Computer Vision Homework 3

資工四 B05902115 陳建丞

Part 1: Estimating Homography

- *Result*

- *Implementation*
  - **Solve the homography matrix**
    - function: `solve_homography(u, v)`
    - method:

      I use **solution2** to solve the homography. Let $h_{11}^2 + \cdots + h_{33}^2 = 1$

      Then we get the linear system below.

$$\begin{bmatrix} u_{x,1} & v_{y,1} & 1 & 0 & 0 & 0 & -u_{x,1}v_{x,1} & -u_{y,1}v_{x,1} & -v_{x,1} \\ 0 & 0 & 0 & u_{x,1} & v_{y,1} & 1 & -u_{x,1}v_{y,1} & -u_{y,1}v_{y,1} & -v_{y,1} \\ u_{x,2} & v_{y,2} & 1 & 0 & 0 & 0 & -u_{x,2}v_{x,2} & -u_{y,2}v_{x,2} & -v_{x,2} \\ 0 & 0 & 0 & u_{x,2} & v_{y,2} & 1 & -u_{x,2}v_{y,2} & -u_{y,2}v_{y,2} & -v_{y,2} \\ & & & & & \vdots & & & \\ & & & & & 2N \times 9 & & & \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix} = \begin{bmatrix} u_{x,1} \\ u_{y,1} \\ u_{x,2} \\ u_{y,2} \\ \vdots \\ 2N \times 1 \end{bmatrix}$$

      Next, apply SVD decomposition $Ah = 0 \Rightarrow A = U\Sigma V^T$

After the decomposition, h is the last column of V

To implement this process, I use `np.linalg.svd(A)`. This function will return the decompositon vectors. Then we can extract V and get h from V.

In Part1, I use forward warping. Therefore, My homography matrix h is the mapping matrix from input image to canvas image(akihabara.jpg).

- **Transformation**
  - function: `transform(img, canvas, corners)`
  - method:

    I traverse through the input image and use `np.dot()` to extract the projected coordinates on the canvas image. Since the projected coordinates should be in the form of $\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$, the result $\begin{bmatrix} x' \\ y' \\ c \end{bmatrix}$ of `np.dot()` should be divided by c. After extracting the projected coordinates, I write the pixels values from input image to the projected coordinates.

Part 2: Unwarp the Screen

- *Result*

The QR code directs to http://media.ee.ntu.edu.tw/courses/cv/19F/?fbclid=IwAR3NrvhmUWHMqdsaXgeZwb6kaAqFQ92A4Ye4Tn7qiCgrmgRNzVOoz_dpJbw

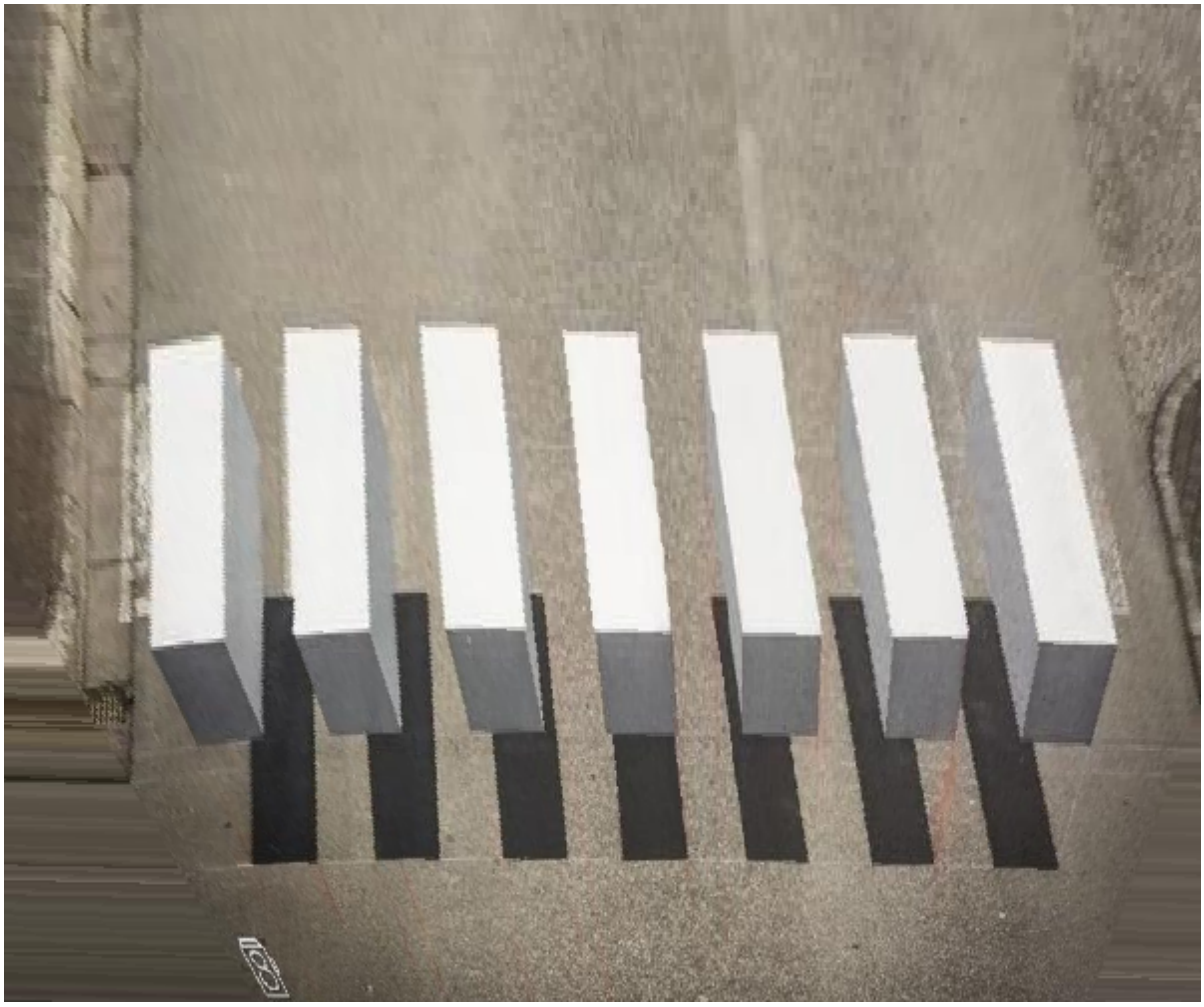
- *Implementation*
  - **Backward Warping**

    Same as foward warping from Part1, I use the `solve_homography(u, v)` function the calculate the homography matrix h. However, the projection direction is different. In this part, the projected matrix is the mapping from canvas image(output image) to input image(QR_code.jpg). After extracting h, I traverse all pixels in canvas image and use `np.dot()` to get the projected coordinates on input image. In most case, the projected coordinates will be in the type of float. Therefore, I use **bilinear interporlation** to calculate the values with the 4 nearest pixels values. This way, the

result image will have no hole in some places if backward warping applied.

Part 3: Unwarp the 3D Illusion

- *Result*



- *Implementation*
    - **Backward Warping**

      The process is totally the same as part2. The only difference is that there are no specific points that we aim to transform to. Different points selection might lead to critical change.

The image above is another result if I choose corner points with longer length to project to. This makes the view looks very weired, far from the ground-truth top view image. Therefore, the selection of corner points is critical is this part. Despite so, it's impossible to fully match the ground-truth image because there is some information that does not exist in the front view image. Like the bottom part of the rock sidewalk on the rightside of top view image. It does not exist in the front view image, so we can't not just use backward warping technics to realize it. Other image processing technics are required if we want to fully match the ground-truth top view image.

-

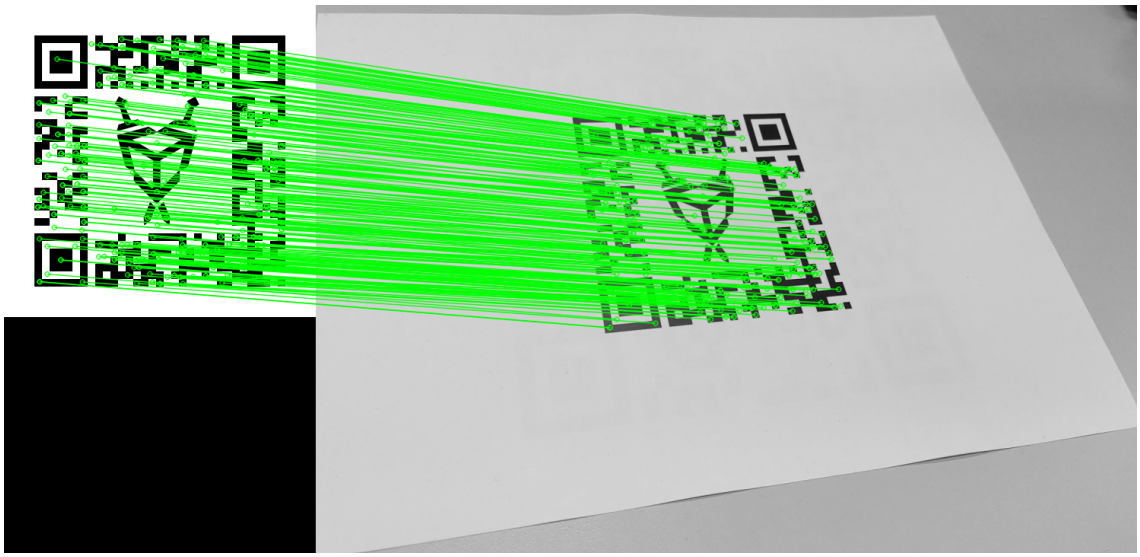
Part 4: Simple AR

- *Result (single frame)*

- *Implementation*
  - **Preprocess**
    1. Resize reference image to the size of template image (310 x 310)
    2. Convert the template image to grayscale
    3. Make a copy of frame image as grayscale
    4. Resize frame image to reduce the processing time (Resize to 1080 x 720)
  - **Feature detection**

    I use **SIFT detector** to detect the features and object in both template image and frame image(gray). The algorithm can be realized with the toolkit in `cv2`

    - `cv2.xfeatures2d.SIFT_create()` : Initialize the SIFT detector.
    - `sift.detecAndCompute()` : Find the keypoints and descriptors with SIFT
  - **Feature matching**



I use **FlannBasedMatcher** to match the features between template image and frame image(gray) from the result of SIFT detector. The algorithm can be realized with the toolkit in `cv2`.

- `cv2.FlannBasedMatcher()` : Initialize a FlannBasedMatcher model.
- `flann.knnMatch()` : Returns the matches result with the given keypoints.

- **Backward warping**

  The result will have some holes between pixels if I apply forward warping to transform the image. Therefore, I use backward warping. The homography solution and transformation method is totally the same as previous part. The only difference is that the projected coordinates to which the reference image will project differ every frame. Also, Unlike previous parts, the coordinates will not form a rectangle, so the coordinates will be more difficult to find. We need to find all these coordinates which we want to transform to so that we can apply backward warping. The process I use to find the coordinates is as follow.

  1. Find the 4 vertex that will shape a quadrilateral which bounds all the coordinate we want to project to. These 4 vertex coordinates can be found by applying forward warping to transform the 4 vertex in template image. I use the 4 projected coordinates as the 4 vertex of the bounding quadrilateral.
  2. Use `Path()` function from `matplotlib.path` to make a polygon with the 4 vertex. Next, we can use `path.contains_points()` to returns a mask to show whether given points are inside the boudning polygon.

  After finding the coordinates on the frame image we want to project to. We can apply backward warping to find the coorsponding pixels values and write it on to frame image. After that, resize the frame image to it original size (1920 x 1080)