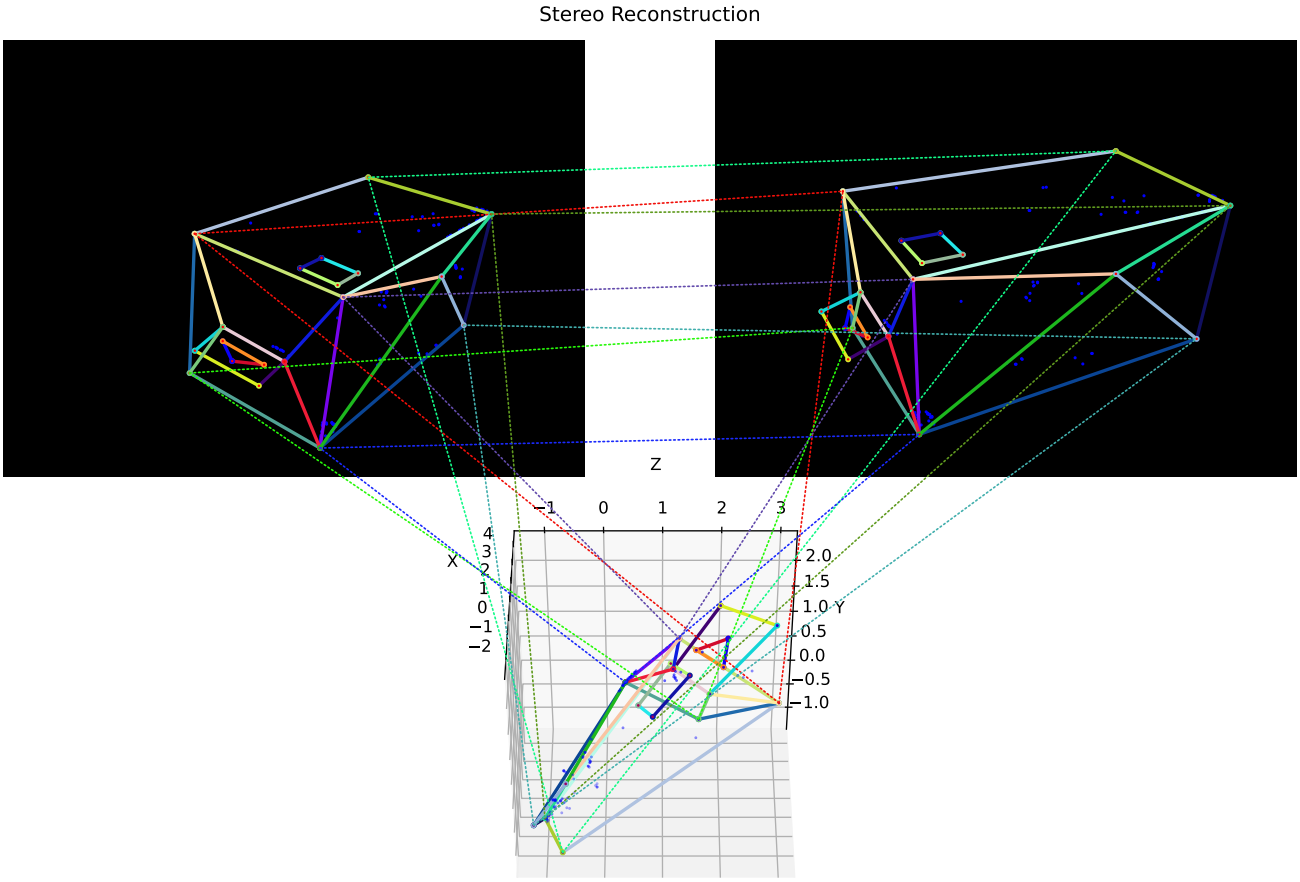


Figure 8: View 2 of the stereo reconstruction. Source images not shown; just outlines shown for better visibility.

3 Task - 2: Loop And Zhang

3.1 Methodology

The Loop and Zhang algorithm for image rectification involves breaking down the rectification homographies as a product of a purely projective homography and an affine homography.

$$\boldsymbol{H} = \boldsymbol{H}_a \ \boldsymbol{H}_p$$
$$\boldsymbol{H}' = \boldsymbol{H}'_a \ \boldsymbol{H}'_p$$

The purely projective homographies $\boldsymbol{H}_p, \boldsymbol{H}'_p$ are homographies that send the epipoles $\boldsymbol{e}, \boldsymbol{e}'$ to infinity in a direction that causes the least distortion to the images.
The affine homographies $\boldsymbol{H}_a, \boldsymbol{H}'_a$ are primarily for mitigating the distortions that are caused by highly-distortive projective homographies. The affine homographies can be further broken down into shearing and similarity homographies.

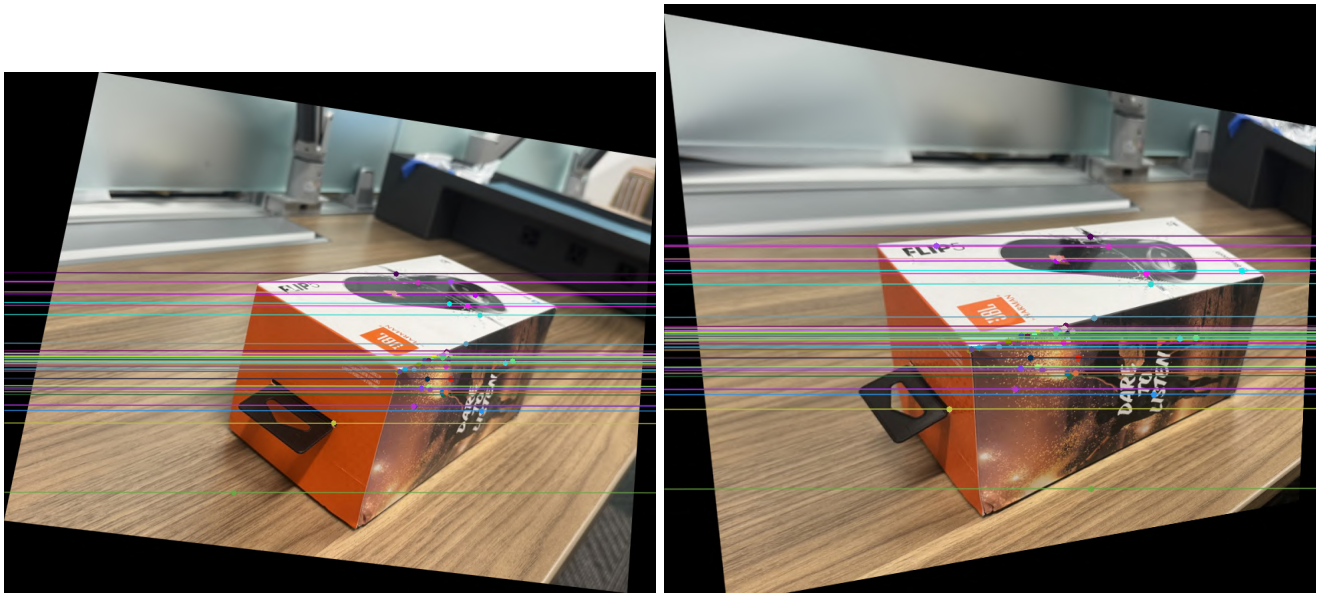
3.2 Discussion

The set of rectified images returned by the Loop and Zhang algorithm have lower projective distortion but detects fewer number of correspondences because it uses the BRISK (open source alternative to SIFT). My pipeline uses canny edges as potential interest points and is effective in finding large number of correspondences for the calculation of the fundamental matrix and camera projection matrices (up to a canonical configuration).

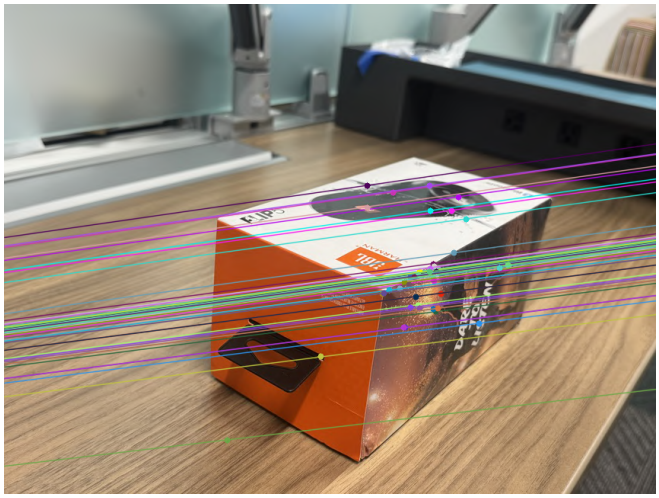
3.3 Results

(a) left.jpg (b) right.jpg

Figure 9: Images for this task



(a) Left image rectified (b) Right image rectified



(a) Matched interest points in left image

(b) Matched interest points in right image

4 Task - 3: Dense Stereo Matching

4.1 Methodology

The census transform is a simple method for dense stereo matching, involving the following steps:

1. For each pixel in the left image, (x_i, y_i), a candidate correspondence pixel in the right image is selected as $(x_i' - d, y_i')$, where d is the disparity value under consideration.
2. Around each pixel in the correspondence pair, an $M \times N$ neighborhood is defined.
3. For each $M \times N$ neighborhood, an $M \times N$ bit-vector is constructed. A bit in the bit-vector is set to 1 if the corresponding pixel value in the neighborhood is strictly greater than the central pixel's value; otherwise, it is set to 0.
4. Two bit-vectors are generated at each iteration: one for the left image and one for the right image. These bit-vectors are then compared using a bit-wise XOR operation. The Hamming distance (the number of 1's in the XOR result) is computed, representing the data cost for the candidate correspondence.
5. The disparity map is populated by finding the disparity d that minimizes the data cost for each pixel position in the left image.

4.2 Discussion

Overall, the estimated disparity maps using census transforms are good and accurate to ground truth disparity maps. The accuracy does not change much when we use a square window of size 11×11 or 21×21. Relaxing the distance threshold (δ) increases the accuracy by $2 - 3\%$. Using a rectangular window of size 11×21 also does not change the accuracy by a lot.

4.3 Results

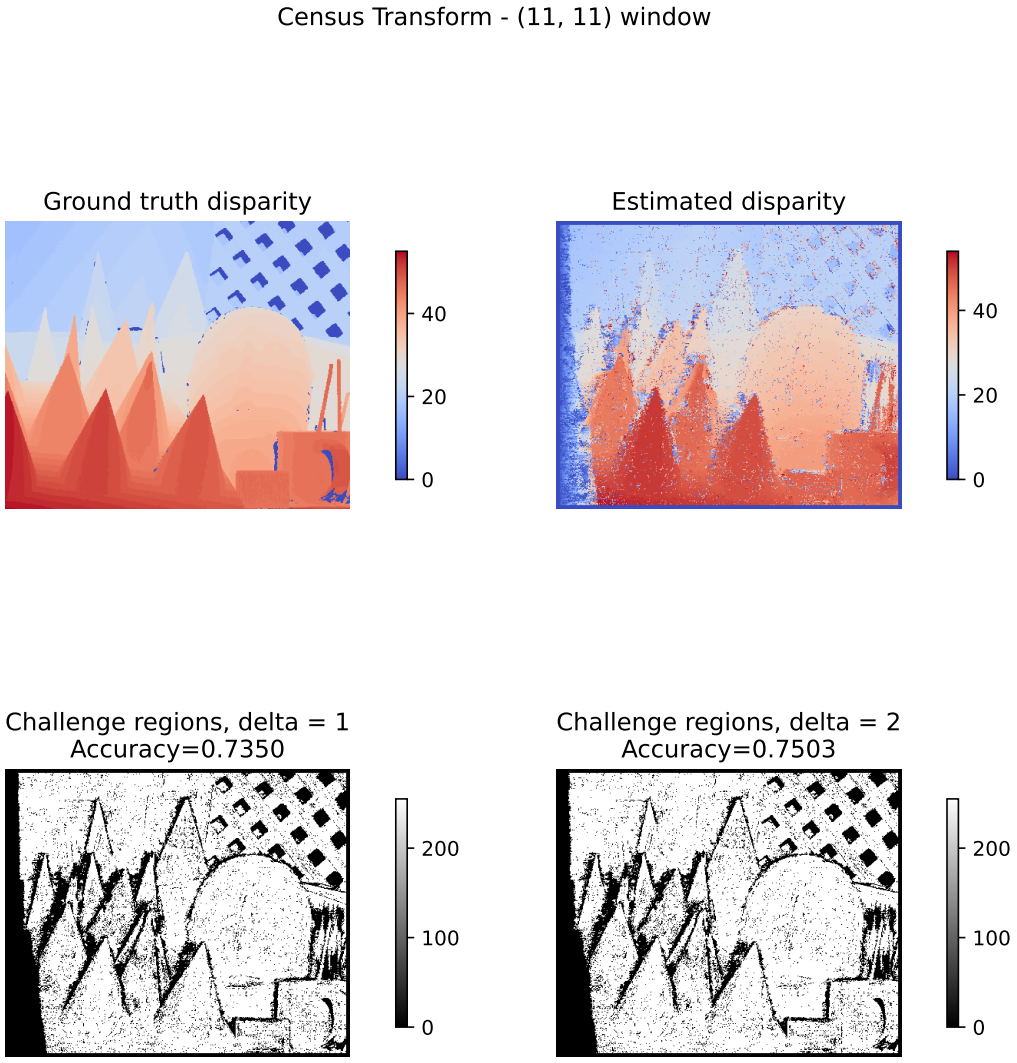


Figure 12

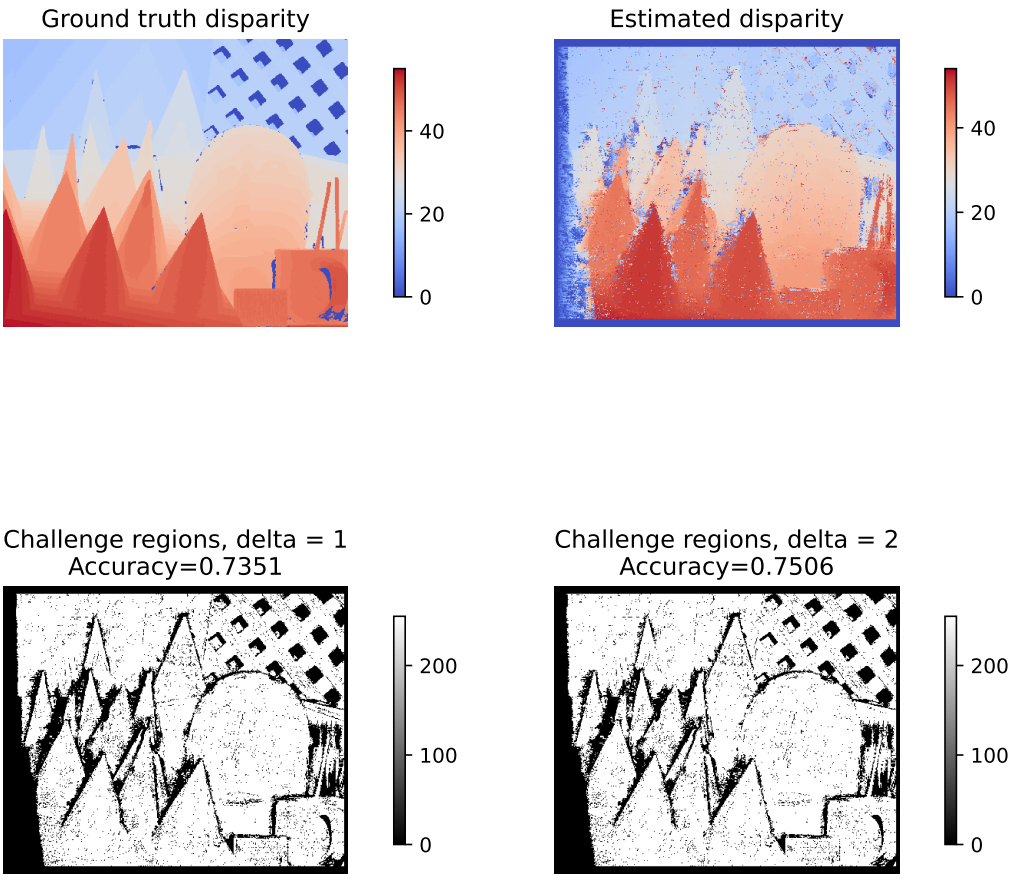

Census Transform - (21, 11) window
Ground truth disparity
40
20
0
Estimated disparity
40
20
0
Challenge regions, delta = 1
Accuracy=0.7351
200
100
0
Challenge regions, delta = 2
Accuracy=0.7506
200
100
0

Figure 13

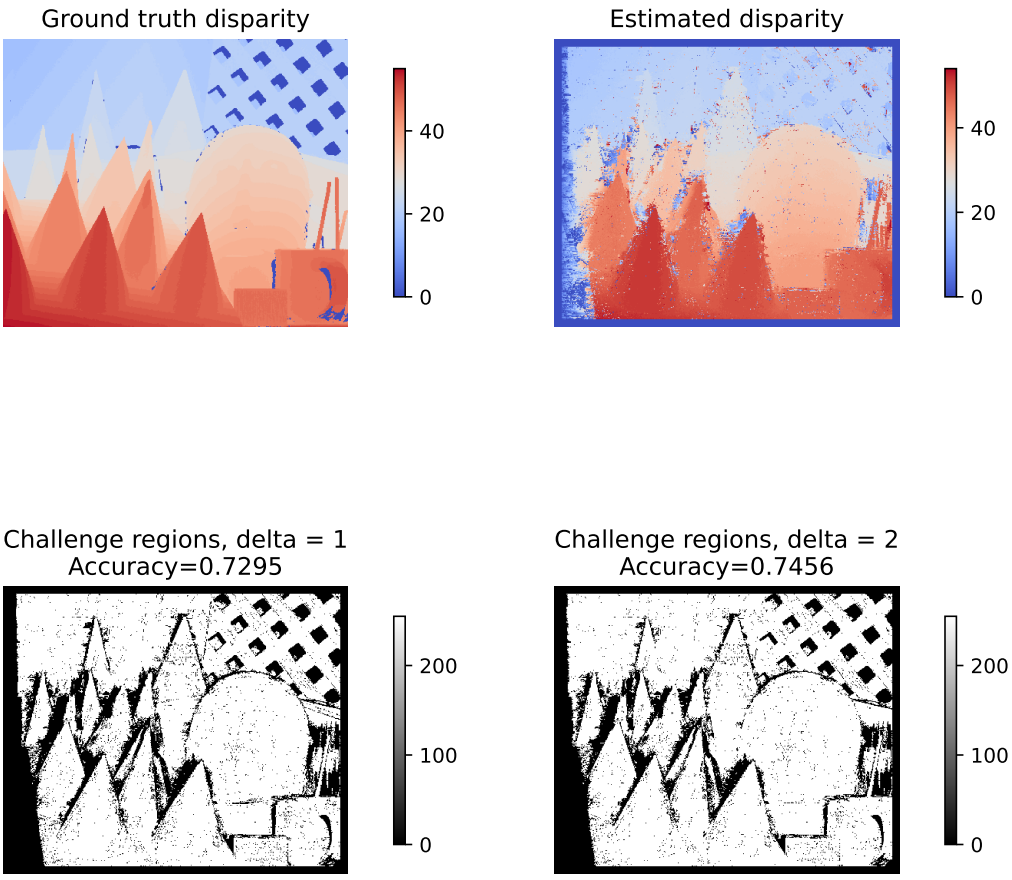

Census Transform - (21, 21) window
Ground truth disparity
40
20
0
Estimated disparity
40
20
0
Challenge regions, delta = 1
Accuracy=0.7295
200
100
0
Challenge regions, delta = 2
Accuracy=0.7456
200
100
0

Figure 14

5 Task - 4: Depth Maps and Automatic Extraction of Dense Correspondences

5.1 Methodology

Automatic extraction of dense correspondences using depth maps leverages the geometric relationship between stereo images and their depth information for efficient and accurate correspondence matching. The process is outlined below:

1. **Depth to 3D Projection:** The depth value $D(\boldsymbol{x})$ for each pixel in the first image is combined with the camera's intrinsic and extrinsic parameters to compute the 3D coordinates of the pixel in the world coordinate system:
$$\boldsymbol{X}_{\text{world}} = \boldsymbol{T}^{-1}\ \boldsymbol{X}_{\text{cam}}, \quad \boldsymbol{X}_{\text{cam}} = D(\boldsymbol{x})\ \boldsymbol{K}^{-1}\ \boldsymbol{x},$$
where $\boldsymbol{K}$ is the intrinsic camera matrix, $\boldsymbol{T}$ is the transformation from the camera to the world frame, and $\boldsymbol{x}$ is the pixel's homogeneous image coordinates.

2. **Reprojection to the Second Image:** The computed 3D world coordinates $\boldsymbol{X}_{\text{world}}$ are re-projected onto the second image using the second camera's intrinsic and extrinsic parameters:
$$\boldsymbol{x}' = \boldsymbol{K}'\ [\boldsymbol{R}' \mid \boldsymbol{t}']\ \boldsymbol{X}_{\text{world}},$$
resulting in a candidate pixel $\boldsymbol{x}'$ in the second image.

3. **Depth Check for Validation:** The depth of the candidate pixel in the second image, $D'(\boldsymbol{x}')$, is compared with the depth derived from reprojection. If the absolute difference is below a threshold δ, the correspondence is validated:
$$\left| D'(\boldsymbol{x}') - \hat{D}(\boldsymbol{x}') \right| < \delta.$$

4. **Output Correspondences:** Validated correspondences are recorded as dense matches between the two images.

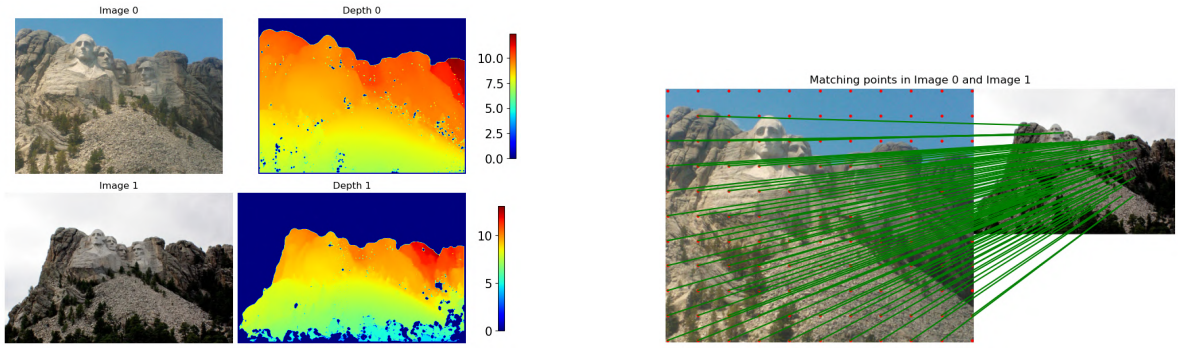
Usefulness:

1. **Efficiency:** Automates dense correspondence extraction without requiring exhaustive searches.

2. **Precision:** Ensures geometric consistency using depth maps.

3. **Applications:** Enables tasks such as stereo matching, image registration, and 3D reconstruction, while also facilitating training of deep-learning-based feature extraction networks like SuperPoint and LoFTR.

5.2 Results



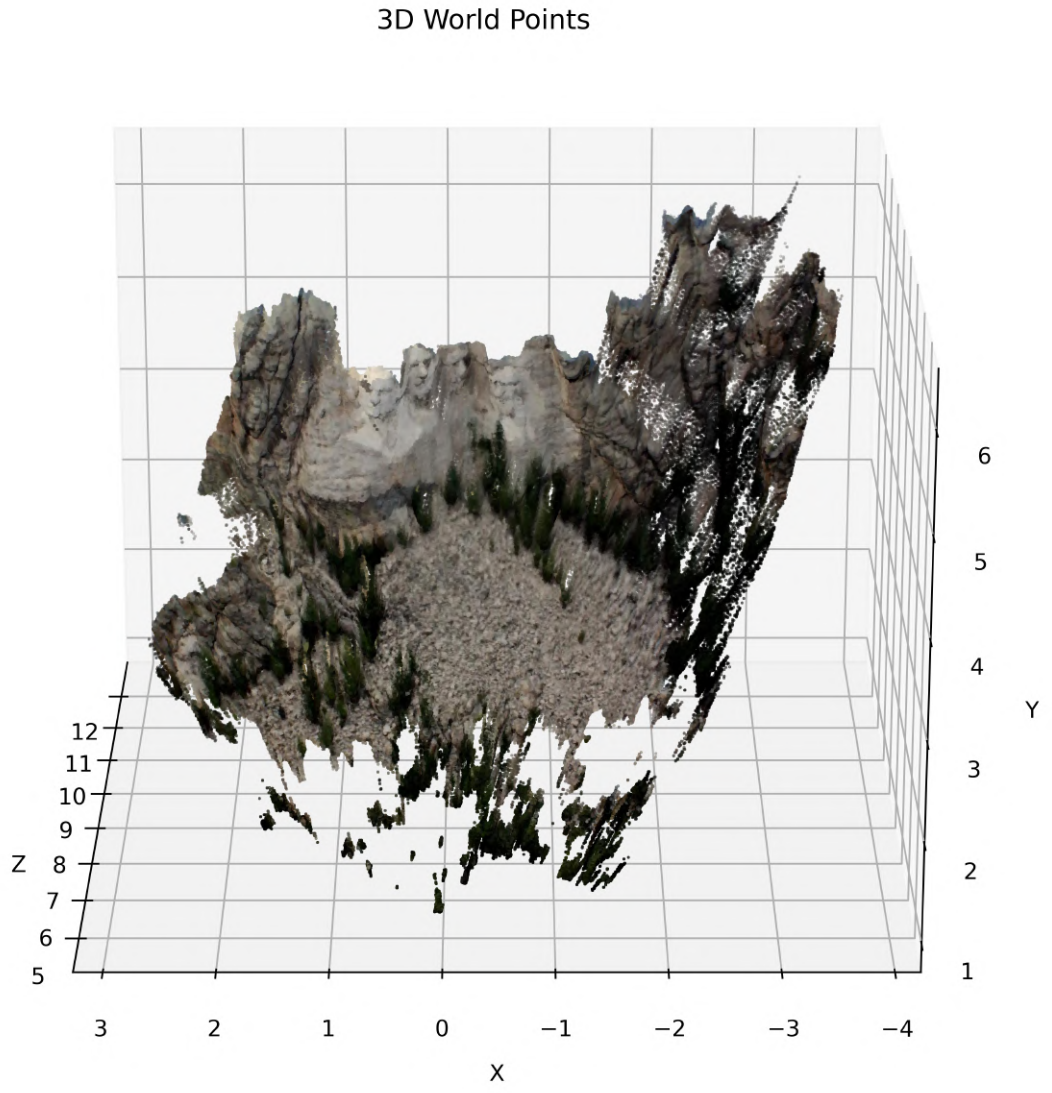
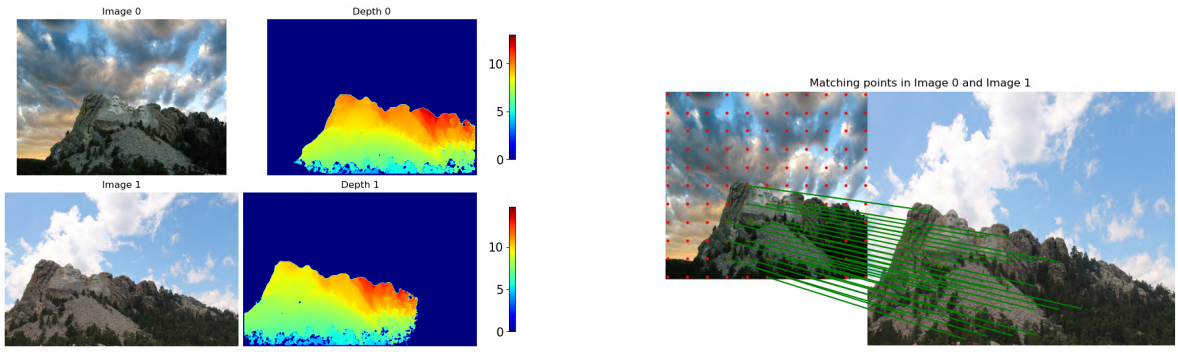


Figure 15: Pair 0



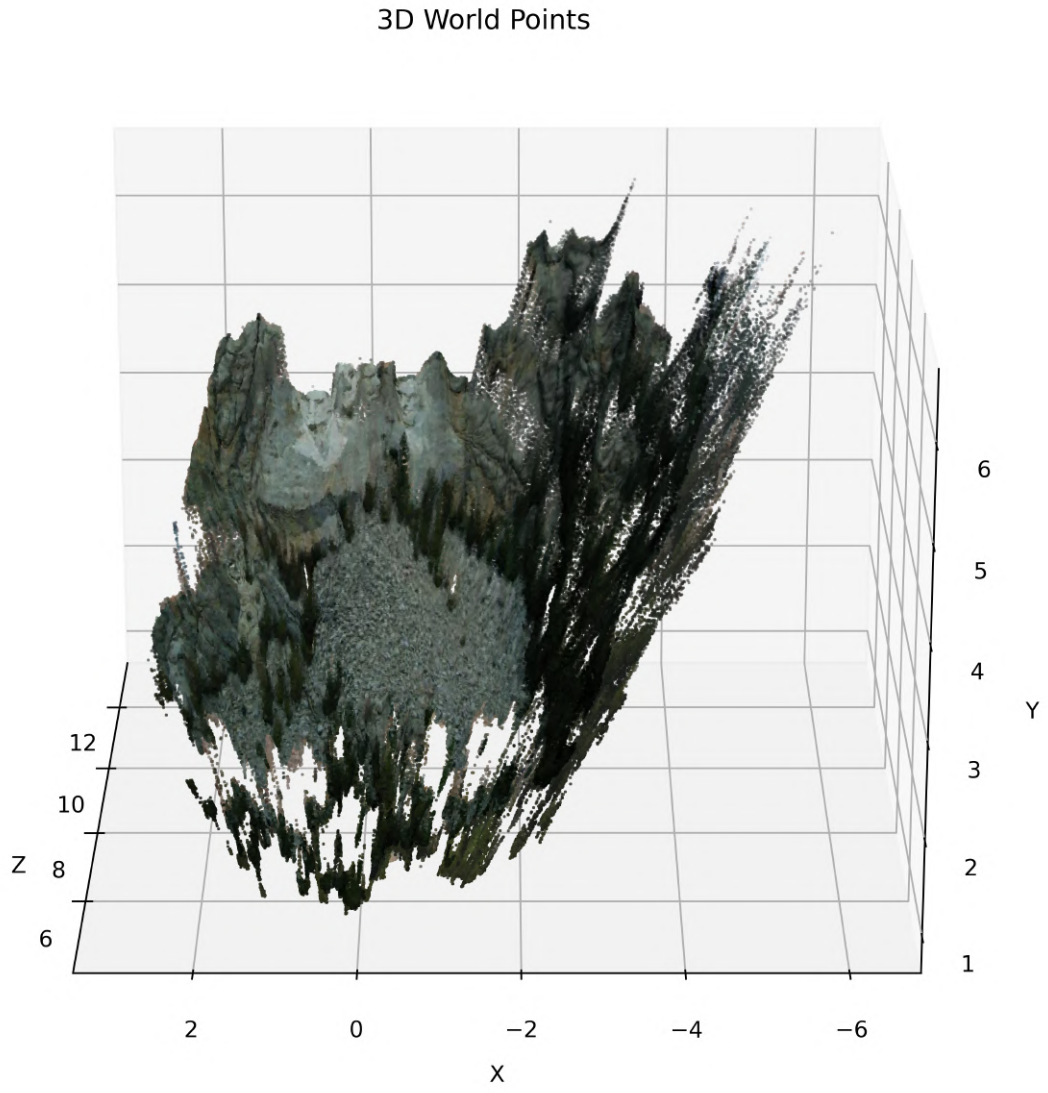
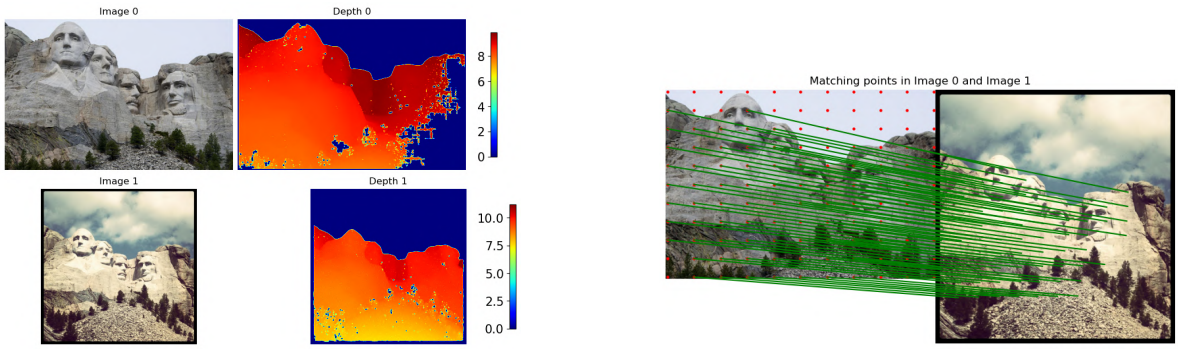


Figure 16: Pair 1



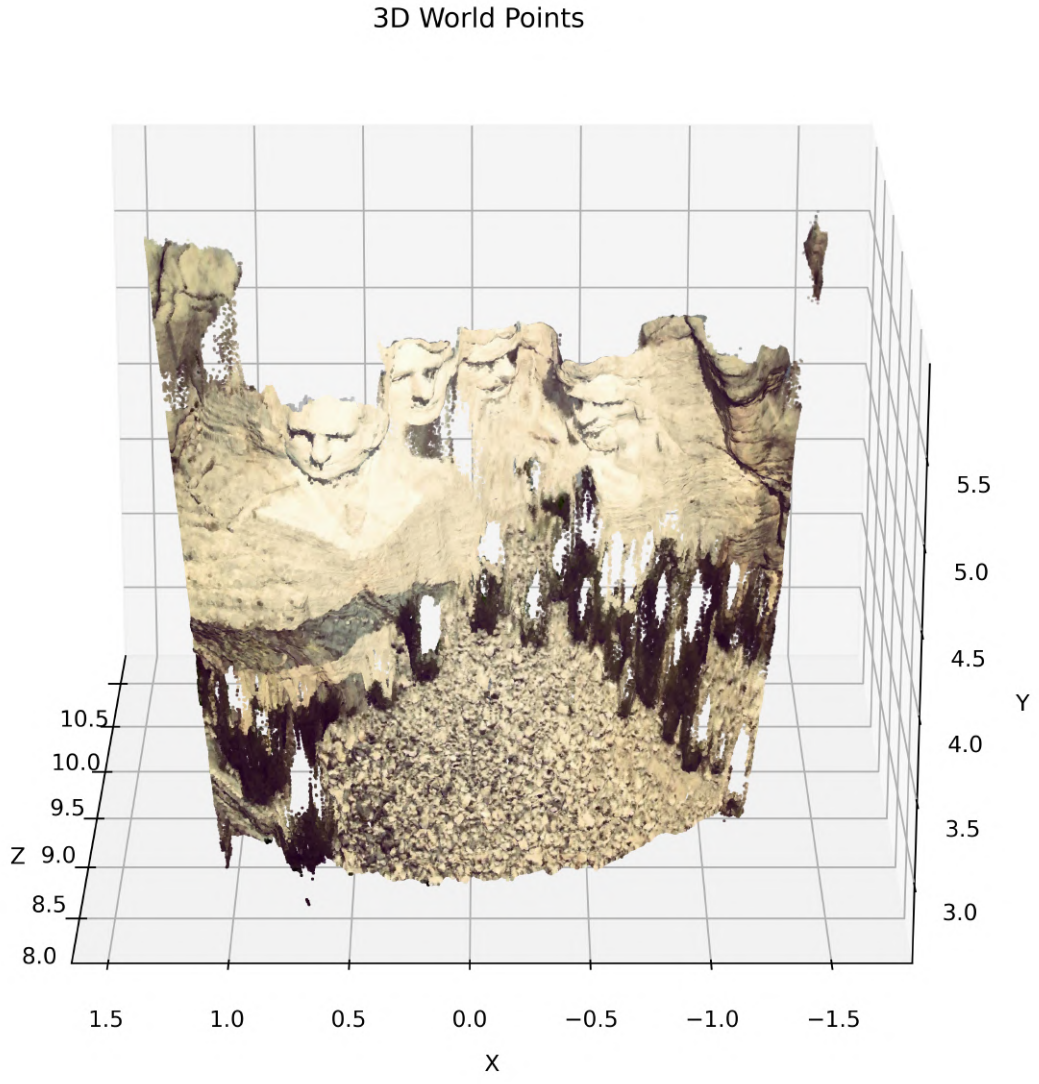
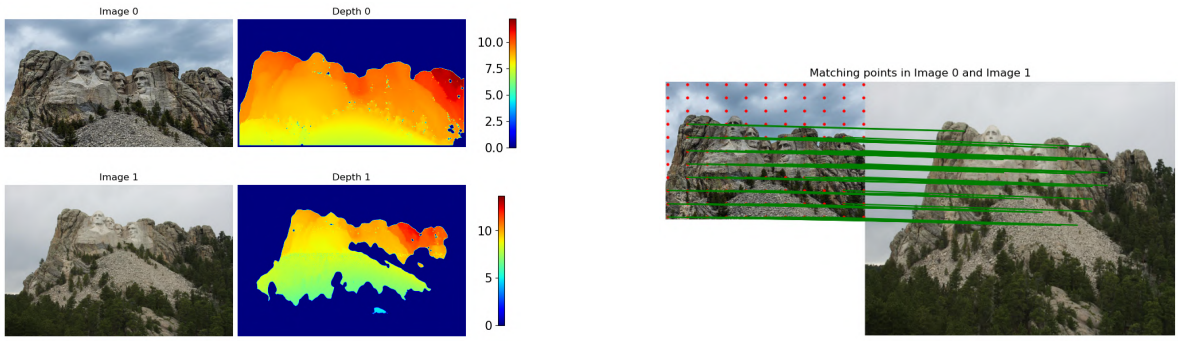


Figure 17: Pair 2



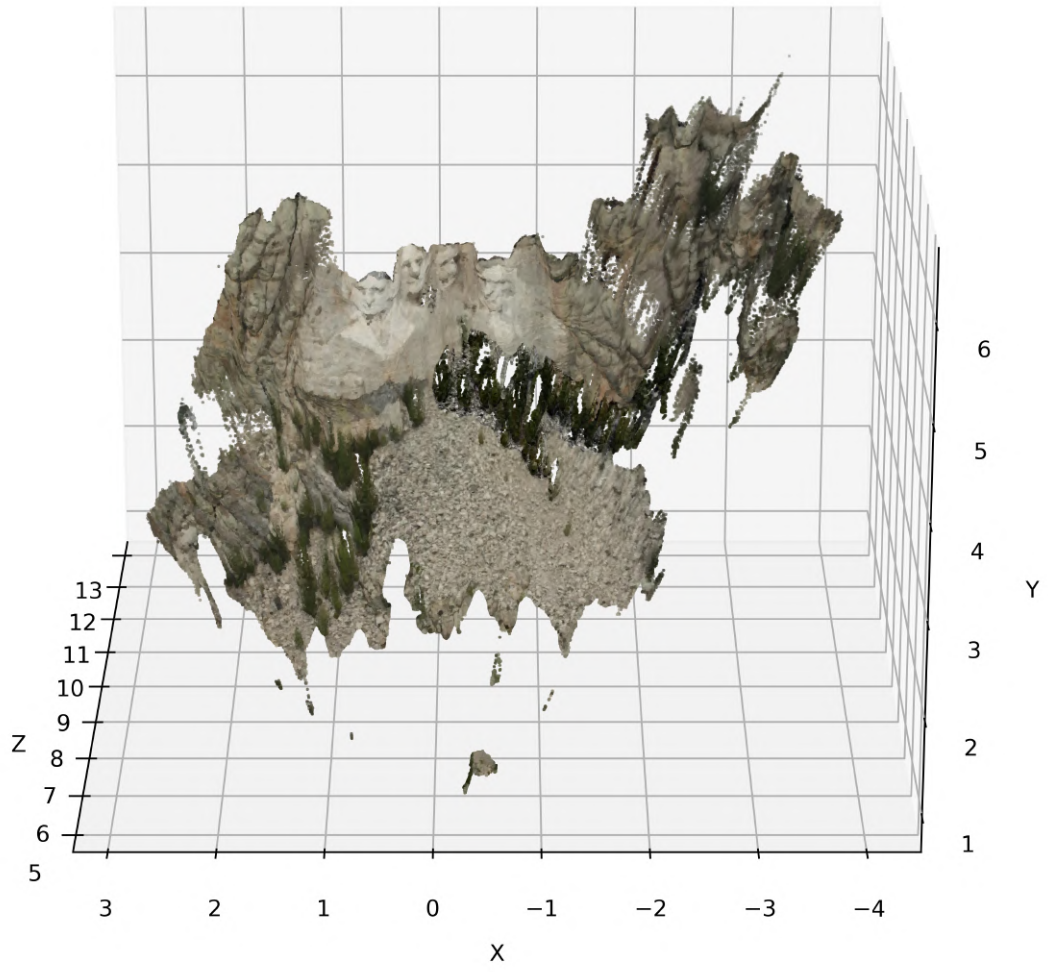
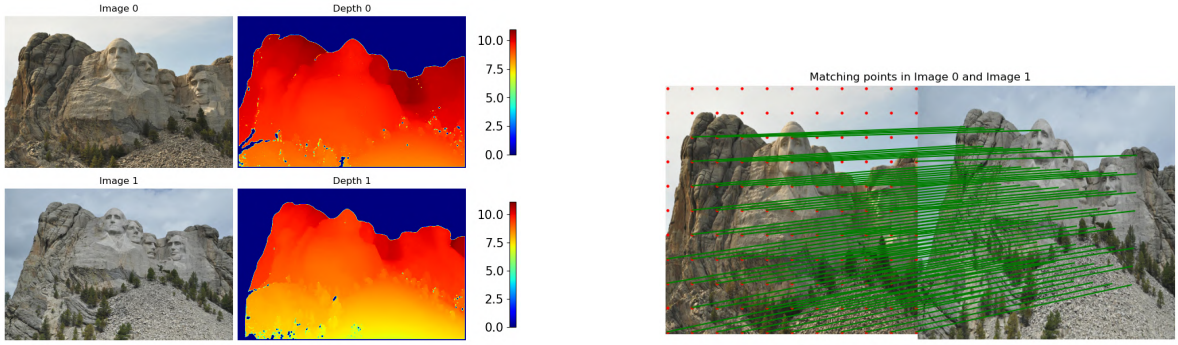


Figure 18: Pair 3



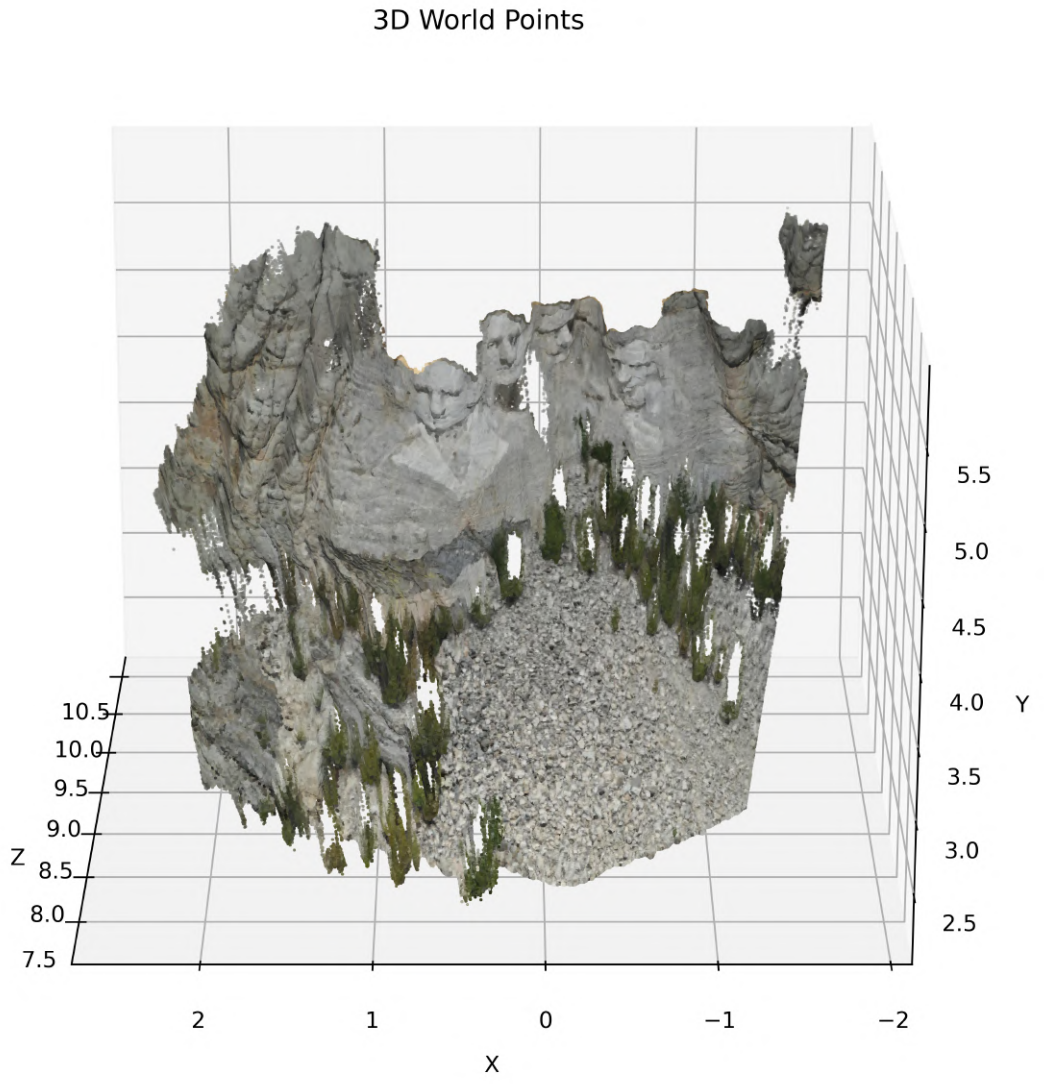
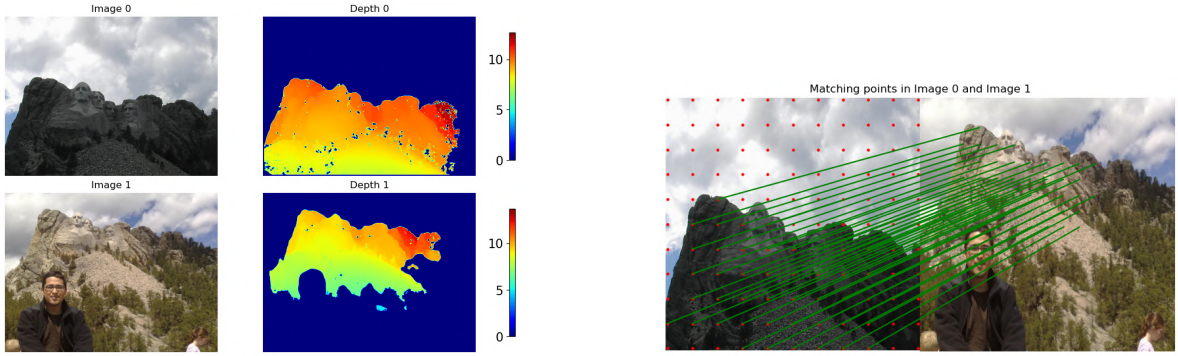


Figure 19: Pair 4



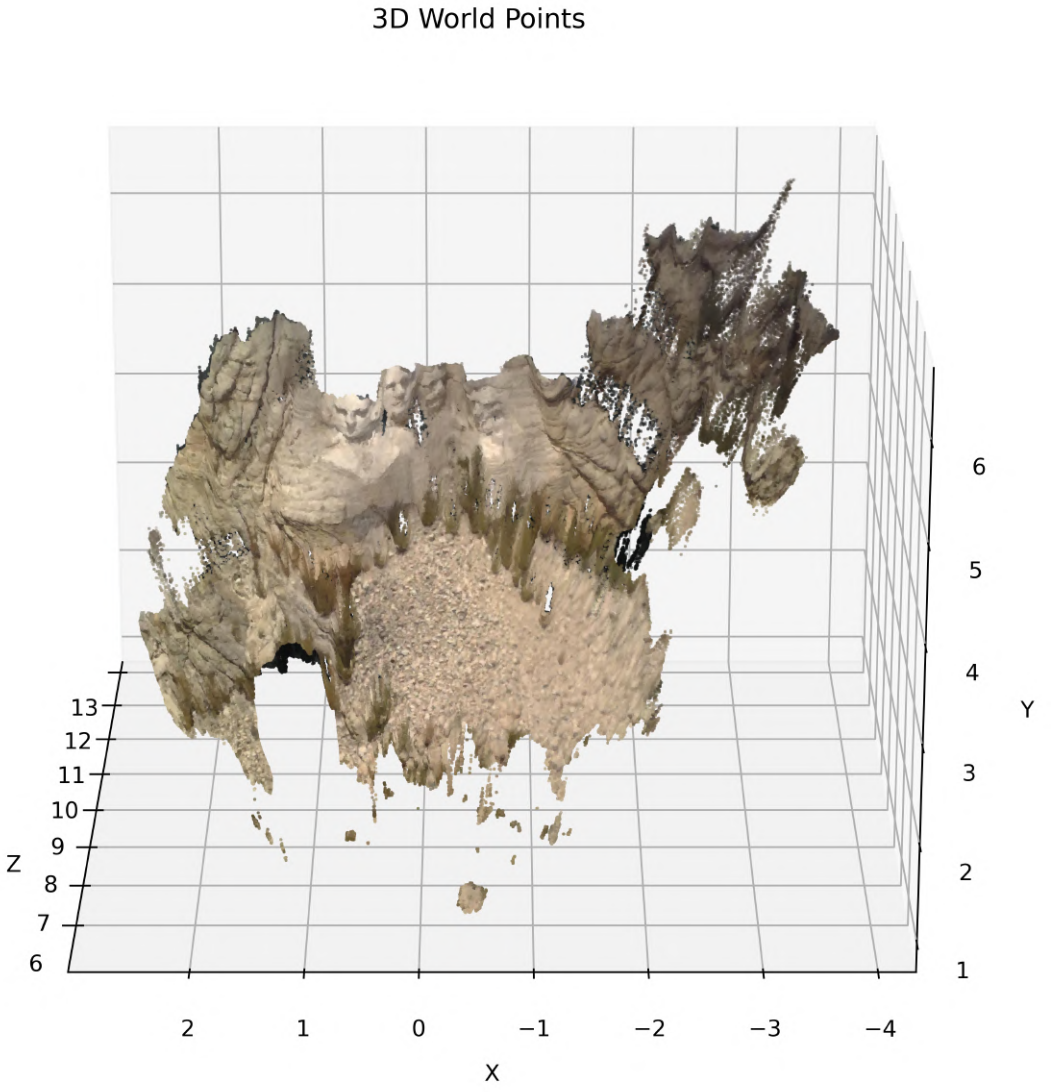
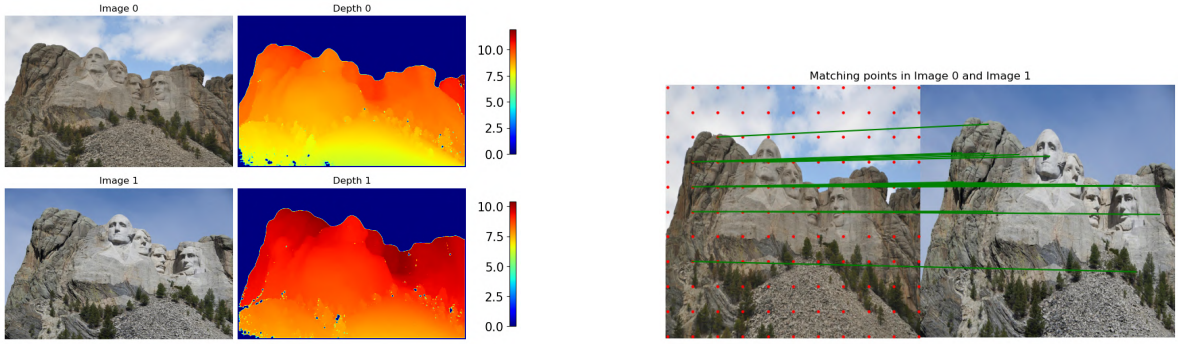


Figure 20: Pair 5



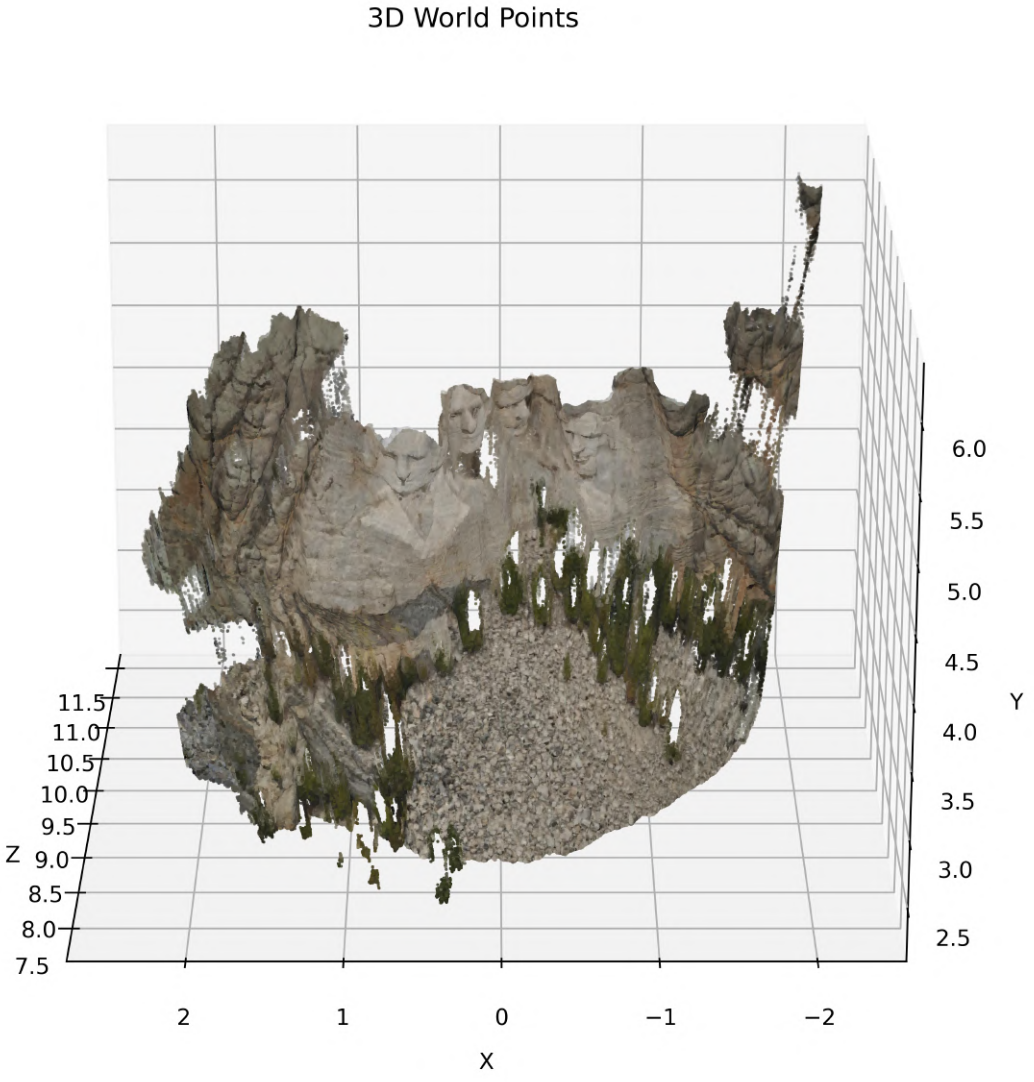
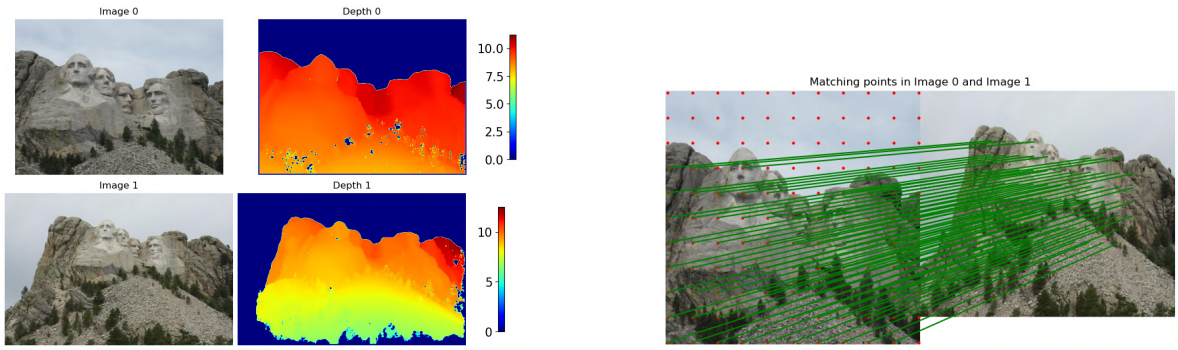


Figure 21: Pair 6



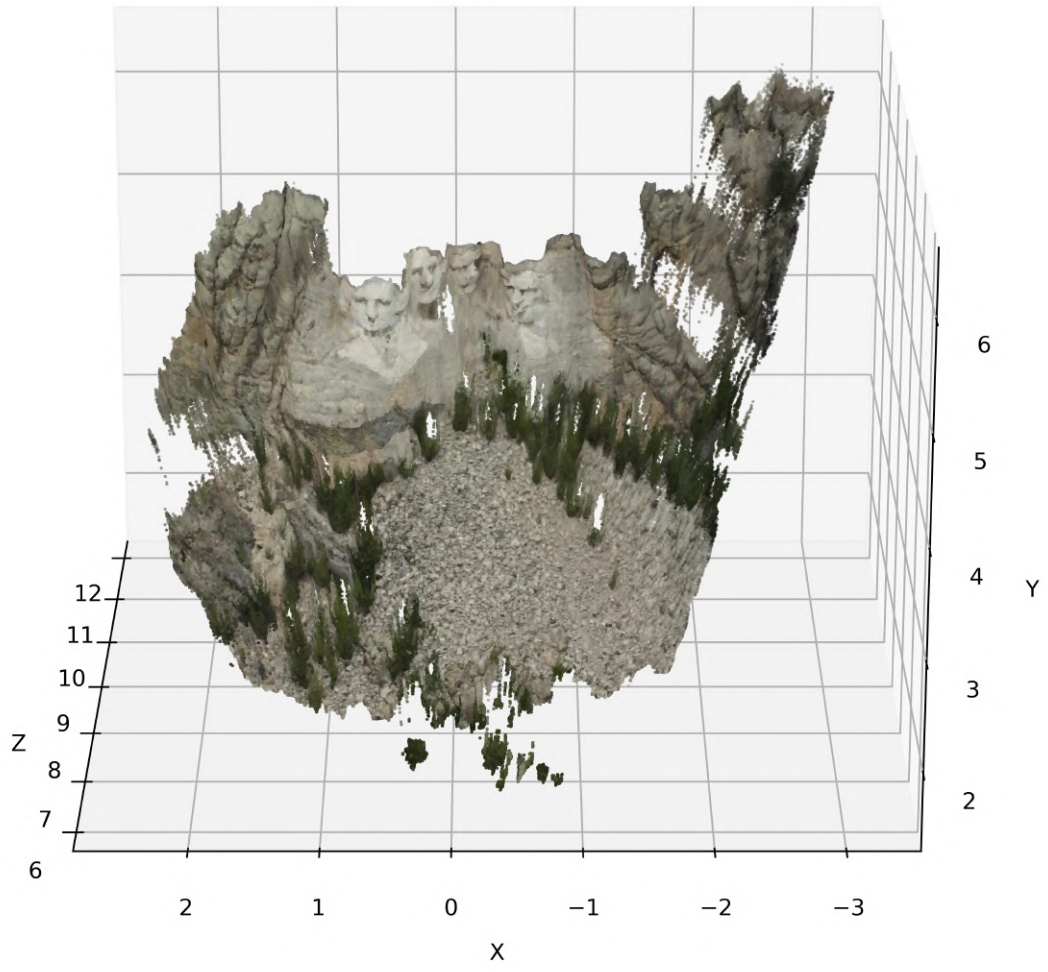
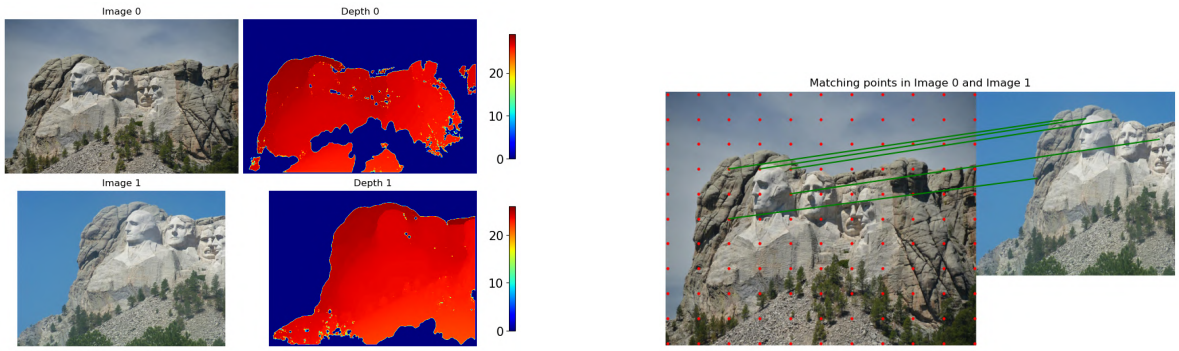


Figure 22: Pair 7



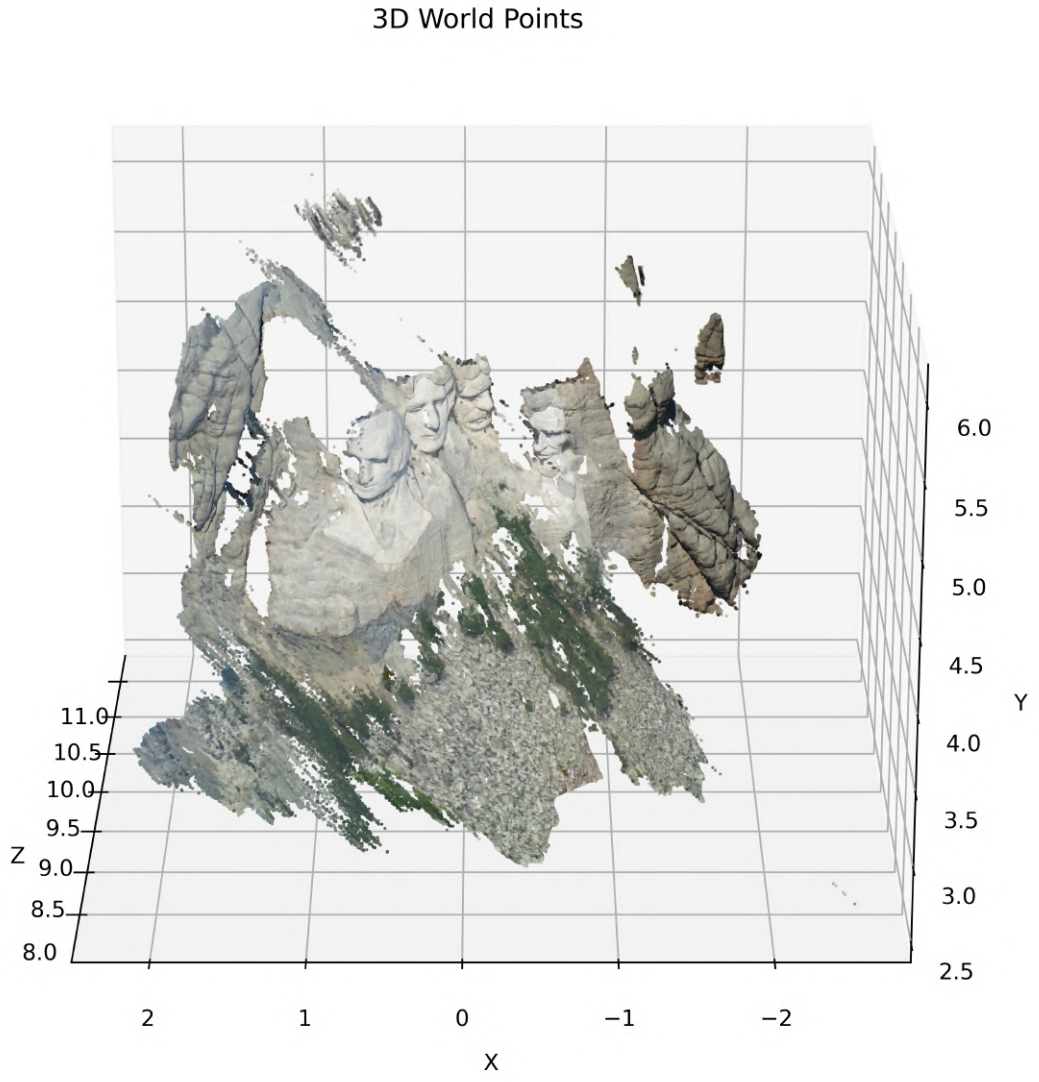


Figure 23: Pair 8

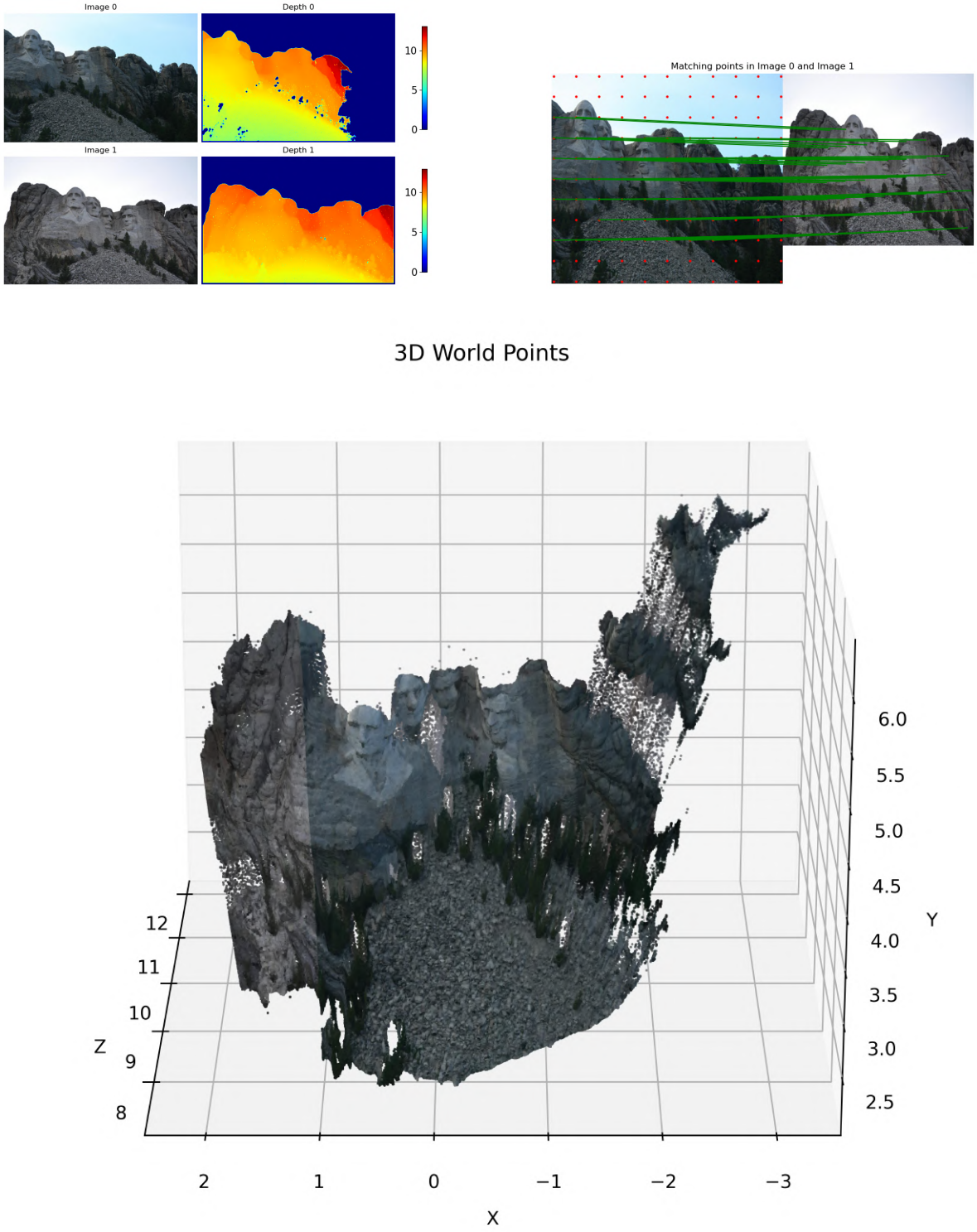


Figure 24: Pair 9

6 Source Code

6.1 Task 1 and 3

6.1.1 stereo_cameras.py

```python
import cv2 as cv
import numpy as np
import os
from scipy.optimize import least_squares
from tqdm import tqdm
from skimage import feature

from LM_optim import LM_optim

eps = np.finfo(float).eps


class ClickRecorder:
    """
    Class to record clicked points.
    """

    def __init__(self):
        self.clicked_points = []

    def mouse_callback(self, event, x, y, flags, param):
        if event == cv.EVENT_LBUTTONDOWN:
            self.clicked_points.append((x, y))
            print(f"Clicked point: {self.clicked_points[-1]}")

    def get_clicked_points(self):

        return self.clicked_points


def select_points(img):
    """
    Function to select and return selected points on an image using OpenCV's mouse callback.
    """
    click_recorder = ClickRecorder()
    cv.namedWindow("select_points", cv.WINDOW_NORMAL)
    cv.imshow("select_points", img)
    cv.setMouseCallback("select_points", click_recorder.mouse_callback)
    cv.waitKey(0)
    print(f"All selected points: {click_recorder.get_clicked_points()}")
    cv.destroyAllWindows()

    return np.array(click_recorder.get_clicked_points())


def _get_cross_product_matrix(a):
    """
    Function to calculate the cross product matrix for a 3D vector a.
    """
    assert len(a) == 3

    return np.array([[0, -a[2], a[1]],
                     [a[2], 0, -a[0]],
                     [-a[1], a[0], 0]])


def _normalize_coords(points):
    """
    Function to normalize 2D points using the mean and standard deviation.
    """
    mu = np.mean(points, axis=0)
    dist = np.sqrt(np.sum((points - mu)**2, axis=1))
    mean_dist = np.mean(dist)
    c = np.sqrt(2) / mean_dist
    T = np.array([[c, 0, -c*mu[0]],
```

```
                    [0, c, -c*mu[1]],
                    [0, 0, 1]])
    points_hc = np.hstack((points, np.ones((len(points), 1)))).T
    points_normalized = T @ points_hc

    return points_normalized[:2, :].T, T


def _calculate_F_from_points(l_points, r_points, l_T, r_T):
    """
    Function to calculate Fundamental Matrix from two sets of corresponding points.
    """
    A = []
    for (x, y), (xp, yp) in zip(l_points, r_points):
        A.append([xp*x, xp*y, xp, yp*x, yp*y, yp, x, y, 1])

    A = np.array(A)
    _, _, vt = np.linalg.svd(A)
    F = vt[-1, :].reshape(3, 3)

    F = r_T.T @ F @ l_T
    u, d, vt = np.linalg.svd(F)
    d[-1] = 0
    F = u @ np.diag(d) @ vt
    F /= F[2, 2] + eps

    return F


def _calculate_epipoles(F):
    """
    Function to calculate the epipoles from the Fundamental Matrix.
    """
    _, _, vt = np.linalg.svd(F)
    l_epipole = vt[-1, :]
    l_epipole /= l_epipole[2] + eps

    _, _, vt = np.linalg.svd(F.T)
    r_epipole = vt[-1, :]
    r_epipole /= r_epipole[2] + eps

    return l_epipole, r_epipole


def _calculate_projection_matrices(F, r_epipole):
    """
    Function to calculate the projection matrices from the Fundamental Matrix and the right
    epipole.
    """
    l_P = np.eye(3, 4, dtype=np.float64)

    s = _get_cross_product_matrix(r_epipole)
    r_P = np.hstack((s @ F, r_epipole[:, None]))

    return l_P, r_P


def _triangulate(l_P, r_P, l_point, r_point):
    """
    Function to triangulate for the world point using the projection matrices.
    """
    l_P1 = l_P[0]
    l_P2 = l_P[1]
    l_P3 = l_P[2]

    r_P1 = r_P[0]
    r_P2 = r_P[1]
    r_P3 = r_P[2]

    A = []
    A.append([l_point[0] * l_P3 - l_P1])
```

```
    A.append([l_point[1] * l_P3 - l_P2])
    A.append([r_point[0] * r_P3 - r_P1])
    A.append([r_point[1] * r_P3 - r_P2])

    A = np.array(A).squeeze()
    _, _, vt = np.linalg.svd(A.T @ A)

    X = vt[-1, :]
    X /= X[3] + eps

    return X


def _cost_func_r_P(params, l_P, l_points, r_points):
    """
    Cost function to refine P' and world points using LM.
    """
    r_P = np.reshape(params[:12], (3, 4))

    X = []
    F = []

    for i, (l_point, r_point) in enumerate(zip(l_points, r_points)):

        X.extend(l_point)
        X.extend(r_point)

        idx = 12 + 3 * i

        world_point = np.array([*params[idx: idx+3], 1])

        l_point_reproj = l_P @ world_point
        l_point_reproj /= l_point_reproj[2] + eps

        r_point_reproj = r_P @ world_point
        r_point_reproj /= r_point_reproj[2] + eps

        F.extend(l_point_reproj[:2])
        F.extend(r_point_reproj[:2])

    X = np.array(X)
    F = np.array(F)

    return X - F


def _cost_func_world_points(params, l_P, r_P, l_points, r_points):
    """
    Cost function to refine world points using LM.
    """
    X = []
    F = []

    for i, (l_point, r_point) in enumerate(zip(l_points, r_points)):

        X.extend(l_point)
        X.extend(r_point)

        world_point = np.array([*params[3*i: 3*(i+1)], 1])

        l_point_reproj = l_P @ world_point
        l_point_reproj /= l_point_reproj[2] + eps

        r_point_reproj = r_P @ world_point
        r_point_reproj /= r_point_reproj[2] + eps

        F.extend(l_point_reproj[:2])
        F.extend(r_point_reproj[:2])

    X = np.array(X)
    F = np.array(F)
```

```
    return X - F


def _calculate_F_from_P(r_P):
    """
    Function to calculate Fundamental Matrix from the right projection matrix.
    """
    r_epipole = r_P[:, -1]
    s = _get_cross_product_matrix(r_epipole)

    F = s @ r_P[:3, :3]
    F /= F[2, 2] + eps

    return F


def _ssd(l_img_rectified, r_img_rectified, l_interest_points, r_interest_points, M=7, T=60, n_rows=3, col_diff=15):
    """
    Function to find matches between the interest points usign SSD metric.
    """
    m = M // 2 + 1
    l_img_padded = np.pad(l_img_rectified, ((m, m), (m, m)),
                          "constant", constant_values=0)
    r_img_padded = np.pad(r_img_rectified, ((m, m), (m, m)),
                          "constant", constant_values=0)

    l_matches = []

    row_min = np.min(l_interest_points[:, 0])
    row_max = np.max(l_interest_points[:, 0])

    for row in tqdm(range(row_min, row_max+1)):

        l_val_points = l_interest_points[:, 0] == row
        r_val_points = (
            (row - n_rows) <= r_interest_points[:, 0]) & (r_interest_points[:, 0] <= (row + n_rows))
        if (len(l_interest_points[l_val_points]) and len(r_interest_points[r_val_points])):
            # print(l_interest_points[l_val_points], r_interest_points[r_val_points])

            l_points_idx = np.arange(len(l_interest_points))[l_val_points]
            r_points_idx = np.arange(len(r_interest_points))[r_val_points]

            for l_idx in l_points_idx:
                y1, x1 = l_interest_points[l_idx]
                nbd1 = l_img_padded[y1: y1+2*m, x1: x1+2*m]
                ssds = []
                if np.all(nbd1):
                    for r_idx in r_points_idx:
                        y2, x2 = r_interest_points[r_idx]
                        nbd2 = r_img_padded[y2: y2+2*m, x2: x2+2*m]
                        if np.all(nbd2):
                            try:
                                ssd = np.mean((nbd1 - nbd2)**2)
                            except:
                                continue

                            ssds.append(ssd)
                        else:
                            ssds.append(np.inf)
                            continue
                else:
                    ssds.append(np.inf)

                if len(ssds) > 0:
                    min_ssd_idx = np.argmin(ssds)
                    if ssds[min_ssd_idx] < T:
                        l_matches.append(
                            (l_idx, r_points_idx[min_ssd_idx], ssds[min_ssd_idx]))
```

```
    r_matches = []
    row_min = np.min(r_interest_points[:, 0])
    row_max = np.max(r_interest_points[:, 0])

    for row in tqdm(range(row_min, row_max+1)):

        r_val_points = r_interest_points[:, 0] == row
        l_val_points = (
            (row - n_rows) <= l_interest_points[:, 0]) & (l_interest_points[:, 0] <= (row + n_rows
    ))
        if (len(l_interest_points[l_val_points]) and len(r_interest_points[r_val_points])):

            l_points_idx = np.arange(len(l_interest_points))[l_val_points]
            r_points_idx = np.arange(len(r_interest_points))[r_val_points]

            for r_idx in r_points_idx:
                y2, x2 = r_interest_points[r_idx]
                nbd2 = r_img_padded[y2: y2+2*m, x2: x2+2*m]
                ssds = []
                if np.all(nbd2):
                    for l_idx in l_points_idx:
                        y1, x1 = l_interest_points[l_idx]
                        nbd1 = l_img_padded[y1: y1+2*m, x1: x1+2*m]
                        if np.all(nbd1):
                            try:
                                ssd = np.mean((nbd1 - nbd2)**2)
                            except:
                                continue
                            ssds.append(ssd)
                        else:
                            ssds.append(np.inf)
                            continue
                else:
                    ssds.append(np.inf)

                if len(ssds) > 0:
                    min_ssd_idx = np.argmin(ssds)
                    if ssds[min_ssd_idx] < T:
                        r_matches.append(
                            (l_points_idx[min_ssd_idx], r_idx, ssds[min_ssd_idx]))

    matches = list(set(l_matches).intersection(r_matches))
    sorted_matches = sorted(matches, key=lambda x: l_interest_points[x[0]][0])
    filtered_matches = []

    curr_row = l_interest_points[matches[0][0]][0]
    curr_col = l_interest_points[matches[0][0]][1]

    for match in sorted_matches:

        l_idx = match[0]
        r_idx = match[1]

        if l_interest_points[l_idx][0] == curr_row:
            if l_interest_points[l_idx][1] >= curr_col:
                if ((l_interest_points[l_idx][1] - r_interest_points[r_idx][1])**2) < col_diff**2:
                    curr_col = l_interest_points[l_idx][1]
                    filtered_matches.append(match)

        else:
            curr_row = l_interest_points[l_idx][0]

            if ((l_interest_points[l_idx][1] - r_interest_points[r_idx][1])**2) < col_diff**2:
                curr_col = l_interest_points[l_idx][1]
                filtered_matches.append(match)

    sorted_matches = sorted(filtered_matches, key=lambda x: x[2])

    return sorted_matches
```

```python

def _calculate_right_H(r_epipole, h, w):
    """
    Calculate the right rectification homography matrix H.
    """
    T1 = np.array([[1, 0, -w / 2],
                   [0, 1, -h / 2],
                   [0, 0, 1]])

    angle = np.arctan2(-(r_epipole[1] - h/2), (r_epipole[0] - w/2))

    R = np.array([[np.cos(angle), -np.sin(angle), 0],
                  [np.sin(angle), np.cos(angle), 0],
                  [0, 0, 1]])

    f = np.abs((r_epipole[0] - w/2) * np.cos(angle) -
               (r_epipole[1] - h/2) * np.sin(angle))

    G = np.array([[1, 0, 0],
                  [0, 1, 0],
                  [-1/f, 0, 1]])

    H = np.linalg.inv(T1) @ G @ R @ T1

    H /= H[2, 2] + eps

    return H


def _calculate_left_H(l_P, r_P, r_H, l_points, r_points):
    """
    Calculate the left rectification homography matrix H.
    """
    left_P_pinv = np.linalg.pinv(l_P)

    H0 = r_H @ r_P @ left_P_pinv

    l_points_hc = np.hstack((l_points, np.ones((len(l_points), 1)))).T
    r_points_hc = np.hstack((r_points, np.ones((len(r_points), 1)))).T

    x1 = H0 @ l_points_hc
    x1 /= x1[2] + eps

    x2 = r_H @ r_points_hc
    x2 /= x2[2] + eps

    A = x1.T
    b = x2[0]

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

    Ha = np.array([[*abc],
                   [0, 1, 0],
                   [0, 0, 1]])

    H = Ha @ H0

    H /= H[2, 2] + eps

    return H


class StereoVision:
    """Class for Stereo Vision.
    """

    def __init__(self, l_img_fn: str, r_img_fn: str, matching: str = "manual", optimizer: str = "
    scipy_lm"):
        """Initialize the StereoVision object.

        Args:
```

```
            l_img_fn (str): Path to the left image file.
            r_img_fn (str): Path to the right image file.
            matching (str): Matching method. If matching is ``load'', it will load saved points. If no file found, will run in ``manual'' mode.
            optimizer (str): Optimizer method. If optimizer is ``scipy_lm``, it will use scipy's least squares method. If optimizer is ``lm``, it will my implementation
                             LM.
        """
        if matching not in ["manual", "load"]:
            raise ValueError(
                "Invalid matching method. Choose from ['manual', 'load'].")
        if optimizer not in ["scipy_lm", "lm"]:
            raise ValueError(
                "Invalid optimizer. Choose from ['scipy_lm', 'lm'].")

        self.l_img_fn = l_img_fn
        self.r_img_fn = r_img_fn
        if matching == "manual":
            self.matching_method = matching
            self._match_manual(l_img_fn, r_img_fn)
            assert len(self.l_points) == len(self.r_points)

        elif matching == "load":
            self.matching_method = matching
            self._match_load(l_img_fn, r_img_fn)
            assert len(self.l_points) == len(self.r_points)

        self.l_norm_points, self.l_T = _normalize_coords(self.l_points)
        self.r_norm_points, self.r_T = _normalize_coords(self.r_points)
        self.F = _calculate_F_from_points(
            self.l_norm_points, self.r_norm_points, self.l_T, self.r_T)
        self.l_epipole, self.r_epipole = _calculate_epipoles(self.F)
        self.l_P, self.r_P = _calculate_projection_matrices(
            self.F, self.r_epipole)

        params = []
        params.extend(self.r_P.ravel())

        for l_point, r_point in zip(self.l_points, self.r_points):
            world_point = _triangulate(self.l_P, self.r_P, l_point, r_point)
            params.extend(world_point[:3].ravel())

        if optimizer == "scipy_lm":
            solution = least_squares(_cost_func_r_P, params, method='lm', verbose=2, args=[
                                     self.l_P, self.l_points, self.r_points])

        elif optimizer == "lm":
            solution = LM_optim(_cost_func_r_P, params, verbose=1, args=[
                                self.l_P, self.l_points, self.r_points])

        self.r_P = np.reshape(solution.x[:12], (3, 4))

        self.F = _calculate_F_from_P(self.r_P)
        self.l_epipole, self.r_epipole = _calculate_epipoles(self.F)

        self.rectified = False
        self.int_detected = False

        return

    def rectify(self, rot: bool = True):
        """Rectify the stereo images.

        Args:
            rot (bool): If True, rotate the rectified images by 180.
        """
        self.rectified = True
        h, w = self.l_img.shape[:2]
        self.r_H = _calculate_right_H(self.r_epipole, h, w)
        self.l_H = _calculate_left_H(
            self.l_P, self.r_P, self.r_H, self.l_points, self.r_points)
```

```
485
486         if rot:
487             H_rot = np.array([[-1, 0, 0],
488                               [0, -1, 0],
489                               [0, 0, 1]])
490
491             H_tr = np.array([[1, 0, -w/2],
492                              [0, 1, -h/2],
493                              [0, 0, 1]])
494
495             H_tr_inv = np.array([[1, 0, w/2],
496                                  [0, 1, h/2],
497                                  [0, 0, 1]])
498
499             self.l_H = H_tr_inv @ H_rot @ H_tr @ self.l_H
500             self.r_H = H_tr_inv @ H_rot @ H_tr @ self.r_H
501
502         temp = np.linalg.inv(self.r_H.T) @ self.F @ np.linalg.inv(self.l_H)
503         # Should look like
504         # [[0, 0, 0]]
505         # [[0, 0, -1]]
506         # [[0, 1, 0]]
507         print(temp/temp[2, 1])
508
509         self.l_img_rectified = cv.warpPerspective(
510             self.l_img, self.l_H, (int(1.2*w), int(1.2*h)))
511         self.r_img_rectified = cv.warpPerspective(
512             self.r_img, self.r_H, (int(1.2*w), int(1.2*h)))
513
514         left_points_hc = np.hstack(
515             (self.l_points, np.ones((len(self.l_points), 1)))).T
516         right_points_hc = np.hstack(
517             (self.r_points, np.ones((len(self.r_points), 1)))).T
518
519         self.l_points_rectified = self.l_H @ left_points_hc
520         self.r_points_rectified = self.r_H @ right_points_hc
521
522         self.l_points_rectified /= self.l_points_rectified[2]
523         self.r_points_rectified /= self.r_points_rectified[2]
524
525         self.l_points_rectified = self.l_points_rectified[:2].astype(int)
526         self.r_points_rectified = self.r_points_rectified[:2].astype(int)
527
528         return
529
530     def detect_interest_points(self, matching: str = "manual", metric: str = "ssd"):
531         """
532         Detect interest points in the rectified images.
533
534         Args:
535             matching (str): Method for matching interest points. Supported methods are 'manual'
    and 'load'.
536             metric (str): Metric for matching interest points. Supported metrics are 'ssd'.
537         """
538         if metric not in ["ssd"]:
539             raise ValueError("Invalid metric. Supported metric is 'ssd'.")
540         if matching not in ["manual", "load"]:
541             raise ValueError(
542                 "Invalid matching method. Supported methods are 'manual' and 'load'.")
543         if not self.rectified:
544             print("Perform rectification first.")
545             return
546
547         l_gray = cv.cvtColor(self.l_img_rectified, cv.COLOR_BGR2GRAY)
548         r_gray = cv.cvtColor(self.r_img_rectified, cv.COLOR_BGR2GRAY)
549
550         p1 = np.vstack(((self.l_points_rectified + 20).T,
551                         (self.l_points_rectified - 20).T))
552         p2 = np.vstack(((self.r_points_rectified + 20).T,
553                         (self.r_points_rectified - 20).T))
554
```

```
        l_hull = cv.convexHull(p1)
        r_hull = cv.convexHull(p2)

        l_mask = cv.fillConvexPoly(np.zeros(l_gray.shape), l_hull, 255)
        r_mask = cv.fillConvexPoly(np.zeros(r_gray.shape), r_hull, 255)

        l_canny_img = 255 * \
            feature.canny(l_gray, 1, 20, 100, l_mask).astype(np.uint8)
        r_canny_img = 255 * \
            feature.canny(r_gray, 1, 20, 100, r_mask).astype(np.uint8)

        # cv.imshow("l_canny", l_canny_img)
        # cv.imshow("r_canny", r_canny_img)
        # cv.waitKey()
        # cv.destroyAllWindows()

        l_interest_points = np.array(np.where(l_canny_img > 0)).T
        r_interest_points = np.array(np.where(r_canny_img > 0)).T

        if matching == "manual":
            if metric == "ssd":
                matches = _ssd(l_gray, r_gray, l_interest_points,
                               r_interest_points)

            l_match_points = []
            r_match_points = []

            for i in range(len(self.l_points_rectified.T)):
                l_match_points.append([*self.l_points_rectified[:, i], 1])
                r_match_points.append([*self.r_points_rectified[:, i], 1])

            # Choose first 300 matches
            for match in matches[:300]:
                l_match_points.append([*l_interest_points[match[0]][::-1], 1])
                r_match_points.append([*r_interest_points[match[1]][::-1], 1])

            l_match_points = np.array(l_match_points).T
            r_match_points = np.array(r_match_points).T

            l_int_points = np.linalg.inv(self.l_H) @ l_match_points
            l_int_points /= l_int_points[2] + eps

            r_int_points = np.linalg.inv(self.r_H) @ r_match_points
            r_int_points /= r_int_points[2] + eps

            self.l_interest_points = (l_int_points[:2, :]).T
            self.r_interest_points = (r_int_points[:2, :]).T

            print(f"Number of matches found: {len(self.l_interest_points)}")

            dire = self.l_img_fn[7:12]
            np.save(os.path.join("./int_points", dire, "left"),
                    self.l_interest_points)
            np.save(os.path.join("./int_points", dire,
                    "right"), self.r_interest_points)

        elif matching == "load":
            dire = self.l_img_fn[7:12]

            try:
                self.l_interest_points = np.load(
                    os.path.join("./int_points", dire, "left.npy"))
                self.r_interest_points = np.load(
                    os.path.join("./int_points", dire, "right.npy"))

            except FileNotFoundError:
                print(
                    f"Points not found. Running with matching = 'manual' to generate and save them
    .")
                return

```

```
            print(f"Number of matches found: {len(self.l_interest_points)}")
        self.int_detected = True

        return

    def refine(self, optimizer: str = "scipy_lm"):
        """
        Refine the fundamental matrix, projection matrices using matched interest points.
        """

        if optimizer not in ["scipy_lm", "lm"]:
            raise ValueError(
                "Invalid optimizer. Supported optimizers are 'scipy_lm' and 'lm'.")

        if not self.int_detected:
            print(
                "Interest points not detected. Please run detect_interest_points first.")
            return

        self.l_norm_int_points, self.l_T = _normalize_coords(
            self.l_interest_points)
        self.r_norm_int_points, self.r_T = _normalize_coords(
            self.r_interest_points)
        self.F_refined = _calculate_F_from_points(
            self.l_norm_int_points, self.r_norm_int_points, self.l_T, self.r_T)

        self.l_epipole, self.r_epipole = _calculate_epipoles(self.F_refined)
        self.l_P, self.r_P = _calculate_projection_matrices(
            self.F_refined, self.r_epipole)

        params = []

        for l_point, r_point in zip(self.l_interest_points, self.r_interest_points):
            world_point = _triangulate(self.l_P, self.r_P, l_point, r_point)
            params.extend(world_point[:3].ravel())

        if optimizer == "scipy_lm":
            solution = least_squares(_cost_func_world_points, params, method='lm', verbose=2, args
=[
                                     self.l_P, self.r_P, self.l_interest_points, self.
r_interest_points])

        elif optimizer == "lm":
            solution = LM_optim(_cost_func_world_points, params, verbose=2, args=[
                                self.l_P, self.r_P, self.l_interest_points, self.r_interest_points
])

        self.world_points = np.reshape(solution.x, (-1, 3))

        self.F = _calculate_F_from_P(self.r_P)
        self.l_epipole, self.r_epipole = _calculate_epipoles(self.F)

        return

    def _match_manual(self, l_img_fn, r_img_fn):
        """
        Function to select the correspondences manually and save them.
        """
        self.l_img = cv.imread(l_img_fn)
        self.r_img = cv.imread(r_img_fn)

        dire = l_img_fn[7:12]

        self.l_points = select_points(self.l_img)
        np.save(os.path.join("./points", dire, "left"), self.l_points)

        self.r_points = select_points(self.r_img)
        np.save(os.path.join("./points", dire, "right"), self.r_points)

        return

```

```
693     def _match_load(self, l_img_fn, r_img_fn):
694         """
695         Function to load the correspondences from saved file.
696         """
697         self.l_img = cv.imread(l_img_fn)
698         self.r_img = cv.imread(r_img_fn)
699 
700         dire = l_img_fn[7:12]
701 
702         try:
703             self.l_points = np.load(os.path.join("./points", dire, "left.npy"))
704             self.r_points = np.load(os.path.join(
705                 "./points", dire, "right.npy"))
706 
707         except FileNotFoundError:
708             print(
709                 f"Points not found. Running with mode = 'manual' to select points and save them.")
710             self._match_manual(l_img_fn, r_img_fn)
711 
712         return
```

6.1.2 dense_matching.py

```
import numpy as np
import cv2 as cv
from tqdm import tqdm


def census_transform(l_img_fn, r_img_fn, l_disp_fn, window_size=(11, 31), delta=2):
    """
    Function to use census transform to calculate disparity map.
    """
    l_img = cv.cvtColor(cv.imread(l_img_fn), cv.COLOR_BGR2GRAY)
    r_img = cv.cvtColor(cv.imread(r_img_fn), cv.COLOR_BGR2GRAY)
    l_disp = (cv.imread(l_disp_fn, cv.IMREAD_GRAYSCALE).astype(
        np.float32) / 4).astype(np.uint8)

    d_max = np.max(l_disp)
    h, w = l_img.shape

    # m = block height
    # n = block width

    m, n = window_size
    b_size_row = d_max + m//2
    b_size_col = d_max + n//2
    l_img_padded = np.pad(l_img, ((b_size_row, b_size_row),
                          (b_size_col, b_size_col)), mode='edge')
    r_img_padded = np.pad(r_img, ((b_size_row, b_size_row),
                          (b_size_col, b_size_col)), mode='edge')

    l_disp_est = np.zeros(l_img.shape)

    for y in tqdm(range(m//2, h-m//2)):

        for x in range(n//2, w-n//2):

            cost = np.array([np.inf] * (d_max))
            p = l_img_padded[y+d_max:y+d_max+m, x+d_max: x+d_max+n]
            p_win = (p > l_img[y, x])

            for d in range(min(x, d_max)):

                q = r_img_padded[y+d_max:y+d_max+m, (x-d)+d_max: (x-d)+d_max+n]
                q_win = (q > r_img[y, x-d])

                c = np.sum(np.logical_xor(p_win, q_win))
                cost[d] = c

            l_disp_est[y, x] = np.argmin(cost)

    ch = (np.abs(l_disp_est - l_disp) <= delta) & (l_disp > 0)

    return l_disp, l_disp_est, ch
```

6.1.3 plotting.py

```
import cv2 as cv
import numpy as np

import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec
from matplotlib.patches import ConnectionPatch
from mpl_toolkits.mplot3d import proj3d

from stereo_cameras import StereoVision

np.random.seed(4)

# Task 1
def show_correspondences(sv: StereoVision, save_fn: str = None):
    """
    Plotting function to display manually selected correspondences.
    """
    n_pts = len(sv.l_points)
    left_img = sv.l_img
    right_img = sv.r_img
    h1, w1 = left_img.shape[:2]
    h2, w2 = right_img.shape[:2]
    canvas = np.zeros((max(h1, h2), w1 + w2, 3), dtype=np.uint8)
    canvas[:h1, :w1] = left_img
    canvas[:h2, w1:] = right_img
    canvas = cv.cvtColor(canvas, cv.COLOR_BGR2RGB)
    plt.figure(figsize=(9, 9))
    plt.imshow(canvas)
    for i in range(n_pts):
        color = np.random.random(3)
        l_point_x = sv.l_points[i][0]
        l_point_y = sv.l_points[i][1]
        r_point_x = sv.r_points[i][0] + w1
        r_point_y = sv.r_points[i][1]
        plt.plot([l_point_x, r_point_x], [l_point_y, r_point_y], color=color,
                 ms=3, linewidth=1, linestyle='--', marker='.', markerfacecolor=color)

    plt.axis('off')
    plt.title("Manually selected points")
    plt.tight_layout()
    if save_fn is not None:
        plt.savefig(save_fn, bbox_inches='tight', pad_inches=0.1, dpi=600)

    plt.show()
    plt.close()

    return


def show_rectification(sv: StereoVision, save_fn: str = None):
    """
    Plotting function to display rectified images.
    """
    n_pts = len(sv.l_points)
    left_img = sv.l_img_rectified
    right_img = sv.r_img_rectified
    h1, w1 = left_img.shape[:2]
    h2, w2 = right_img.shape[:2]
    canvas = np.zeros((max(h1, h2), w1 + w2, 3), dtype=np.uint8)
    canvas[:h1, :w1] = left_img
    canvas[:h2, w1:] = right_img
    canvas = cv.cvtColor(canvas, cv.COLOR_BGR2RGB)
    plt.figure(figsize=(9, 9))
    plt.imshow(canvas)
    for i in range(n_pts):
        color = np.random.random(3)
        l_point_x = sv.l_points_rectified[0, i]
        l_point_y = sv.l_points_rectified[1, i]
        r_point_x = sv.r_points_rectified[0, i] + w1
```

```
        r_point_y = sv.r_points_rectified[1, i]
        plt.plot([l_point_x, r_point_x], [l_point_y, r_point_y], color=color,
                 ms=3, linewidth=1, linestyle='--', marker='.', markerfacecolor=color)

    plt.axis('off')
    plt.title("Rectified images")
    plt.tight_layout()
    if save_fn is not None:
        plt.savefig(save_fn, bbox_inches='tight', pad_inches=0.1, dpi=600)

    plt.show()
    plt.close()

    return


def show_interest_points(sv: StereoVision, save_fn: str = None):
    """
    Plotting function to display detected interest points.
    """
    left_img = sv.l_img
    right_img = sv.r_img
    h1, w1 = left_img.shape[:2]
    h2, w2 = right_img.shape[:2]
    canvas = np.zeros((max(h1, h2), w1 + w2, 3), dtype=np.uint8)
    canvas[:h1, :w1] = left_img
    canvas[:h2, w1:] = right_img
    canvas = cv.cvtColor(canvas, cv.COLOR_BGR2RGB)
    plt.figure(figsize=(9, 9))
    plt.imshow(canvas)

    for i in range(len(sv.l_interest_points)):
        color = np.random.random(3)
        x1, y1 = sv.l_interest_points[i]
        x2, y2 = sv.r_interest_points[i]

        x2 += w1

        plt.plot([x1, x2], [y1, y2], color=color, ms=3, linewidth=1,
                 linestyle='--', marker='.', markerfacecolor=color)

    plt.axis('off')
    plt.title("Detected Interest Points")
    plt.tight_layout()

    if save_fn is not None:
        plt.savefig(save_fn, bbox_inches='tight', pad_inches=0.1, dpi=600)

    plt.show()
    plt.close()

    return


def show_reconstruction(sv: StereoVision, lines_to_show: list, save_fn: str = None):
    """
    Plotting function to display 3D stereo reconstruction.
    """
    fig = plt.figure(figsize=(12, 8))
    fig.suptitle("Stereo Reconstruction")
    gs = GridSpec(2, 2, fig)
    ax1 = fig.add_subplot(gs[0, 0])
    ax2 = fig.add_subplot(gs[0, 1])
    ax3 = fig.add_subplot(gs[1, :], projection='3d')

    ax3.set_xlabel('X')
    ax3.set_ylabel('Y')
    ax3.set_zlabel('Z')

    ax3.view_init(elev=45, azim=-135, roll=120, vertical_axis='y')
    # ax3.view_init(elev=-45, azim=0, roll=0, vertical_axis='x')
```

```
    ax3.dist = 10

    ax1.axis('off')
    ax2.axis('off')

    def _norm_world_points(world_points):
        mu = np.mean(world_points, axis=0)
        std = np.std(world_points, axis=0)

        norm_world_points = (world_points - mu) / std
        return norm_world_points

    norm_w_p = _norm_world_points(sv.world_points)

    ax1.imshow(cv.cvtColor(sv.l_img, cv.COLOR_BGR2GRAY), cmap='gray')
    ax2.imshow(cv.cvtColor(sv.r_img, cv.COLOR_BGR2GRAY), cmap='gray')

    for start_idx, end_idx in lines_to_show:

        c = np.random.random(3)
        c_r = [1, 0, 0]

        ax1.plot(sv.l_interest_points[[start_idx, end_idx], 0], sv.l_interest_points[[start_idx,
    end_idx], 1],
                 color=c, ms=5, linewidth=2, linestyle='-', marker='.', markerfacecolor=c_r)

        ax2.plot(sv.r_interest_points[[start_idx, end_idx], 0], sv.r_interest_points[[start_idx,
    end_idx], 1],
                 color=c, ms=5, linewidth=2, linestyle='-', marker='.', markerfacecolor=c_r)

        ax3.plot(norm_w_p[[start_idx, end_idx], 0],
                 norm_w_p[[start_idx, end_idx], 1],
                 norm_w_p[[start_idx, end_idx], 2],
                 color=c, ms=5, linewidth=2, linestyle='-', marker='.', markerfacecolor=c_r)

    ax1.scatter(sv.l_interest_points[:, 0], sv.l_interest_points[:, 1], s=3, color=[
                0, 0, 1], marker='.')
    ax2.scatter(sv.r_interest_points[:, 0], sv.r_interest_points[:, 1], s=3, color=[
                0, 0, 1], marker='.')

    ax3.scatter(norm_w_p[:, 0], norm_w_p[:, 1],
                norm_w_p[:, 2], s=2, color=[0, 0, 1], marker='.')

    for i in range(7):

        c = np.random.random(3)
        x_sc_1, y_sc_1, _ = proj3d.proj_transform(*norm_w_p[i], ax3.get_proj())

        con1 = ConnectionPatch((x_sc_1, y_sc_1), (sv.l_interest_points[i, 0], sv.l_interest_points
    [i, 1]),
                               coordsA=ax3.transData, coordsB=ax1.transData, axesA=ax3, axesB=ax1,
     ls=':', color=c)
        fig.add_artist(con1)

        con2 = ConnectionPatch((x_sc_1, y_sc_1), (sv.r_interest_points[i, 0], sv.r_interest_points
    [i, 1]),
                               coordsA=ax3.transData, coordsB=ax2.transData, axesA=ax3, axesB=ax2,
     ls=':', color=c)
        fig.add_artist(con2)

        con3 = ConnectionPatch((sv.l_interest_points[i, 0], sv.l_interest_points[i, 1]),
                               (sv.r_interest_points[i, 0],
                                sv.r_interest_points[i, 1]),
                               coordsA=ax1.transData, coordsB=ax2.transData, axesA=ax1, axesB=ax2,
     ls=':', color=c)
        fig.add_artist(con3)

    plt.tight_layout()

    if save_fn is not None:
        plt.savefig(save_fn, bbox_inches='tight', pad_inches=0.1, dpi=600)
```

```
205
206     plt.show()
207     plt.close()
208
209     return
210
211 # Task 3
212
213
214 def show_disp_maps(l_disp: np.ndarray, l_disp_est: np.ndarray, ch: np.ndarray, window_size: tuple,
        delta: int, accuracy: float, save_fn: str = None):
215     """
216     Visualize the disparity map and estimated disparity map, along with the challenge regions.
217     """
218     fig = plt.figure(figsize=(9, 9))
219     gs = GridSpec(2, 2, fig)
220     ax1 = fig.add_subplot(gs[0, :])
221     ax2 = fig.add_subplot(gs[1, 0])
222     ax3 = fig.add_subplot(gs[1, 1])
223
224     fig.suptitle(f"Census Transform - {window_size} window")
225
226     im1 = ax1.imshow(l_disp, cmap="GnBu")
227     ax1.set_title("Ground truth disparity")
228     ax1.axis('off')
229     plt.colorbar(im1, ax=ax1, shrink=0.5, pad=0.1)
230
231     im2 = ax2.imshow((l_disp_est).astype(np.uint8), cmap="GnBu")
232     ax2.set_title("Estimated disparity")
233     ax2.axis('off')
234     plt.colorbar(im2, ax=ax2, shrink=0.5, pad=0.1)
235
236     im3 = ax3.imshow((255 * ch.astype(np.uint8)), cmap="gray")
237     ax3.set_title(f"Challenge regions, delta={delta}\nAccuracy={accuracy:.4f}")
238     ax3.axis('off')
239     plt.colorbar(im3, ax=ax3, shrink=0.5, pad=0.1)
240
241     if save_fn is not None:
242         plt.savefig(save_fn, bbox_inches='tight', pad_inches=0.2, dpi=600)
243
244     plt.show()
245     plt.close()
246
247     return
```

6.1.4 main.py

```python
import os
import numpy as np
from stereo_cameras import StereoVision
from plotting import show_correspondences, show_rectification, show_interest_points,
    show_reconstruction, show_disp_maps

from dense_matching import census_transform

if __name__ == '__main__':
    run_stereo = True
    run_dense_matching = False

    if run_stereo:
        data_dir = "./data"
        out_dir = "./outs"
        pairs = [f"pair{i}" for i in range(4, 5)]
        # rot = True
        rot = False

        for pair in pairs:
            left_img_filename = os.path.join(data_dir, pair, "left.jpg")
            right_img_filename = os.path.join(data_dir, pair, "right.jpg")

            sv = StereoVision(left_img_filename,
                              right_img_filename, matching="load")

            save_fn = os.path.join(out_dir, pair, "manual_corr.pdf")
            # save_fn = None
            show_correspondences(sv, save_fn)

            sv.rectify(rot=rot)

            save_fn = os.path.join(out_dir, pair, "rectified.pdf")
            show_rectification(sv, save_fn)

            sv.detect_interest_points(matching="load")

            save_fn = os.path.join(out_dir, pair, "interest_points.pdf")
            show_interest_points(sv, save_fn)

            sv.refine()

            lines_to_show = [(0, 1), (1, 2), (2, 3), (0, 3),
                             (4, 5), (5, 6), (6, 2), (4, 0),
                             (3, 5), (8, 9), (9, 10), (10, 7),
                             (8, 7), (12, 13), (13, 11), (11, 12),
                             (14, 15), (16, 15), (17, 16), (17, 14),
                             (8, 3), (8, 5), (7, 0), (7, 4),
                             (18, 3), (18, 5), (18, 2), (18, 6)]

            save_fn = os.path.join(out_dir, pair, "reconstruction1.pdf")
            show_reconstruction(sv, lines_to_show, save_fn)

    if run_dense_matching:
        data_dir = "./Task3Images"
        out_dir = "./outs"

        img_fnames = [os.path.join(data_dir, im)
                      for im in ["im2.png", "im6.png"]]
        disp_fnames = [os.path.join(data_dir, disp) for disp in ["disp2.png"]]
        # windows = [(11, 11), (11, 21), (21, 11), (21, 21), (31, 31)]
        windows = [(5, 5)]
        deltas = [1, 2]
        for w in windows:
            for d in deltas:
                print(f"Window size: {w}, Delta: {d}")

                l_disp, l_disp_est, ch = census_transform(
                    *img_fnames, *disp_fnames, w, d, save_fn)
```

```
                accuracy = np.count_nonzero(ch) / np.count_nonzero(l_disp)
                print(f"Accuracy: {accuracy:.4f}")

                save_fn = os.path.join(
                    out_dir, "task3", f"dense_{w[0]}_{w[1]}_{d}.pdf")
                show_disp_maps(l_disp, l_disp_est, ch, w, d, accuracy, save_fn)
```

Usage:

```
python ./src/main.py
```

6.2 Task 4

```python
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 (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.
- 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 from one camera to the other. The resulting transformation matrix is a 4x4 matrix is of the form:
    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(2, 2, figsize=(9, 6), constrained_layout=True)

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

    # Depth 0
    im1 = ax[0, 1].imshow(depth0, cmap='jet')
    ax[0, 1].set_title('Depth 0')
    ax[0, 1].axis('off')
    cbar1 = fig.colorbar(im1, ax=ax[0, 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[1, 0].imshow(img1)
    ax[1, 0].set_title('Image 1')
    ax[1, 0].axis('off')

    # Depth 1
    im2 = ax[1, 1].imshow(depth1, cmap='jet')
    ax[1, 1].set_title('Depth 1')
    ax[1, 1].axis('off')
    cbar2 = fig.colorbar(im2, ax=ax[1, 1], 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=0.1)
    plt.close()


def depth_to_world(depth_map, K, T, img):
    valid_coords = np.column_stack(np.where(depth_map > 0))
    depth_values = depth_map[valid_coords[:, 0],
                             valid_coords[:, 1]].reshape((-1, 1))
    c = (img[valid_coords[:, 0], valid_coords[:, 1]] / 255).reshape((-1, 3))
    K_inv = np.linalg.inv(K)
    pixel_coords = np.hstack(
        (valid_coords[:, ::-1], np.ones((valid_coords.shape[0], 1))))
    cam_coords = (depth_values.T) * (K_inv @ pixel_coords.T)

    cam_coords_hc = np.vstack((cam_coords, np.ones((1, cam_coords.shape[1]))))
    X_world_hc = np.linalg.inv(T) @ cam_coords_hc

```

```
    return X_world_hc[:3, :].T, c


def plot_3D_world_points(X_world_A, X_world_B, c_A, c_B, save_fn=None):
    fig = plt.figure(figsize=(10, 8))
    ax = fig.add_subplot(111, projection='3d')

    ax.scatter(-X_world_A[:, 0], -X_world_A[:, 1],
               X_world_A[:, 2], c=c_A, s=0.1, marker=',')
    ax.scatter(-X_world_B[:, 0], -X_world_B[:, 1],
               X_world_B[:, 2], c=c_B, s=0.1, marker=',')

    ax.set_title('3D World Points')
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
    ax.view_init(elev=20, azim=180, vertical_axis='y')
    ax.dist = 8
    if save_fn is not None:
        plt.savefig(save_fn, bbox_inches='tight', dpi=600, pad_inches=0.4)
    # plt.show()
    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, 11)  # ignore a border of 10 pxls
        y = np.linspace(10, img0.shape[0]-10, 11)  # ignore a border of 10 pxls

        xx, yy = np.meshgrid(x, y)
        # make homogeneous coordinates for points0 #[3, N]

```

```
        points0 = np.vstack((xx.ravel(), yy.ravel()))
        points0 = np.vstack((points0, np.ones(points0.shape[1])))

        # (2) get depth values at points0
        depth_values0 = depth0[(yy).astype(np.int32), (xx).astype(np.int32)]
        # remove points with depth 0 (invalid points)
        valid_points = depth_values0 > 0
        # mask points0 and depth_values0

        valid_depth_values0 = depth_values0[valid_points]
        points0 = points0[:, valid_points.ravel()]

        # (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 = valid_depth_values0 * xyz_cam0
        # make homogeneous coordinates [4,N]
        xyz_cam0_hc = np.vstack((xyz_cam0, np.ones(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[:3, :] @ xyz_world_hc
        # get z coordinates for depth check
        estimated_depth_values1 = xyz_cam1_hc[2]

        # project to image 1
        points1 = K1 @ xyz_cam1_hc  # [3, N]
        # normalize by dividing by last row
        points1 /= points1[2]  # [3, N]
        # check if points1 are within image bounds
        valid_points1 = (points1[0] >= 0) & (points1[0] < w1) & (
            points1[1] >= 0) & (points1[1] < h1)
        # mask points1 and estimated_depth_values1
        points1 = (points1[:2, valid_points1.ravel()]).astype(np.int32)
        # get the depth values at these points using the depth map
        true_depth_values1 = depth1[points1[1, :], points1[0, :]]

        # (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 (true_depth_values1[i] > 0) and (np.abs(estimated_depth_values1[i] -
    true_depth_values1[i]) < DEPTH_THR):
                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=0.1)
        plt.close()

        # (6) Plot all 3D points for the pair
        """
        <Student code>
        ...
        """
        X_world_A, c_A = depth_to_world(depth0, K0, T0, img0)
        X_world_B, c_B = depth_to_world(depth1, K1, T1, img1)
        # save_fn = None
        save_fn = f"./pics/reconstruct_pair_{i_pair}.png"
        plot_3D_world_points(X_world_A, X_world_B, c_A, c_B, save_fn)
        print(f"Done with pair {i_pair}")
```

Usage:

```
python ./plot_depth_check_v1.py
```