After your finish the assignment, remember to run all cells and save the note book to your local machine as a PDF for gradescope submission by pressing Ctrl-P or Cmd-P. Make sure images are not split between pages; insert Text blocks to make sure this is the case before printing to PDF!

List your collaborators here:

16720 HW 4: 3D Reconstruction

Pablo Ortega-Kral (portegak)

Problem 1: Theory

1.1

Show that if the image projection of the 3d point lies at pixel (0, 0) for both images, the f_{33} element of the fundamental matrix must be zero

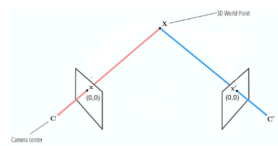


Figure 1: Figure for Q1.1. $\mathbf{C}$ and $\mathbf{C}'$ are the camera centers. $\mathbf{x}$ is the projection of the 3D point $\mathbf{X}$ onto the left image, and $\mathbf{x}'$ is the projection of $\mathbf{X}$ onto the right image, and both are (0,0) in their respective image frames. Thus the camera's principal axes intersect at 3D point $\mathbf{X}$ (which is in the global coordinate frame).

The fundamental matrix is that which enforces the epipolar constraint for a 3D point projected into two camera views, it captures the extrinsic and intrinsic transformations. For this example, this means

$$\mathbf{x}'F\mathbf{x} = 0$$

Giving that F is a product of various 3×3 matrices, we expand it as $F = \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix}$. While $\mathbf{x}'$ and $\mathbf{x}$ are homogenous coordinates in each of the camera's views.

$$\begin{bmatrix} x'_1 & x'_2 & 1 \end{bmatrix} \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix} = 0$$

Given that we known the projected point corresponds to (0,0) we can rewrite the above as

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

We can see that when applying the multiplication we are essentially picking the element in the 3 row and 3 column such that

$$\begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = f_{33} \therefore f_{33} = 0$$

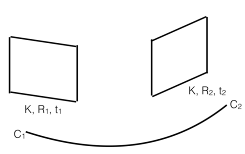


Figure 2: Figure for Q1.2. $C1$ and $C2$ are the camera centers. K is the camera intrinsic matrix. R_1 and t_1 are the rotation and translation of the robot in global coordinate frame at *e.g.* timestep 1, and R_2 and t_2 are the rotation and translation of the robot in global coordinate frame at the next timestep 2. R_{rel} and $\mathbf{t}_{rel}$ are the relative rotation and translation between the two frames.

Q: What is relative rotation (R_{rel}) and relative translation (t_{rel}) between time i and i + 1?

By the problem definition, R_{rel} is the rotation between two consecutive frames such that

$$R_{i+1} = R_{rel}Ri \rightarrow R_{rel} = R_{i+1}Ri^T$$

For translation, we must account for the relative rotation between the consecutive frames

$$t_{rel} = t_{i+1} - R_{rel}t_i$$

$$t_{rel} = t_{i+1} - R_{i+1}Ri^T t_i$$

Q: Suppose the camera intrinsics (K) are known. Express the essential matrix (E) and the fundamental matrix (F) in terms of K, Rrel and trel

The essential matrix is obtained by writting a traditional 3D-to-3D transform with an epipolar constraint such that

$$\begin{bmatrix} X_1' & X_2' & X_3' \end{bmatrix} [t] \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} = 0$$

Where $[t]$ is the matrix represenation of the translation, and $\begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$ is the rotation matrix. Given this all are 3×3 we can combine them into the essential matrix. For the giving example, this would mean.

$$E = [t_{rel}]R_{rel}$$

The fundamental matrix adjusts the essential matrix to be able to relate the projection of a point in a camera rather than the 3D point itself. It does this by incorporating the camera matrix. Assuming that the a point in the camera is giving by $\mathbf{x} = \begin{bmatrix} \mathbf{x_1} \\ \mathbf{x_2} \end{bmatrix}$, we find it's corresponding 3D point X by,

$$X = K^{-1}\mathbf{x}$$

This allows us to rewrite the above expression as

$$\mathbf{x'}^T K^{-1^T}[t_{rel}]R_{rel}K^{-1}\mathbf{x} = 0$$

Therefore, the fundamental matrix is ,

$$F = K^{-1^T}[t_{rel}]R_{rel}K^{-1}$$

Coding

Initialization

Run the following code, which imports the modules you'll need and defines helper functions you may need to use later in your implementations.

In [22]:
```
import os
import numpy as np
import scipy
import scipy.optimize
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import cv2


connections_3d = [[0,1], [1,3], [2,3], [2,0], [4,5], [6,7], [8,9], [9,11], [10,11], [10,8], [0,4], [4,8],
                  [1,5], [5,9], [2,6], [6,10], [3,7], [7,11]]
color_links = [(255,0,0),(255,0,0),(255,0,0),(255,0,0),(0,0,255),(255,0,255),(0,255,0),(0,255,0),(0,255,0)
colors = ['blue','blue','blue','blue','red','magenta','green','green','green','green','red','red','red','r
```

```
def visualize_keypoints(image, pts, Threshold=100):
    '''
    This function visualizes the 2d keypoint pairs in connections_3d
    (as define above) whose match score lies above a given Threshold
    in an OpenCV GUI frame, against an image background.

    :param image: image as a numpy array, of shape (height, width, 3) where 3 is the number of color channe
    :param pts: np.array of shape (num_points, 3)
    '''
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    for i in range(12):
        cx, cy = pts[i][0:2]
        if pts[i][2]>Threshold:
            cv2.circle(image,(int(cx),int(cy)),5,(0,255,255),5)

    for i in range(len(connections_3d)):
        idx0, idx1 = connections_3d[i]
        if pts[idx0][2]>Threshold and pts[idx1][2]>Threshold:
            x0, y0 = pts[idx0][0:2]
            x1, y1 = pts[idx1][0:2]
            cv2.line(image, (int(x0), int(y0)), (int(x1), int(y1)), color_links[i], 2)

    plt.imshow(image)

    return image


def plot_3d_keypoint(pts_3d):
    '''
    this function visualizes 3d keypoints on a matplotlib 3d axes

    :param pts_3d: np.array of shape (num_points, 3)
    '''
    fig = plt.figure()
    num_points = pts_3d.shape[0]
    ax = fig.add_subplot(111, projection='3d')
    for j in range(len(connections_3d)):
        index0, index1 = connections_3d[j]
        xline = [pts_3d[index0,0], pts_3d[index1,0]]
        yline = [pts_3d[index0,1], pts_3d[index1,1]]
        zline = [pts_3d[index0,2], pts_3d[index1,2]]
        ax.plot(xline, yline, zline, color=colors[j])
    np.set_printoptions(threshold=1e6, suppress=True)
    ax.set_xlabel('X Label')
    ax.set_ylabel('Y Label')
    ax.set_zlabel('Z Label')
    plt.show()


def calc_epi_error(pts1_homo, pts2_homo, F):
    '''
    Helper function to calcualte the sum of squared distance between the
    corresponding points and the estimated epipolar lines.

    pts1_homo \dot F.T \dot pts2_homo = 0

    :param pts1_homo: of shape (num_points, 3); in homogeneous coordinates, not normalized.
    :param pts2_homo: same specification as to pts1_homo.
    :param F: Fundamental matrix
    '''

    line1s = pts1_homo.dot(F.T)
    dist1 = np.square(np.divide(np.sum(np.multiply(
        line1s, pts2_homo), axis=1), np.linalg.norm(line1s[:, :2], axis=1)))

    line2s = pts2_homo.dot(F)
    dist2 = np.square(np.divide(np.sum(np.multiply(
        line2s, pts1_homo), axis=1), np.linalg.norm(line2s[:, :2], axis=1)))

    ress = (dist1 + dist2).flatten()
    return ress


def toHomogenous(pts):
    """
```

```
    Adds a stack of ones at the end, to turn a set of points into a set of
    homogeneous points.

    :params pts: in shape (num_points, 2).
    """
    return np.vstack([pts[:,0],pts[:,1],np.ones(pts.shape[0])]).T.copy()


def _epipoles(E):
    """
    gets the epipoles from the Essential Matrix.

    :params E: Essential matrix.
    """
    U, S, V = np.linalg.svd(E)
    e1 = V[-1, :]
    U, S, V = np.linalg.svd(E.T)
    e2 = V[-1, :]
    return e1, e2


def displayEpipolarF(I1, I2, F, points):
    """
    GUI interface you may use to help you verify your calculated fundamental
    matrix F. Select a point I1 in one view, and it should correctly correspond
    to the displayed point in the second view.
    """
    e1, e2 = _epipoles(F)

    sy, sx, _ = I2.shape

    f, [ax1, ax2] = plt.subplots(1, 2, figsize=(12, 9))
    ax1.imshow(I1)
    ax1.set_title('The point you selected:')
    ax2.imshow(I2)
    ax2.set_title('Verify that the corresponding point \n is on the epipolar line in this image')

    plt.sca(ax1)

    colors = ['r','g','b','y','m','k']
    for i, out in enumerate(points):
      x, y = out #[0]

      xc = x
      yc = y
      v = np.array([xc, yc, 1])
      l = F.dot(v)
      s = np.sqrt(l[0]**2+l[1]**2)

      if s==0:
          print('Zero line vector in displayEpipolar')

      l = l/s

      if l[0] != 0:
          ye = sy-1
          ys = 0
          xe = -(l[1] * ye + l[2])/l[0]
          xs = -(l[1] * ys + l[2])/l[0]
      else:
          xe = sx-1
          xs = 0
          ye = -(l[0] * xe + l[2])/l[1]
          ys = -(l[0] * xs + l[2])/l[1]

      # plt.plot(x,y, '*', 'MarkerSize', 6, 'LineWidth', 2);
      ax1.plot(x, y, '*', markersize=6, linewidth=2, color=colors[i%len(colors)])
      ax2.plot([xs, xe], [ys, ye], linewidth=2, color=colors[i%len(colors)])
    plt.draw()


def _singularize(F):
    U, S, V = np.linalg.svd(F)
    S[-1] = 0
    F = U.dot(np.diag(S).dot(V))
```

```
    return F

def _objective_F(f, pts1, pts2):
    F = _singularize(f.reshape([3, 3]))
    num_points = pts1.shape[0]
    hpts1 = np.concatenate([pts1, np.ones([num_points, 1])], axis=1)
    hpts2 = np.concatenate([pts2, np.ones([num_points, 1])], axis=1)
    Fp1 = F.dot(hpts1.T)
    FTp2 = F.T.dot(hpts2.T)

    r = 0
    for fp1, fp2, hp2 in zip(Fp1.T, FTp2.T, hpts2):
        r += (hp2.dot(fp1))**2 * (1/(fp1[0]**2 + fp1[1]**2) + 1/(fp2[0]**2 + fp2[1]**2))
    return r

def refineF(F, pts1, pts2):
    f = scipy.optimize.fmin_powell(
        lambda x: _objective_F(x, pts1, pts2), F.reshape([-1]),
        maxiter=100000,
        maxfun=10000,
        disp=False
    )
    return _singularize(f.reshape([3, 3]))


# Used in 4.2 Epipolar Correspondence
def epipolarMatchGUI(I1, I2, F, points, epipolarCorrespondence):
    e1, e2 = _epipoles(F)

    sy, sx, _ = I2.shape

    f, [ax1, ax2] = plt.subplots(1, 2, figsize=(12, 9))
    ax1.imshow(I1)
    ax1.set_title('The point you selected:')
    ax2.imshow(I2)
    ax2.set_title('Verify that the corresponding point \n is on the epipolar line in this image \nand that

    plt.sca(ax1)

    colors = ['r','g','b','y','m','k']

    for i, out in enumerate(points):
      x, y = out

      xc = int(x)
      yc = int(y)
      v = np.array([xc, yc, 1])
      l = F.dot(v)
      s = np.sqrt(l[0]**2+l[1]**2)

      if s==0:
          print('Zero line vector in displayEpipolar')

      l = l/s

      if l[0] != 0:
          ye = sy-1
          ys = 0
          xe = -(l[1] * ye + l[2])/l[0]
          xs = -(l[1] * ys + l[2])/l[0]
      else:
          xe = sx-1
          xs = 0
          ye = -(l[0] * xe + l[2])/l[1]
          ys = -(l[0] * xs + l[2])/l[1]

      ax1.plot(x, y, '*', markersize=6, linewidth=2, color=colors[i%len(colors)])
      ax2.plot([xs, xe], [ys, ye], linewidth=2, color=colors[i%len(colors)])

      # draw points
      x2, y2 = epipolarCorrespondence(I1, I2, F, xc, yc)
      ax2.plot(x2, y2, 'ro', markersize=8, linewidth=2)
      plt.draw()
```

Set up data

In this section, we will download the test case image views, camera intrinsics, and point correnspondences, which you will use for testing your implementations.

In [23]:
```
if not os.path.exists('data'):
  !wget https://www.andrew.cmu.edu/user/eweng/data.zip -O data.zip
  !unzip -qq "data.zip"
  print("downloaded and unzipped data")
```

Problem 2: Estimating the Fundamental Matrix with the Eight-point Algorithm

In this part, implement the 8-point algorithm you learned in class, which estimates the fundamental matrix from corresponding points in two images.

In [24]:
```
def eightpoint(pts1, pts2, M):
  '''
  Q2.1: Eight Point Algorithm
  Input:  pts1, Nx2 Matrix
          pts2, Nx2 Matrix
          M, a scalar parameter computed as max(imwidth, imheight)
  Output: F, the fundamental matrix

  HINTS:
  (1) Normalize the input pts1 and pts2 using the matrix T.
  (2) Setup the eight point algorithm's equation.
  (3) Solve for the least square solution using SVD.
  (4) Use the function `_singularize` (provided in the helper functions above) to enforce the singularity c
  (5) Use the function `refineF` (provided in the helper functions above) to refine the computed fundamenta
      (Remember to use the normalized points instead of the original points)
  (6) Unscale the fundamental matrix by the lower right corner element
  '''
  def normalize(pts, M):
    # Compute norm matrix
    scale = 2.0/M
    mean = np.mean(pts, axis=0)
    T = np.array([
      [scale, 0, -scale*mean[0]],
      [0, scale, -scale*mean[1]],
      [0, 0, 1]
    ])
    # Make points homogeneous
    pts_norm = np.hstack((pts, np.ones((pts.shape[0], 1))))
    pts_norm = (T @ pts_norm.T).T
    return pts_norm, T

  def denormalize(F, T1, T2):
    return T2.T @ F @ T1

  F = None
  N = pts1.shape[0]

  pts1_norm, T1 = normalize(pts1, M)
  pts2_norm, T2 = normalize(pts2, M)

  # Construct A matrix
  A = np.zeros((N, 9))
  for i in range(N):
    x, y = pts1_norm[i, :2]
    x_prime, y_prime = pts2_norm[i, :2]
    A[i, :] = np.array([x*x_prime, x*y_prime, x, y*x_prime, y*y_prime, y, x_prime, y_prime, 1])

  # Solve using SVD
  _, _, f = np.linalg.svd(A.T @ A)
  F = f[-1, :].reshape(3, 3)

  F = _singularize(F)
```

```
    F = refineF(F, pts1_norm[:, :2], pts2_norm[:, :2])

    # Denormalize
    F = denormalize(F, T1, T2)
    F /= F[2, 2]

    return F
```

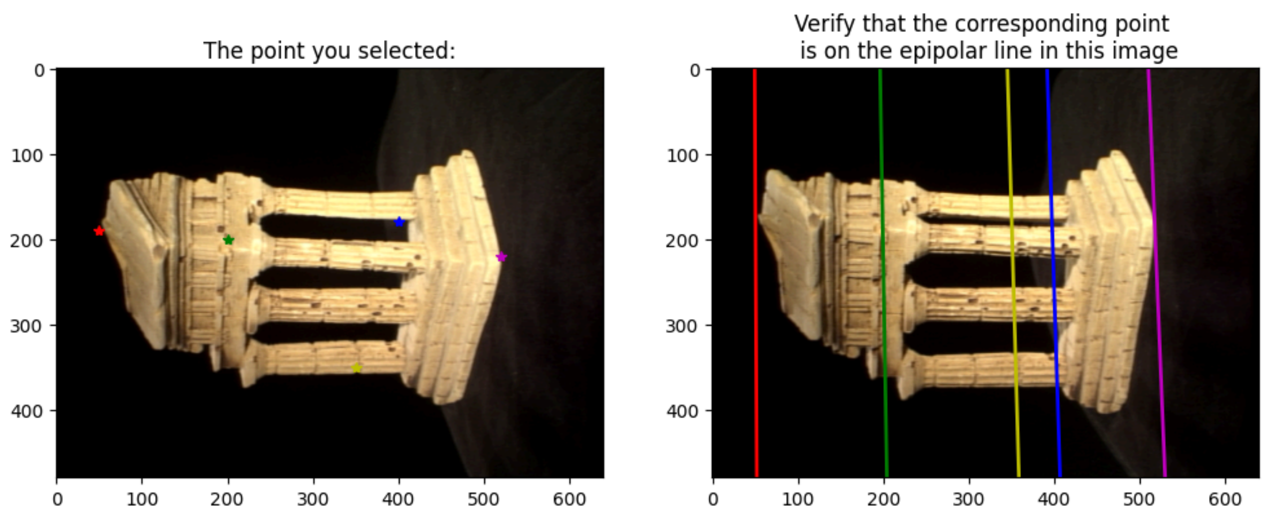
Run this code to test your implementation of the 8-point algorithm. Your code should pass all the assert statements at the end.

In [25]:
```
correspondence = np.load('data/some_corresp.npz') # Loading correspondences
intrinsics = np.load('data/intrinsics.npz') # Loading the intrinscis of the camera
K1, K2 = intrinsics['K1'], intrinsics['K2']
pts1, pts2 = correspondence['pts1'], correspondence['pts2']
im1 = plt.imread('data/im1.png')
im2 = plt.imread('data/im2.png')

F = eightpoint(pts1, pts2, M=np.max([*im1.shape, *im2.shape]))
np.set_printoptions(suppress=True)
print(f'recovered F:\n{F.round(4)}')

# Simple Tests to verify your implementation:
pts1_homogenous, pts2_homogenous = toHomogenous(pts1), toHomogenous(pts2)

assert F.shape == (3, 3), "F is wrong shape"
assert F[2, 2] == 1, "F_33 != 1"
assert np.linalg.matrix_rank(F) == 2, "F should have rank 2"
assert np.mean(calc_epi_error(pts1_homogenous, pts2_homogenous, F)) < 1, "F error is too high to be accura
```

```
recovered F:
[[-0.      0.     -0.2483]
 [ 0.     -0.      0.0004]
 [ 0.2387 -0.0047  1.    ]]
```

The following tool may help you debug. You may specify a point in im1, and view the corresponding epipolar line in im2 based on the F you found. In your submission, make sure you include the debug picture below, with at least five epipolar point-line correspondences taht show that your calculation of F is correct.

In [26]:
```
# the points in im1, whose correnponding epipolar line in im2 you'd like to verify
point = [(50,190),(200, 200), (400,180), (350,350), (520, 220)]
# feel free to change these point, to verify different point correspondences
displayEpipolarF(im1, im2, F, point)
```



Problem 3: Metric Reconstruction

3.1 Essential Matrix

In [27]:
```
def essentialMatrix(F, K1, K2):
    '''
    Q3.1: Compute the essential matrix E.
    Input:  F, fundamental matrix
            K1, internal camera calibration matrix of camera 1
            K2, internal camera calibration matrix of camera 2
    Output: E, the essential matrix
    '''
    E = K2.T @ F @ K1
    E /=E[-1,-1]

    return E
```

Run the following code to check your implementation.

In [28]:
```
correspondence = np.load('data/some_corresp.npz') # Loading correspondences
intrinsics = np.load('data/intrinsics.npz') # Loading the intrinscis of the camera
K1, K2 = intrinsics['K1'], intrinsics['K2']
pts1, pts2 = correspondence['pts1'], correspondence['pts2']
im1 = plt.imread('data/im1.png')
im2 = plt.imread('data/im2.png')

F = eightpoint(pts1, pts2, M=np.max([*im1.shape, *im2.shape]))
E = essentialMatrix(F, K1, K2)
print(f'recovered E:\n{E.round(4)}')

# Simple Tests to verify your implementation:
assert(E[2, 2] == 1)
assert(np.linalg.matrix_rank(E) == 2)
```

```
recovered E:
[[   -1.6901   375.9816 -2539.983 ]
 [  295.7064    -6.3446    62.0918]
 [ 2547.5074    24.3781     1.    ]]
```

3.2 Triangulation

In [29]:
```
def triangulate(C1, pts1, C2, pts2):
    '''
    Q3.2: Triangulate a set of 2D coordinates in the image to a set of 3D points.
    Input:  C1, the 3x4 camera matrix
            pts1, the Nx2 matrix with the 2D image coordinates per row
            C2, the 3x4 camera matrix
            pts2, the Nx2 matrix with the 2D image coordinates per row
    Output: P, the Nx3 matrix with the corresponding 3D points per row
            err, the reprojection error.

    Hints:
    (1) For every input point, form A using the corresponding points from pts1 & pts2 and C1 & C2
    (2) Solve for the least square solution using np.linalg.svd
    (3) Calculate the reprojection error using the calculated 3D points and C1 & C2 (do not forget to conve
    homogeneous coordinates to non-homogeneous ones)
    (4) Keep track of the 3D points and projection error, and continue to next point
    (5) You do not need to follow the exact procedure above.
    '''

    err = []

    N = pts1.shape[0]
    A = []
    P = np.zeros((N, 3))


    for i in range(N):
        cam_pts1 = pts1[i]
        cam_pts2 = pts2[i]
        # Construct projected points
        A = np.stack([
            cam_pts1[1] * C1[2] - C1[1],
            C1[0] - cam_pts1[0] * C1[2],
            cam_pts2[1] * C2[2] - C2[1],
            C2[0] - cam_pts2[0] * C2[2]
```

```
        ])
        assert A.shape == (4, 4) # Sanity check to see if A is constructed correctly

        # Solve using SVD
        _, _, V = np.linalg.svd(A)
        cam_pts3D = V[-1, :3] / V[-1, -1]
        P[i] = cam_pts3D[:3] # Only save non-homogeneous coordinates

        # Deproject to 2D to calculate error
        cam_pts3D_homogeneous = np.append(cam_pts3D, 1)
        cam_pts1_reprojected = C1 @ cam_pts3D_homogeneous
        cam_pts2_reprojected = C2 @ cam_pts3D_homogeneous

        cam_pts1_reprojected /= cam_pts1_reprojected[2] * np.sign(cam_pts1_reprojected[2])
        cam_pts2_reprojected /= cam_pts2_reprojected[2] * np.sign(cam_pts2_reprojected[2])

        cam1_err = np.linalg.norm(cam_pts1 - cam_pts1_reprojected[:2])
        cam2_err = np.linalg.norm(cam_pts2 - cam_pts2_reprojected[:2])
        err.append(cam1_err + cam2_err)

    err = np.array(err)
    err = np.sum(err)

    return P, err
```

3.3 Find M2

In [30]:

```
def camera2(E):
  """helper function to find the 4 possibile M2 matrices"""
  U,S,V = np.linalg.svd(E)
  m = S[:2].mean()
  E = U.dot(np.array([[m,0,0], [0,m,0], [0,0,0]])).dot(V)
  U,S,V = np.linalg.svd(E)
  W = np.array([[0,-1,0], [1,0,0], [0,0,1]])

  if np.linalg.det(U.dot(W).dot(V))<0:
      W = -W

  M2s = np.zeros([3,4,4])
  M2s[:,:,0] = np.concatenate([U.dot(W).dot(V), U[:,2].reshape([-1, 1])/abs(U[:,2]).max()], axis=1)
  M2s[:,:,1] = np.concatenate([U.dot(W).dot(V), -U[:,2].reshape([-1, 1])/abs(U[:,2]).max()], axis=1)
  M2s[:,:,2] = np.concatenate([U.dot(W.T).dot(V), U[:,2].reshape([-1, 1])/abs(U[:,2]).max()], axis=1)
  M2s[:,:,3] = np.concatenate([U.dot(W.T).dot(V), -U[:,2].reshape([-1, 1])/abs(U[:,2]).max()], axis=1)
  return M2s


def findM2(F, pts1, pts2, intrinsics):
    '''
    Q3.3: Function to find camera2's projective matrix given correspondences
        Input:  F, the pre-computed fundamental matrix
                pts1, the Nx2 matrix with the 2D image coordinates per row
                pts2, the Nx2 matrix with the 2D image coordinates per row
                intrinsics, the intrinsics of the cameras, load from the .npz file
                filename, the filename to store results
        Output: [M2, C2, P] the computed M2 (3x4) camera projective matrix, C2 (3x4) K2 * M2, and the 3D po

    ***
    Hints:
    (1) Loop through the 'M2s' and use triangulate to calculate the 3D points and projection error. Keep tr
        of the projection error through best_error and retain the best one.
    (2) Remember to take a look at camera2 to see how to correctly reterive the M2 matrix from 'M2s'.

    '''

    K1, K2 = intrinsics['K1'], intrinsics['K2']
    E = essentialMatrix(F, K1, K2)
    M1 = np.array([
        [1, 0, 0, 0],
        [0, 1, 0, 0],
        [0, 0, 1, 0]
    ])
    M2_candidates = camera2(E)
    M2 = None
```

```
    C2 = None
    best_P = None
    best_error = np.inf

    _, _, N = M2_candidates.shape

    for i in range(N):
        candidate_cam = M2_candidates[:,:,i]
        C2 = K2 @ candidate_cam

        P, error = triangulate((K1 @ M1), pts1, C2, pts2)
        if error < best_error:
            best_error = error
            M2 = candidate_cam
            best_P = P

    return M2, C2, best_P
```

Run the following code to check your implementation of triangulation and findM2.

In [31]:
```
correspondence = np.load('data/some_corresp.npz') # Loading correspondences
intrinsics = np.load('data/intrinsics.npz') # Loading the intrinscis of the camera
K1, K2 = intrinsics['K1'], intrinsics['K2']
pts1, pts2 = correspondence['pts1'], correspondence['pts2']
im1 = plt.imread('data/im1.png')
im2 = plt.imread('data/im2.png')

F = eightpoint(pts1, pts2, M=np.max([*im1.shape, *im2.shape]))

M2, C2, P = findM2(F, pts1, pts2, intrinsics)

# Simple Tests to verify your implementation:
M1 = np.array([
    [1, 0, 0, 0],
    [0, 1, 0, 0],
    [0, 0, 1, 0]
])
C1 = K1.dot(M1)
C2 = K2.dot(M2)
P_test, err = triangulate(C1, pts1, C2, pts2)
print(f'Error: {err}')
assert(err < 500)
```

```
Error: 208.76725901940262
```

Problem 4: 3D Visualization

In [32]:
```
def epipolarCorrespondence(im1, im2, F, x, y, window_size=45):
    '''
    Q4.1: 3D visualization of the temple images.
    Input:  im1, the first image
            im2, the second image
            F, the fundamental matrix
            x1, x-coordinates of a pixel on im1
            y1, y-coordinates of a pixel on im1
    Output: x2, x-coordinates of the pixel on im2
            y2, y-coordinates of the pixel on im2

    Hints:
    (1) Given input [x1, x2], use the fundamental matrix to recover the corresponding epipolar line on imag
    (2) Search along this line to check nearby pixel intensity (you can define a search window) to
        find the best matches
    (3) Use gaussian weighting to weight the pixel simlairty

    '''

    def gaussian(size, sigma=1):
        coords = np.arange(-size, size+1)
        x, y = np.meshgrid(coords, coords)
        dist_sq = (x)**2 + (y)**2
        kernel_2d = np.exp(-dist_sq / (2 * sigma**2))
        kernel_2d /= kernel_2d.sum()
        kernel_3d = np.dstack([kernel_2d] * 3)
```

```
        return kernel_3d

    def get_line_points(line, image, start, points = 100):
        H, W, _ = image.shape
        a,b,c   = line

        y1 = start
        y2 = H - (start + 1)

        x1 = -(b*y1 + c)/a
        x2 = -(b*y2 + c)/a

        x_vals = np.linspace(x1, x2, points)
        y_vals = np.linspace(y1, y2, points)
        pts = np.array([x_vals, y_vals]).T
        return pts

    x1, y1 = int(x), int(y)
    # Find epipolar line
    epipolar_line = F @ np.array([x, y, 1])
    # Read pixel intensity
    template = im1[y1-window_size:y1+window_size+1, x1-window_size:x1+window_size+1]

    # Search along the epipolar line
    best_score = np.inf
    best_x2, best_y2 = None, None
    weights = gaussian(window_size, sigma=window_size)
    line_points = get_line_points(epipolar_line, im2, start=window_size)

    for x2, y2 in line_points:
        x2, y2 = int(x2), int(y2)
        if x2 < window_size or x2 >= im2.shape[1] - window_size or y2 < window_size or y2 >= im2.shape[0]
            continue
        search_window = im2[y2-window_size:y2+window_size+1, x2-window_size:x2+window_size+1]
        score = np.mean(np.abs(template - search_window) * weights)
        if score < best_score:
            best_score = score
            best_x2, best_y2 = x2, y2
    x2, y2 = best_x2, best_y2
    return x2, y2
```

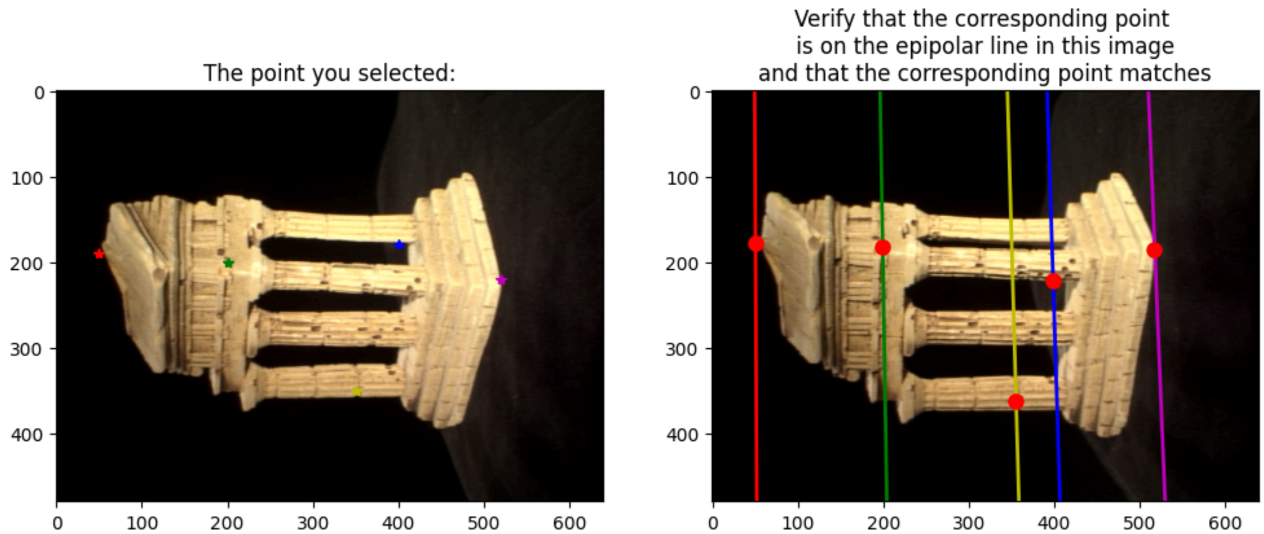
Run the following code to check your implementation.

In [33]:
```
correspondence = np.load('data/some_corresp.npz') # Loading correspondences
intrinsics = np.load('data/intrinsics.npz') # Loading the intrinscis of the camera
K1, K2 = intrinsics['K1'], intrinsics['K2']
pts1, pts2 = correspondence['pts1'], correspondence['pts2']
im1 = plt.imread('data/im1.png')
im2 = plt.imread('data/im2.png')

F = eightpoint(pts1, pts2, M=np.max([*im1.shape, *im2.shape]))

# Simple Tests to verify your implementation:
x2, y2 = epipolarCorrespondence(im1, im2, F, 119, 217, window_size=45)
print(np.linalg.norm(np.array([x2, y2]) - np.array([118, 181])))
assert(np.linalg.norm(np.array([x2, y2]) - np.array([118, 181])) < 10)
```

```
1.0
```

Use the below tool to debug your code.

In [34]:
```
# the points in im1 whose correnponding epipolar line in im2 you'd like to verify
points = [(50,190), (200, 200), (400,180), (350,350), (520, 220)]
# feel free to change these points to verify different point correspondences
epipolarMatchGUI(im1, im2, F, points, epipolarCorrespondence)
```



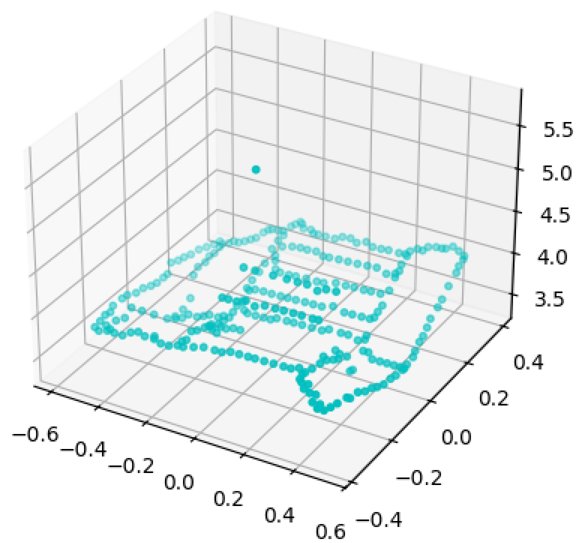
4.2 Temple Visualization

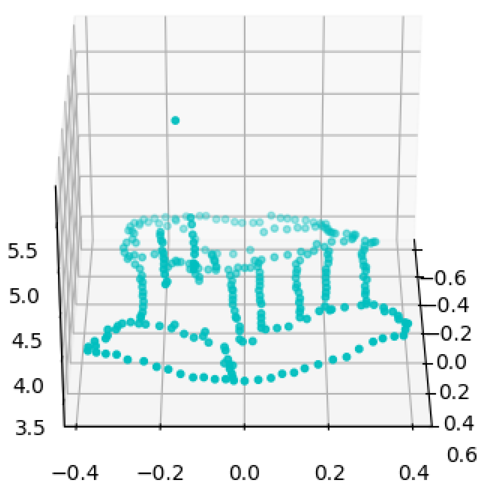
In [35]:
```
def compute3D_pts(temple_pts1, intrinsics, F, im1, im2):
    '''
    Q4.2: Finding the 3D position of given points based on epipolar correspondence and triangulation
    Input:  temple_pts1, chosen points from im1
            intrinsics, the intrinsics dictionary for calling epipolarCorrespondence
            F, the fundamental matrix
            im1, the first image
            im2, the second image
    Output: P (Nx3) the recovered 3D points

    Hints:
    (1) Use epipolarCorrespondence to find the corresponding point for [x1 y1] (find [x2, y2])
    (2) Now you have a set of corresponding points [x1, y1] and [x2, y2], you can compute the M2
        matrix and use triangulate to find the 3D points.
    (3) Use the function findM2 to find the 3D points P (do not recalculate fundamental matrices)
    (4) As a reference, our solution's best error is around ~2200 on the 3D points.
    '''
    points = []
    for x1, y1 in temple_pts1:
        x2, y2 = epipolarCorrespondence(im1, im2, F, x1, y1)
        points.append([x2, y2])
    temple_pts2 = np.array(points)
    _, _, P = findM2(F, temple_pts1, temple_pts2, intrinsics)
    return P
```

Below, integrate everything together. The provided starter code loads in the temple data found at `data/templeCoords.npz`, which contains 288 hand-selected points from im1 saved in the variables x1 and y1. Then, get the 3d points from the 2d point point correspondences by calling the function you just implemented, as well as other necessary function. Finally, visualize the 3D reconstruction using matplotlib or plotly 3d scatter plot.

In [36]:
```
temple_coords = np.load('data/templeCoords.npz') # Loading temple coordinates
correspondence = np.load('data/some_corresp.npz') # Loading correspondences
intrinsics = np.load('data/intrinsics.npz') # Loading the intrinscis of the camera
K1, K2 = intrinsics['K1'], intrinsics['K2']
pts1, pts2 = correspondence['pts1'], correspondence['pts2']
im1 = plt.imread('data/im1.png')
im2 = plt.imread('data/im2.png')

# ----- TODO -----
# Call eightpoint to get the F matrix
F = eightpoint(pts1, pts2, M=np.max([*im1.shape, *im2.shape]))
# Call compute3D_pts to get the 3D points and visualize using matplotlib scatter
# hint: you can change the viewpoint of a matplotlib 3d axes using
# `ax.view_init(azim, elev)` where azim is the rotation around the vertical z
# axis, and elev is the angle of elevation from the x-y plane
temple_pts1 = np.hstack([temple_coords['x1'], temple_coords['y1']])
P = compute3D_pts(temple_pts1, intrinsics, F, im1, im2)
```

```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(P[:, 0], P[:, 1], P[:, 2], s=10, c='c', depthshade=True)
plt.draw()

# also show a different viewpoint
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(P[:, 0], P[:, 1], P[:, 2], s=10, c='c', depthshade=True)
ax.view_init(30, 0)
plt.draw()
```





Problem 5: Bundle Adjustment

Below is the implementation of RANSAC for Fundamental Matrix Recovery.

In [37]:
```
def ransacF(pts1, pts2, M, nIters=100, tol=10):
  '''
  Input:  pts1, Nx2 Matrix
          pts2, Nx2 Matrix
          M, a scaler parameter
          nIters, Number of iterations of the Ransac
          tol, tolerence for inliers
  Output: F, the fundamental matrix
```

```
        inliers, Nx1 bool vector set to true for inliers

    '''
    N = pts1.shape[0]
    pts1_homo, pts2_homo = toHomogenous(pts1), toHomogenous(pts2)
    best_inlier = 0
    inlier_curr = None

    for i in range(nIters):
        choice = np.random.choice(range(pts1.shape[0]), 8)
        pts1_choice = pts1[choice, :]
        pts2_choice = pts2[choice, :]
        F = eightpoint(pts1_choice, pts2_choice, M)
        ress = calc_epi_error(pts1_homo, pts2_homo, F)
        curr_num_inliner = np.sum(ress < tol)
        if curr_num_inliner > best_inlier:
            F_curr = F
            inlier_curr = (ress < tol)
            best_inlier = curr_num_inliner
    inlier_curr = inlier_curr.reshape(inlier_curr.shape[0], 1)
    indixing_array = inlier_curr.flatten()
    pts1_inlier = pts1[indixing_array]
    pts2_inlier = pts2[indixing_array]
    F = eightpoint(pts1_inlier, pts2_inlier, M)
    return F, inlier_curr
```

Below is the implementation of Rodrigues and Inverse Rodrigues Formulas. See the pdf for the detailed explanation of the functions.

In [38]:

```
def rodrigues(r):
    '''
        Input:  r, a 3x1 vector
        Output: R, a rotation matrix
    '''

    r = np.array(r).flatten()
    I = np.eye(3)
    theta = np.linalg.norm(r)
    if theta == 0:
        return I
    else:
        U = (r/theta)[:, np.newaxis]
        Ux, Uy, Uz = r/theta
        K = np.array([[0, -Uz, Uy], [Uz, 0, -Ux], [-Uy, Ux, 0]])
        R = I * np.cos(theta) + np.sin(theta) * K + \
            (1 - np.cos(theta)) * np.matmul(U, U.T)
    return R


def invRodrigues(R):
    '''
    Input:  R, a rotation matrix
    Output: r, a 3x1 vector
    '''

    def s_half(r):
        r1, r2, r3 = r
        if np.linalg.norm(r) == np.pi and (r1 == r2 and r1 == 0 and r2 == 0 and r3 < 0) or (r1 == 0 and r2 <
            return -r
        else:
            return r

    A = (R - R.T)/2
    ro = [A[2, 1], A[0, 2], A[1, 0]]
    s = np.linalg.norm(ro)
    c = (np.sum(np.matrix(R).diagonal()) - 1)/2
    if s == 0 and c == 1:
        r = np.zeros(3)
    elif s == 0 and c == -1:
        col = np.eye(3) + R
        col_idx = np.nonzero(
            np.array(np.sum(col != 0, axis=0)).flatten())[0][0]
        v = col[:, col_idx]
        u = v/np.linalg.norm(v)
```

```
        r = s_half(u * np.pi)
    else:
        u = ro/s
        theta = np.arctan2(s, c)
        r = u * theta

    return r
```

Rodrigues Residual objective function

In [39]:
```
def rodriguesResidual(K1, M1, p1, K2, p2, x):
    '''
    Q5.1: Rodrigues residual.
    Input:  K1, the intrinsics of camera 1
        M1, the extrinsics of camera 1
        p1, the 2D coordinates of points in image 1
        K2, the intrinsics of camera 2
        p2, the 2D coordinates of points in image 2
        x, the flattened concatenationg of P, r2, and t2.
    Output: residuals, 4N x 1 vector, the difference between original and estimated projections
    '''
    N = p1.shape[0]
    # Unpack X
    points3D = x[:3*N].reshape(N, 3)
    points3D_homogeneous = np.hstack((points3D, np.ones((N, 1))))
    r2 = x[3*N:3*N+3]
    t2 = x[3*N+3:]
    rodrigues_r2 = rodrigues(r2)

    # Build Camera projections
    M2 = np.hstack((rodrigues_r2, t2[:, np.newaxis]))
    C1 = K1 @ M1
    C2 = K2 @ M2

    # Project 3D points into 2D
    cam1_proj = C1 @ points3D_homogeneous.T
    cam1_proj /= cam1_proj[2]

    cam2_proj = C2 @ points3D_homogeneous.T
    cam2_proj /= cam2_proj[2]

    # Calculate residuals
    r1 = p1 - cam1_proj[:2].T
    r2 = p2 - cam2_proj[:2].T

    residuals = np.concatenate([r1.flatten(), r2.flatten()])
    residuals = residuals.reshape(4*N, 1)

    return residuals
```

Bundle Adjustment

In [40]:
```
def bundleAdjustment(K1, M1, p1, K2, M2_init, p2, P_init):
    '''
    Q5.2 Bundle adjustment.
    Input:  K1, the intrinsics of camera 1
            M1, the extrinsics of camera 1
            p1, the 2D coordinates of points in image 1
            K2,  the intrinsics of camera 2
            M2_init, the initial extrinsics of camera 1
            p2, the 2D coordinates of points in image 2
            P_init, the initial 3D coordinates of points
    Output: M2, the optimized extrinsics of camera 1
            P2, the optimized 3D coordinates of points
            o1, the starting objective function value with the initial input
            o2, the ending objective function value after bundle adjustment

    Hints:
    (1) Use the scipy.optimize.minimize function to minimize the objective function, rodriguesResidual.
        You can try different (method='..') in scipy.optimize.minimize for best results.

    '''
```

```
    N  = len(p1)
    # Extract the rotation and translation from M2 init
    r0 = invRodrigues(M2_init[:, :3]).flatten()
    t0 = M2_init[:, 3]
    # Concatentate
    x0 = np.hstack([P_init.flatten(), r0, t0])
    obj_start = np.sum(rodriguesResidual(K1, M1, p1, K2, p2, x0)**2)

    cost_f = lambda x: np.sum(rodriguesResidual(K1, M1, p1, K2, p2, x) **2)
    params = scipy.optimize.minimize(cost_f, x0)

    # Unpack results
    x = params.x
    P = x[:3*N].reshape(N, 3)
    rot_optimal = x[3*N:3*N+3]
    trans_optimal = x[3*N+3:][:, np.newaxis]
    rot_optimal = rodrigues(rot_optimal)

    M2 = np.hstack([rot_optimal, trans_optimal])
    obj_end = np.sum(rodriguesResidual(K1, M1, p1, K2, p2, x)**2)

    return M2, P, obj_start, obj_end
```

Put it all together

1. Call the ransacF function to find the fundamental matrix
2. Call the findM2 function to find the extrinsics of the second camera
3. Call the bundleAdjustment function to optimize the extrinsics and 3D points
4. Plot the 3D points before and after bundle adjustment using the plot_3D_dual function

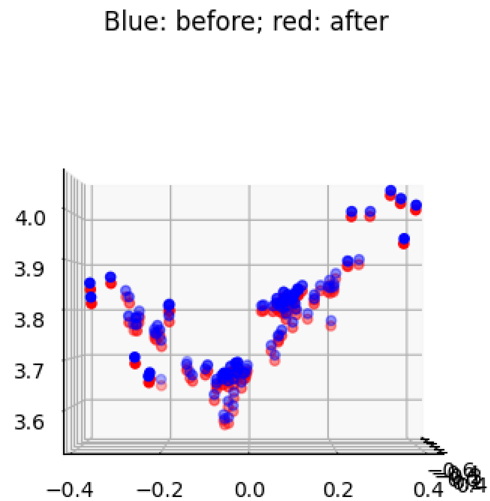
On the given temple data, bundle adjustment can take up to 2 min to run.

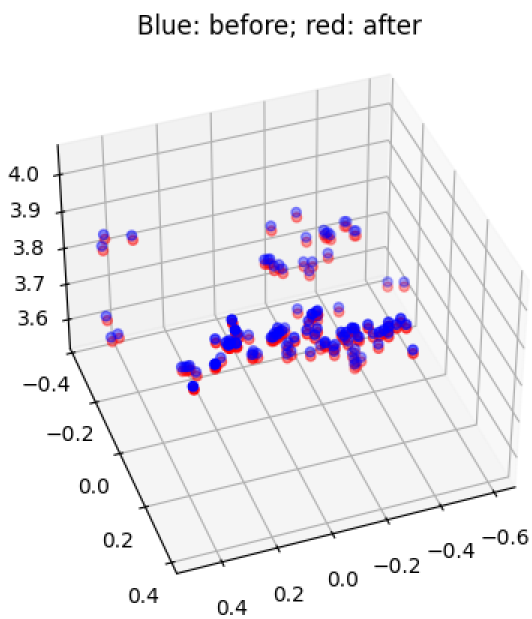
In [41]:
```
# Visualization:
np.random.seed(1)
correspondence = np.load('data/some_corresp_noisy.npz') # Loading noisy correspondences
intrinsics = np.load('data/intrinsics.npz') # Loading the intrinscis of the camera
K1, K2 = intrinsics['K1'], intrinsics['K2']
pts1, pts2 = correspondence['pts1'], correspondence['pts2']
im1 = plt.imread('data/im1.png')
im2 = plt.imread('data/im2.png')
M=np.max([*im1.shape, *im2.shape])

# YOUR CODE HERE
'''
Call the ransacF function to find the fundamental matrix
Call the findM2 function to find the extrinsics of the second camera
Call the bundleAdjustment function to optimize the extrinsics and 3D points
'''
F, inliers = ransacF(pts1, pts2, M, nIters=1000, tol=10)
pts1_inliers = pts1[inliers[:, 0]]
pts2_inliers = pts2[inliers[:, 0]]
M1 = np.array([
    [1, 0, 0, 0],
    [0, 1, 0, 0],
    [0, 0, 1, 0]
])

M2, C2, P0 = findM2(F, pts1_inliers, pts2_inliers, intrinsics)
optimal_M2, optimal_P, obj_start, obj_end = bundleAdjustment(K1, M1, pts1_inliers, K2, M2, pts2_inliers, P(
print(f"Before reprojection error: {obj_start}, After: {obj_end}")
```

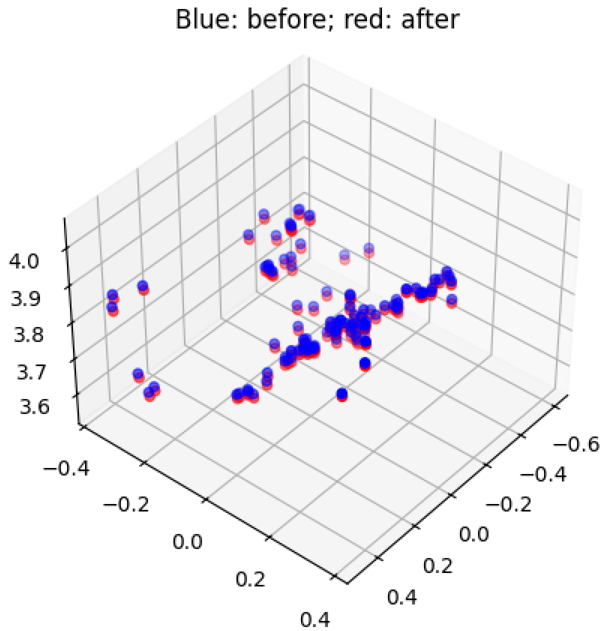
```
Before reprojection error: 210.22837327449096, After: 10.88601863395687
```

In [42]:
```
# helper function for visualization
def plot_3D_dual(P_before, P_after, azim=70, elev=45):
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.set_title("Blue: before; red: after")
    ax.scatter(P_before[:,0], P_before[:,1], P_before[:,2], c = 'blue')
    ax.scatter(P_after[:,0], P_after[:,1], P_after[:,2], c='red')
    ax.view_init(azim=azim, elev=elev)
    plt.draw()
```

```python
# plots the 3d points before and after BA from different viewpoints
plot_3D_dual(P0, optimal_P, azim=0, elev=0)
plot_3D_dual(P0, optimal_P, azim=70, elev=40)
plot_3D_dual(P0, optimal_P, azim=40, elev=40)
```






Blue: before; red: after
4.0
3.9
3.8
3.7
3.6
−0.4
−0.2
0.0
0.2
0.4
−0.6
−0.4
−0.2
0.0
0.2
0.4