Computer Vision HW3 Report

Student ID: B11901164
Name: 陳秉緯

Part 1.

- **Paste your warped canvas**

Part 2.

- **Paste the function code *solve_homography(u, v)* & *warping( )* (both forward & backward)**

```
def solve_homography(u, v):
    """
    This function should return a 3-by-3 homography matrix,
    u, v are N-by-2 matrices, representing N corresponding points for v = T(u)
    :param u: N-by-2 source pixel location matrices
    :param v: N-by-2 destination pixel location matrices
    :return:
    """
    N = u.shape[0]
    H = None


    if v.shape[0] is not N:
        print('u and v should have the same size')
        return None
    if N < 4:
        print('At least 4 points should be given')


    # TODO: 1.forming A
    A = np.zeros((2*N, 9))
```

```
    for i in range(N):
        x, y = u[i, 0], u[i, 1]
        x_prime, y_prime = v[i, 0], v[i, 1]

        A[2*i] = [x, y, 1, 0, 0, 0, -x*x_prime, -y*x_prime, -x_prime]
        A[2*i+1] = [0, 0, 0, x, y, 1, -x*y_prime, -y*y_prime, -y_prime]


    # TODO: 2.solve H with A
    _, _, Vt = np.linalg.svd(A)
    h = Vt[-1, :]  # h is the last column of V == the last row of Vt
    H = h.reshape(3, 3) # reshape h

    # Normalize the homography matrix
    H = H / H[2, 2]


    return H

def warping(src, dst, H, ymin, ymax, xmin, xmax, direction='b'):
    """
    Perform forward/backward warpping without for loops. i.e.
    for all pixels in src(xmin~xmax, ymin~ymax),  warp to destination
          (xmin=0,ymin=0)  source                      destination
                        |--------|              |------------------------|
                        |        |              |                        |
                        |        |     warp     |                        |
    forward warp        |        |  --------->  |                        |
                        |        |              |                        |
                        |--------|              |------------------------|
                                (xmax=w,ymax=h)


    for all pixels in dst(xmin~xmax, ymin~ymax),  sample from source
                           source                      destination
                        |--------|              |------------------------|
                        |        |              | (xmin,ymin)            |
                        |        |     warp     |           |--|         |
    backward warp       |        |  <---------  |           |__|         |
                        |        |              |             (xmax,ymax)|
                        |--------|              |------------------------|


    :param src: source image
    :param dst: destination output image
    :param H:
    :param ymin: lower vertical bound of the destination(source, if forward warp)
pixel coordinate
    :param ymax: upper vertical bound of the destination(source, if forward warp)
pixel coordinate
    :param xmin: lower horizontal bound of the destination(source, if forward warp)
pixel coordinate
```

```
    :param xmax: upper horizontal bound of the destination(source, if forward warp)
pixel coordinate
    :param direction: indicates backward warping or forward warping
    :return: destination output image
    """


    h_src, w_src, ch = src.shape
    h_dst, w_dst, ch = dst.shape
    H_inv = np.linalg.inv(H)


    # TODO: 1.meshgrid the (x,y) coordinate pairs
    y_grid, x_grid = np.meshgrid(np.arange(ymin, ymax), np.arange(xmin, xmax),
indexing='ij')

    # TODO: 2.reshape the destination pixels as N x 3 homogeneous coordinate
    y_grid = y_grid.flatten()
    x_grid = x_grid.flatten()

    # Create homogeneous coordinates [x,y,1]
    ones = np.ones_like(x_grid)
    coords = np.stack((x_grid, y_grid, ones), axis=0)  # 3 x N


    if direction == 'b':
        # TODO: 3.apply H_inv to the destination pixels and retrieve (u,v) pixels,
then reshape to (ymax-ymin),(xmax-xmin)
        # Apply H_inv to destination pixels
        src_coords = H_inv @ coords  # 3 x N

        # Convert to inhomogeneous coordinates
        src_coords = src_coords / src_coords[2, :]  # normalize by z

        # Get source coordinates
        src_x = src_coords[0, :]
        src_y = src_coords[1, :]

        # Reshape to (ymax-ymin),(xmax-xmin)
        src_x = src_x.reshape((ymax-ymin, xmax-xmin))
        src_y = src_y.reshape((ymax-ymin, xmax-xmin))


        # TODO: 4.calculate the mask of the transformed coordinate (should not
exceed the boundaries of source image)
        # Create mask for valid coordinates
        mask = (src_x >= 0) & (src_x < w_src-1) & (src_y >= 0) & (src_y < h_src-1)

        # TODO: 5.sample the source image with the masked and reshaped transformed
coordinates
        # Apply mask before conversion to avoid warnings
        src_x_masked = src_x.copy()
        src_y_masked = src_y.copy()
```

```
        # Replace invalid values with zeros to avoid warnings
        src_x_masked[~mask] = 0
        src_y_masked[~mask] = 0

        # Use bilinear interpolation on masked coordinates
        src_x_floor = np.floor(src_x_masked).astype(np.int32)
        src_y_floor = np.floor(src_y_masked).astype(np.int32)
        src_x_ceil = np.ceil(src_x_masked).astype(np.int32)
        src_y_ceil = np.ceil(src_y_masked).astype(np.int32)

        # Ensure indices don't exceed boundaries
        src_x_floor = np.clip(src_x_floor, 0, w_src-1)
        src_y_floor = np.clip(src_y_floor, 0, h_src-1)
        src_x_ceil = np.clip(src_x_ceil, 0, w_src-1)
        src_y_ceil = np.clip(src_y_ceil, 0, h_src-1)

        # Get weights for bilinear interpolation
        x_weight = src_x_masked - src_x_floor
        y_weight = src_y_masked - src_y_floor

        # Sample the source image
        top_left = src[src_y_floor, src_x_floor]
        top_right = src[src_y_floor, src_x_ceil]
        bottom_left = src[src_y_ceil, src_x_floor]
        bottom_right = src[src_y_ceil, src_x_ceil]

        # Apply bilinear interpolation
        top = top_left * (1 - x_weight[..., np.newaxis]) + top_right *
x_weight[..., np.newaxis]
        bottom = bottom_left * (1 - x_weight[..., np.newaxis]) + bottom_right *
x_weight[..., np.newaxis]

        interpolated = top * (1 - y_weight[..., np.newaxis]) + bottom *
y_weight[..., np.newaxis]

        # TODO: 6. assign to destination image with proper masking
        # Apply the mask to update only valid pixels
        dst[ymin:ymax, xmin:xmax][mask] = interpolated[mask]


    elif direction == 'f':
        # TODO: 3.apply H to the source pixels and retrieve (u,v) pixels, then
reshape to (ymax-ymin),(xmax-xmin)
        # Apply H to source pixels
        dst_coords = H @ coords  # 3 x N

        # Convert to inhomogeneous coordinates
        dst_coords = dst_coords / dst_coords[2, :]  # normalize by z

        # Get destination coordinates
        dst_x = dst_coords[0, :]
        dst_y = dst_coords[1, :]
```

```
        # TODO: 4.calculate the mask of the transformed coordinate (should not
exceed the boundaries of destination image)
        # Create mask for valid coordinates
        mask = (dst_x >= 0) & (dst_x < w_dst) & (dst_y >= 0) & (dst_y < h_dst)

        # TODO: 5.filter the valid coordinates using previous obtained mask
        # Get the valid source and destination coordinates
        valid_dst_x = dst_x[mask].astype(np.int32)
        valid_dst_y = dst_y[mask].astype(np.int32)
        valid_src_x = x_grid[mask]
        valid_src_y = y_grid[mask]

        # TODO: 6. assign to destination image using advanced array indicing
        # Copy pixel values from source to destination
        dst[valid_dst_y, valid_dst_x] = src[valid_src_y, valid_src_x]


    return dst
```

● Briefly introduce the interpolation method you use

For the backward warping implementation, I used bilinear interpolation to sample pixel values from the source image. This method provides smoother results than nearest neighbor interpolation by considering the weighted average of the four nearest pixels.
The bilinear interpolation works as follows:

1. For each pixel in the destination image, we compute its corresponding location in the source image using the inverse homography.
2. Since this location is typically not at integer coordinates, we take the four nearest pixels (top-left, top-right, bottom-left, bottom-right).
3. We calculate weights based on the distance to each of these four pixels.
4. The final pixel value is the weighted sum of these four pixels:
    - First interpolate horizontally to get top and bottom values
    - Then interpolate vertically between these values

Part 3.

● Paste the 2 warped images and the link you find

http://media.ee.ntu.edu.tw/courses/cv/25S/

http://media.ee.ntu.edu.tw/courses/cv/25S/

● Discuss the difference between 2 source images, are the warped results the same or different?

Although both QR codes decode to the same URL, they appear slightly different visually.

● If the results are the same, explain why. If the results are different, explain why?

The background image of BL_secret2.png may be distorted from that of BL_secret1.png.
The homography transformation might introduce slight differences in pixel interpolation during warping.

Part 4.

● Paste your stitched panorama

● Can all consecutive images be stitched into a panorama?

No

● If yes, explain your reason. If not, explain under what conditions will result in a failure?

Conditions:

1. Insufficient overlap between images: Feature matching requires enough common content between adjacent images. If the overlap is too small, not enough reliable feature matches can be found.
2. Dynamic objects: Moving objects appearing in different positions across the input images can cause ghosting or stitching failures.