Computer Vision HW3 Report

Student ID: R12942159
Name: 邱亮茗

Part 1.

- **Paste your warped canvas**

Part 2.

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

```python
def solve_homography(u, v):
    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 = []
    for (a, b), (c, d) in zip(u, v):
        A.append([a, b, 1, 0, 0, 0, -a*c, -b*c, -c])
        A.append([0, 0, 0, a, b, 1, -a*d, -b*d, -d])
    A = np.array(A)

    # TODO: 2.solve H with A
    _, _, Vt = np.linalg.svd(A)
    h = Vt[-1].reshape(3, 3)
    h = h / h[2, 2]  # let h(2,2) = 1
    return h
```

```python
def warping(src, dst, H, ymin, ymax, xmin, xmax, direction='b'):
    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
    x_grid, y_grid = np.meshgrid(np.arange(xmin, xmax, 1), np.arange(ymin, ymax, 1), sparse = False)

    # TODO: 2.reshape the destination pixels as N x 3 homogeneous coordinate
    x_flat = x_grid.flatten()
    y_flat = y_grid.flatten()
    ones = np.ones_like(x_flat)
    src_coords = np.stack([x_flat, y_flat, ones], axis=0)

    if direction == 'b':
        # TODO: 3.apply H_inv to the destination pixels and retrieve (u,v) pixels, then reshape to (ymax-ymin),
        dst2src_coords = H_inv @ src_coords
        dst2src_coords /= dst2src_coords[-1]
        Ux, Uy = dst2src_coords[0].reshape(x_grid.shape), dst2src_coords[1].reshape(x_grid.shape)

        # TODO: 4.calculate the mask of the transformed coordinate (should not exceed the boundaries of source
        mask = ((Ux >= 0) & (Ux < w_src-1)) & ((Uy >= 0) & (Uy < h_src-1))

        # TODO: 5.sample the source image with the masked and reshaped transformed coordinates
        Ux_mask, Uy_mask = Ux[mask], Uy[mask]

        # Bilinear interpolation
        Ux_mask_floor = Ux_mask.astype(int)
        Uy_mask_floor = Uy_mask.astype(int)
        dx = (Ux_mask - Ux_mask_floor).reshape(-1, 1)
        dy = (Uy_mask - Uy_mask_floor).reshape(-1, 1)
        src_interpol = np.zeros((h_src, w_src, ch))
        src_interpol[Uy_mask_floor, Ux_mask_floor, :] = ((1-dy) * (1-dx) * src[Uy_mask_floor, Ux_mask_floor, :]
                                                         + (1-dy) * (dx) * src[Uy_mask_floor, Ux_mask_floor+1,
                                                         + (dy) * (1-dx) * src[Uy_mask_floor+1, Ux_mask_floor,
                                                         + (dy) * (dx) * src[Uy_mask_floor+1, Ux_mask_floor+1,

        # TODO: 6. assign to destination image with proper masking
        dst[ymin:ymax,xmin:xmax][mask] = src_interpol[Uy_mask_floor, Ux_mask_floor]

    elif direction == 'f':
        # TODO: 3.apply H to the source pixels and retrieve (u,v) pixels, then reshape to (ymax-ymin),(xmax-xmi
        src2dst_coords = H @ src_coords
        src2dst_coords /= src2dst_coords[-1]
        src2dst_coords = src2dst_coords.astype(int)
        Vx, Vy = src2dst_coords[0].reshape(x_grid.shape), src2dst_coords[1].reshape(x_grid.shape)

        # TODO: 4.calculate the mask of the transformed coordinate (should not exceed the boundaries of destina
        mask = ((Vx < w_dst) & (0 <= Vx)) & ((Vy < h_dst) & (0 <= Vy))

        # TODO: 5.filter the valid coordinates using previous obtained mask
        Vx_mask, Vy_mask = Vx[mask], Vy[mask]

        # TODO: 6. assign to destination image using advanced array indicing
        dst[Vy_mask, Vx_mask, :] = src[mask]

    return dst
```

• Briefly introduce the interpolation method you use

Bilinear interpolation is a method used to estimate the value of a function—such as pixel intensity—at a non-integer coordinate by computing the weighted average of its four nearest neighboring pixels.

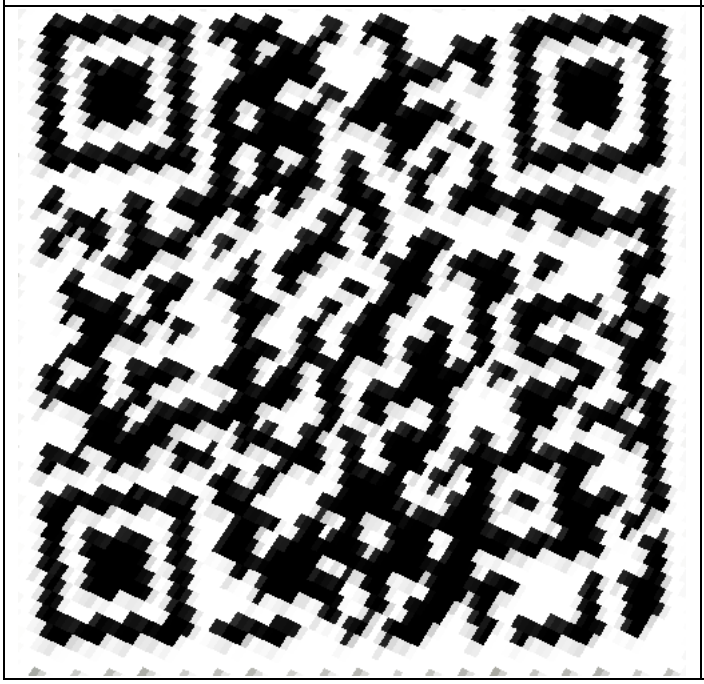
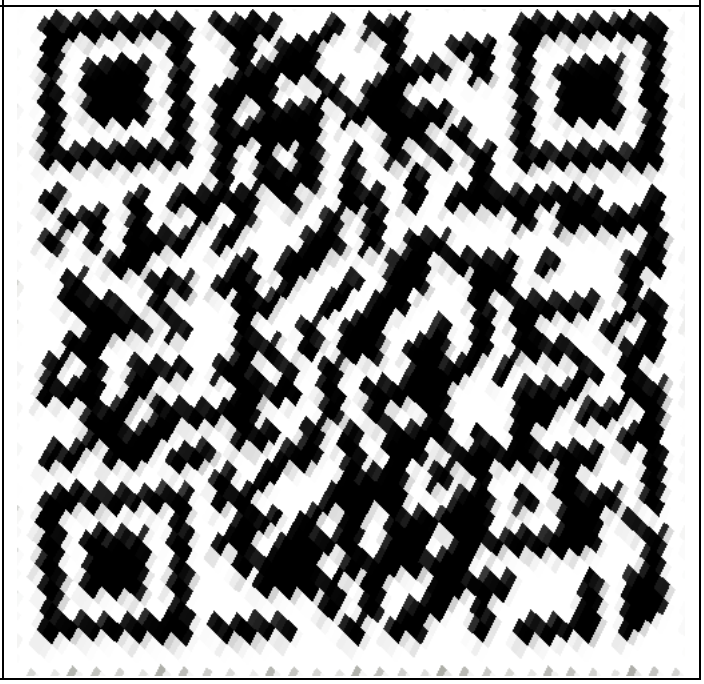
For example, given a point (x,y) that lies between four known pixel values—Q_{11} (top-left), Q_{21} (top-right), Q_{12} (bottom-left), and Q_{22} (bottom-right)—the value at (x,y) is calculated as follows:

$$f(x,y) = (1-dx)(1-dy)Q_{11} + dx(1-dy)Q_{21} + (1-dx)dyQ_{12} + dxdyQ_{22}$$

Where $dx = x - \lfloor x \rfloor,\ dy = y - \lfloor y \rfloor$.

Part 3.

• Paste the 2 warped images and the link you find

Output3_1.png	Output3_2.png
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?

The two images show the same location from different perspectives. The BL_secret2 image appears to be taken with a wide-angle lens, which creates a more exaggerated perspective. You can notice the curvature of the cylindrical structure and how the walls on both sides appear to bend outward slightly. In contrast, the BL_secret1 image has a more natural or standard field of view. The lines are straighter, and there's less distortion, indicating it was likely taken with a normal or narrower lens. Despite the perspective distortion, the warped QR codes in both images encode the same content.

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

Even when the image is taken with a wide-angle lens—causing perspective distortion—homography estimation allows us to model the geometric relationship between the image plane and the flat QR code surface. The estimated homography matrix captures this transformation and enables us to rectify the distorted QR code region. By applying backward warping, each pixel in the rectified (output) image is mapped back to its corresponding location in the original image. This effectively corrects the perspective distortion. Since the same physical QR code exists in both images, and homography normalization brings them to a common view.

Part 4.

• Paste your stitched panorama

• Can all consecutive images be stitched into a panorama?

Not all consecutive images can be successfully stitched into a panorama. For stitching to work, consecutive images must have sufficient overlapping regions with consistent visual features that can be matched accurately. If the overlap is too small, the lighting conditions differ significantly, or there's motion blur or parallax (caused by large viewpoint changes), feature matching and homography estimation can fail, resulting in misalignment or visible seams. Therefore, successful panorama stitching depends on image quality, proper overlap, and stable camera movement.

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

Insufficient Overlap: If there's not enough shared region between consecutive images, feature detectors like ORB cannot find enough matching keypoints for homography estimation.

Lack of Distinct Features: Images with large textureless areas (e.g., sky or blank walls) make it hard to detect and match keypoints.

Lighting and Exposure Differences: Significant differences in brightness, contrast, or shadows can confuse feature matching and blending.

Motion Blur or Occlusion: Moving objects or blur from camera shake introduce inconsistent features that disrupt alignment.