Computer Vision HW3 Report

Student ID: R12945040

Name: 郭思言

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):
    """
    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 = []
    for i in range(N):
        A.append([u[i][0], u[i][1], 1, 0, 0, 0, -u[i][0] * v[i][0], -u[i][1] * v[i][0], -v[i][0]])
        A.append([0, 0, 0, u[i][0], u[i][1], 1, -u[i][0] * v[i][1], -u[i][1] * v[i][1], -v[i][1]])

    A = np.array(A)
    U, S, V = np.linalg.svd(A)

    # TODO: 2.solve H with A
    # (last column of V matrix of SVD)
    H = V[-1, :].reshape(3, 3)

    return H
```

```python
def warping(src, dst, H, direction="b"):
    h_src, w_src, ch = src.shape
    h_dst, w_dst, ch = dst.shape  # (height, width) = (1275, 1920)
    H_inv = np.linalg.inv(H)

    warped = dst.copy()

    # backward warping: inverse transform destination pixels to source pixels
    if direction == "b":
        # TODO: 1.meshgrid the (x,y) coordinate pairs (with destination points)
        x = np.arange(0, w_dst, 1)
        y = np.arange(0, h_dst, 1)
        xx, yy = np.meshgrid(x, y)

        # TODO: 2.reshape the destination pixels as N x 3 homogeneous coordinate
        xx, yy = xx.flatten()[:, np.newaxis], yy.flatten()[:, np.newaxis]
        ones = np.ones((len(xx), 1))

        des_coor = np.concatenate((xx, yy, ones), axis=1).astype(np.int32)

        # TODO: 3.apply H_inv to the destination pixels and retrieve (u,v) pixels, then reshape to (ymax-ymin), (xmax-xmin)
        Resource_pixel = H_inv.dot(des_coor.T).T  # (N * 3)

        # divide the first two columns by the third, converts the homogeneous coordinates back to Cartesian coordinates
        Resource_pixel[:, :2] = Resource_pixel[:, :2] / Resource_pixel[:, 2][:, np.newaxis]

        # TODO: 4.calculate the mask of the transformed coordinate (should not exceed the boundaries of source image)

        out_boundary = []

        if (Resource_pixel[:, 0] < 0).any():
            out_boundary += np.where(Resource_pixel[:, 0] < 0)[0].tolist()
        if (Resource_pixel[:, 1] < 0).any():
            out_boundary += np.where(Resource_pixel[:, 1] < 0)[0].tolist()
        if (Resource_pixel[:, 0] > w_src - 1).any():
            out_boundary += np.where(Resource_pixel[:, 0] > (w_src - 1))[0].tolist()
        if (Resource_pixel[:, 1] > h_src - 1).any():
            out_boundary += np.where(Resource_pixel[:, 1] > (h_src - 1))[0].tolist()

        # TODO: 5.sample the source image with the masked and reshaped transformed coordinates
        if len(out_boundary):
            Resource_pixel = np.delete(Resource_pixel, out_boundary, 0)
            des_coor = np.delete(des_coor, out_boundary, 0)

        # TODO: 6. assign to destination image with proper masking
        tx = Resource_pixel[:, 0].astype(np.int)
        ty = Resource_pixel[:, 1].astype(np.int)
        dx = Resource_pixel[:, 0] - tx
        dy = Resource_pixel[:, 1] - ty

        # Bilinear interpolation
        ones = np.ones(len(dx)).astype(np.float)
        warped[des_coor[:, 1], des_coor[:, 0]] = (
            (((ones - dx) * (ones - dy))[:, np.newaxis] * src[ty, tx])
            + ((dx * (ones - dy))[:, np.newaxis] * src[ty, tx + 1])
            + ((dx * dy)[:, np.newaxis] * src[ty + 1, tx + 1])
            + (((ones - dx) * dy)[:, np.newaxis] * src[ty + 1, tx])
        )

    elif direction == "f":

        # TODO: 1.meshgrid the (x,y) coordinate pairs
        x = np.arange(0, w_src - 1, 1)
        y = np.arange(0, h_src - 1, 1)
        xx, yy = np.meshgrid(x, y)

        # TODO: 2.reshape the resource pixels as N x 3 homogeneous coordinate
        xx, yy = xx.flatten()[:, np.newaxis], yy.flatten()[:, np.newaxis]
        ones = np.ones((len(xx), 1))

        des_coor = np.concatenate((xx, yy, ones), axis=1).astype(np.int)

        # TODO: 3.apply H to the source pixels and retrieve (u,v) pixels, then reshape to (ymax-ymin), (xmax-xmin)
        Resource_pixel = H.dot(des_coor.T).T  # (N * 3)

        # divide the first two columns by the third, converts the homogeneous coordinates back to Cartesian coordinates
        Resource_pixel[:, :2] = Resource_pixel[:, :2] / Resource_pixel[:, 2][:, np.newaxis]

        # TODO: 4.calculate the mask of the transformed coordinate (should not exceed the boundaries of destination image)
        out_boundary = []

        if (Resource_pixel[:, 0] < 0).any():
            out_boundary += np.where(Resource_pixel[:, 0] < 0)[0].tolist()
        if (Resource_pixel[:, 1] < 0).any():
            out_boundary += np.where(Resource_pixel[:, 1] < 0)[0].tolist()
        if (Resource_pixel[:, 0] > w_dst - 1).any():
            out_boundary += np.where(Resource_pixel[:, 0] > (w_dst - 1))[0].tolist()
        if (Resource_pixel[:, 1] > h_dst - 1).any():
            out_boundary += np.where(Resource_pixel[:, 0] > (h_dst - 1))[0].tolist()

        # TODO: 5.filter the valid coordinates using previous obtained mask
        if len(out_boundary):
            Resource_pixel = np.delete(Resource_pixel, out_boundary, 0)
            des_coor = np.delete(des_coor, out_boundary, 0)

        # TODO: 6. assign to destination image using advanced array indicing
        tx = Resource_pixel[:, 0].astype(np.int)
        ty = Resource_pixel[:, 1].astype(np.int)
        dx = Resource_pixel[:, 0] - tx
        dy = Resource_pixel[:, 1] - ty

        # warped[Resource_pixel[:, 1], Resource_pixel[:, 0]] = src[des_coor[:, 1], des_coor[:, 0]]

        ones = np.ones(len(dx)).astype(np.float)

        # Bilinear interpolation
        warped[ty, tx] = (
            (((ones - dx) * (ones - dy))[:, np.newaxis] * src[des_coor[:, 1], des_coor[:, 0]])
            + ((dx * (ones - dy))[:, np.newaxis] * src[des_coor[:, 1], des_coor[:, 0] + 1])
            + ((dx * dy)[:, np.newaxis] * src[des_coor[:, 1] + 1, des_coor[:, 0] + 1])
            + (((ones - dx) * dy)[:, np.newaxis] * src[des_coor[:, 1] + 1, des_coor[:, 0]])
        )

    return warped
```

- **Briefly introduce the interpolation method you use**

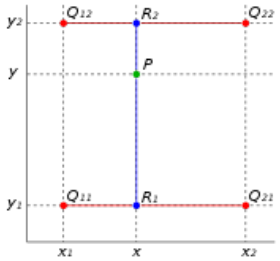
I use "**bilinear interpolation**" for both forward and backward warping. Bilinear interpolation is a widely used method for estimating the value of a function at any point inside a rectangle based on the values at the rectangle's four corners.

Brief introduction of "**bilinear interpolation**":

1. **Data Points:**
   Bilinear interpolation assumes that the function is linear between adjacent data points. For a point (x, y) inside the rectangle defined by four points (x1, y1), (x1, y2), (x2, y1), and (x2, y2), the value f(x, y) is estimated using the values f(x1, y1), f(x1, y2), f(x2, y1), and f(x2, y2).

2. **Weights:**
   Bilinear interpolation computes the weights based on the distances between the point (x, y) and the four corners of the rectangle. The closer the point is to a corner, the higher the weight assigned to that corner.

3. **Interpolation formula:**



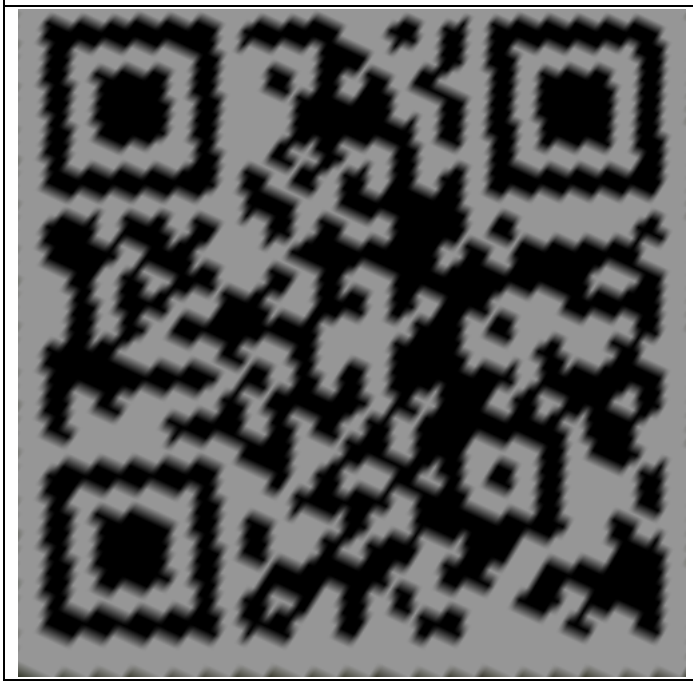
$$
\begin{aligned}
f(x,y) &= \frac{y_2-y}{y_2-y_1}f(x,y_1)+\frac{y-y_1}{y_2-y_1}f(x,y_2)\\
&= \frac{y_2-y}{y_2-y_1}\left(\frac{x_2-x}{x_2-x_1}f(Q_{11})+\frac{x-x_1}{x_2-x_1}f(Q_{21})\right)+\frac{y-y_1}{y_2-y_1}\left(\frac{x_2-x}{x_2-x_1}f(Q_{12})+\frac{x-x_1}{x_2-x_1}f(Q_{22})\right)\\
&= \frac{1}{(x_2-x_1)(y_2-y_1)}\left(f(Q_{11})(x_2-x)(y_2-y)+f(Q_{21})(x-x_1)(y_2-y)+f(Q_{12})(x_2-x)(y-y_1)+f(Q_{22})(x-x_1)(y-y_1)\right)
\end{aligned}
$$

(reference: wiki)

3. **Advantages:**
   Bilinear interpolation produces smoother results compared to nearest neighbor interpolation, and it is computationally efficient and easy to implement.

Part 3.

- **Paste the 2 warped images and the link you find**

output3_1.png	output3_2.png
	
	

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

By observing the two warped results, it can be noted that the first result is clearer, while the second is blurrier. Comparing the two source images, although their angles are roughly the same, the QR code in the second source image is slightly distorted, rotated, and deformed. This introduces additional factors during the calculation of the homography, resulting in some degree of error.

- **If the results are the same, explain why. If the results are different, explain why?**

The difference in the results is due to the QR code in the second source image being distorted, rotated, and deformed, while the first image remains relatively clear. These variations lead to a more complex situation during the calculation of the homography, increasing the uncertainty of the estimation and consequently causing an increase in the blurriness of the warped result.

Part 4.

- **Paste your stitched panorama**

- **Can all consecutive images be stitched into a panorama?**

Not all consecutive images can be stitched into a panorama. The success of stitching images into a panorama depends on several factors, including the overlap between the images, the presence of distinctive features for matching, and the absence of significant distortions or changes in perspective between the images.

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

If there is insufficient overlap between images, making it difficult to find matching features, or if there are significant distortions or changes in perspective between images, the stitching process may fail, resulting in a disjointed or distorted panorama.