Computer Vision HW3 - Projective Geometry

資工四 b05902058 陳竣宇

Part 1: Estimating Homography

I implement the second solution for estimating homography matrix.

- Function solve_homography(u, v):

```python
# u, v are N-by-2 matrices, representing N corresponding points for v = T(u)
# this function should return a 3-by-3 homography matrix
def solve_homography(u, v):
    N = u.shape[0]
    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')
    # A = np.zeros((2*N, 8))
    # if you take solution 2:
    A = np.zeros((2*N, 9))
    b = np.zeros((2*N, 1))
    H = np.zeros((3, 3))

    # TODO: some magic
    dst = convert_to_homography_param(v.T)
    src = convert_to_homography_param(u.T)
    ## adjust the points
    src, c1 = normalize(src)
    dst, c2 = normalize(dst)
    ## contruct matrix A
    for i in range(N):
        A[2 * i] = np.array([-src[0][i], -src[1][i], -1, 0, 0, 0, dst[0][i] * src[0][i], dst[0][i] * src[1][i], dst[0][i]])
        A[2 * i + 1] = np.array([0, 0, 0, -src[0][i], -src[1][i], -1, dst[1][i] * src[0][i], dst[1][i] * src[1][i], dst[1][i]])
    ## use svd to get homography matrix
    U, S, V = np.linalg.svd(A)
    H = V[-1].reshape(3, 3)
    H = np.dot(np.linalg.inv(c2), np.dot(H, c1))
    return H / H[2, 2]

def normalize(point_list):
    # type: (np.ndarray) -> (np.ndarray, np.ndarray)
    """
    :param point_list: point list to be normalized
    :return: normalization results
    """
    m = np.mean(point_list[:2], axis=1)
    max_std = max(np.std(point_list[:2], axis=1)) + 1e-9
    c = np.diag([1 / max_std, 1 / max_std, 1])
    c[0][2] = -m[0] / max_std
    c[1][2] = -m[1] / max_std
    return np.dot(c, point_list), c

def convert_to_homography_param(point_list):
    """
    :return: matrix of homography param (3 x N). N = width x height.
    """
    return np.vstack((point_list, np.ones((1, point_list.shape[1]))))
```

- Canvas image:

- Material (5 images projected to the canvas image):

img1	img2	img3	img4	img5
				

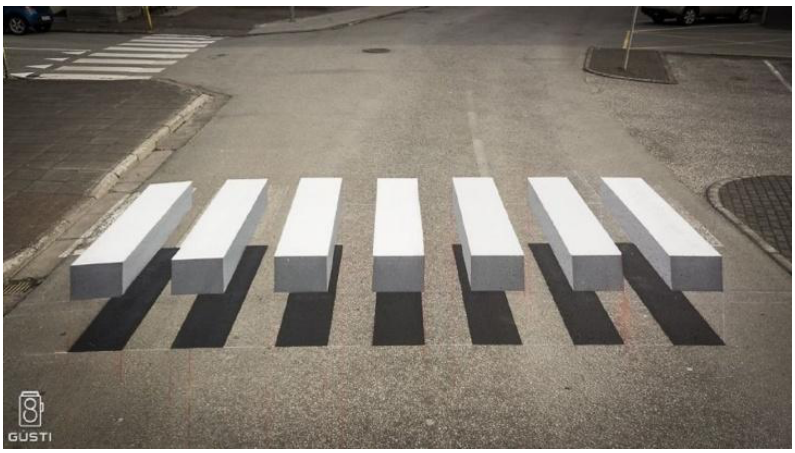
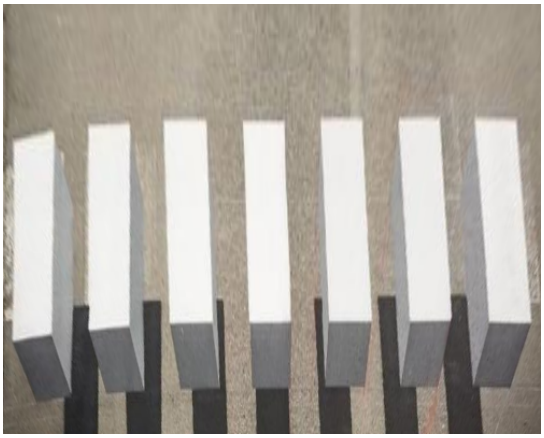
- Result

Part 2: Unwarp the Screen

source image	frontal QR-code image
	

- Decoded link: http://media.ee.ntu.edu.tw/courses/cv/19F/ (http://media.ee.ntu.edu.tw/courses/cv/19F/)

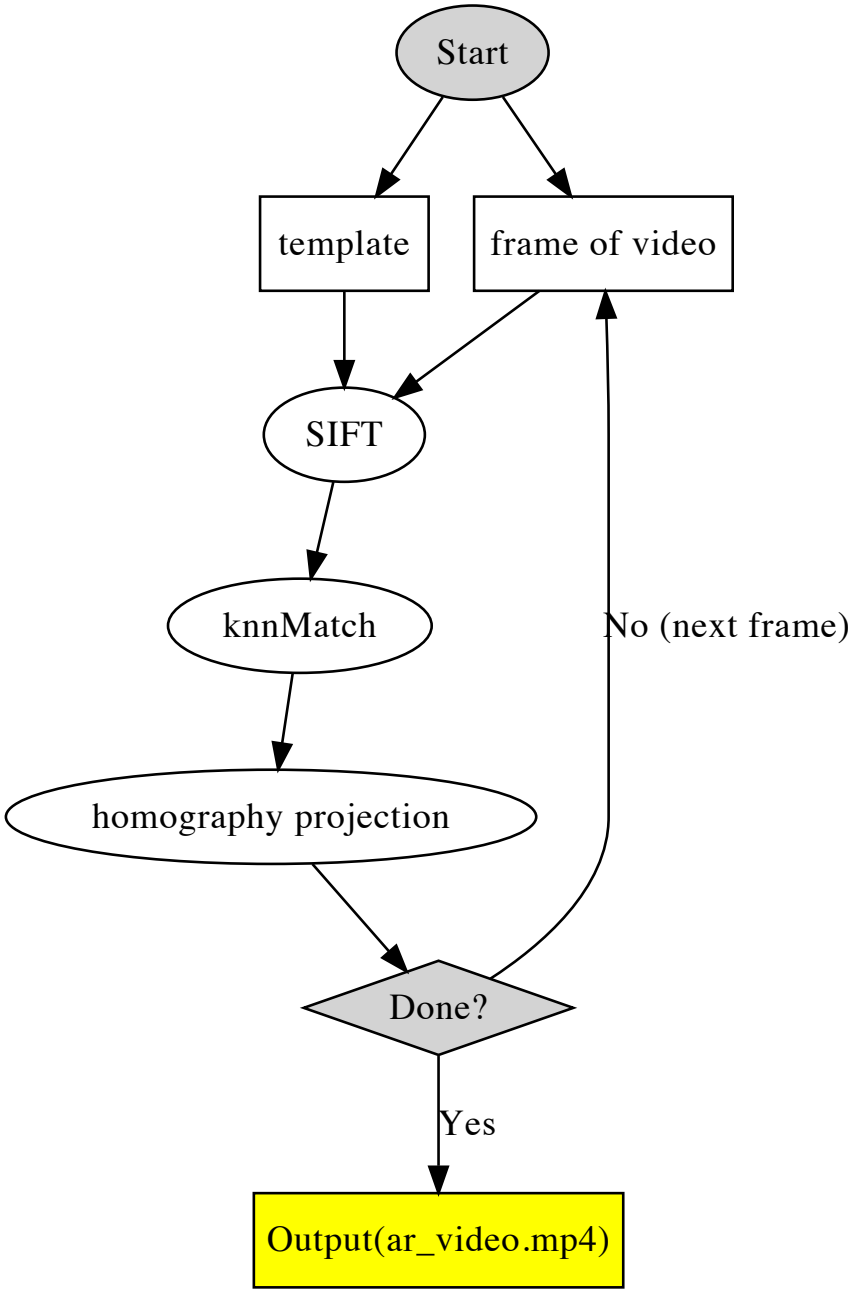
Part 3: Unwarp the 3D Illusion

source image	image after backward warping
	

- We can not get the parallel bars from the top view perfectly because
  - we don't have enough information to complete this task with only one view
  - the relationship between the bars(**eg. parallelism**) can not be recovered
  - the distance between two points is hard to maintained

Part 4: Simple AR

- We can implement a simple augmented reality(AR) by using the homography matrix, and it requires a few more complicated steps to make it:
  - First, **SIFT**(Scale-invariant feature transform) is included in my algorithm to detect and retrieve important features of the template image and the frame of the input video. The reason why I use **SIFT** instead of other feature extraction algorithm is that it provides more accurate performance of detecting interest points even through the efficiency be slightly sacrificed.
  - After detecting the descriptors, we need a matcher which calculates the distance between the template image and the target image and match the interest points. I employ **cv2.FlannBasedMatcher**(faster than **BFMatcher**) and its **knnMatch** to find top k closest descriptors that match the template descriptor.
  - If there is a good match between the template and the frame, we can easily compute the homography matrix and use it to project the reference image to the frame of the input video.
- The flow of this part is shown below:


Start
template
frame of video
SIFT
knnMatch
homography projection
Done?
No (next frame)
Yes
Output(ar_video.mp4)