Computer Vision HW3　Projective Geometry

電機四 B04901096 蔡昕宇

Part 1: Estimating Homography

I implemented the first solution for computing homography matrix *H*.

The code of the function: solve_homography(u, v) is included on the right side.

```
    print('u and v should have the same size')
    return None
if N < 4:
    print('At least 4 points should be given')
A = np.array([[u[0][0], u[0][1], 1, 0, 0, 0, -u[0][0] * v[0][0], -u[0][1] * v[0][0]],
              [0, 0, 0, u[0][0], u[0][1], 1, -u[0][0] * v[0][1], -u[0][1] * v[0][1]],
              [u[1][0], u[1][1], 1, 0, 0, 0, -u[1][0] * v[1][0], -u[1][1] * v[1][0]],
              [0, 0, 0, u[1][0], u[1][1], 1, -u[1][0] * v[1][1], -u[1][1] * v[1][1]],
              [u[2][0], u[2][1], 1, 0, 0, 0, -u[2][0] * v[2][0], -u[2][1] * v[2][0]],
              [0, 0, 0, u[2][0], u[2][1], 1, -u[2][0] * v[2][1], -u[2][1] * v[2][1]],
              [u[3][0], u[3][1], 1, 0, 0, 0, -u[3][0] * v[3][0], -u[3][1] * v[3][0]],
              [0, 0, 0, u[3][0], u[3][1], 1, -u[3][0] * v[3][1], -u[3][1] * v[3][1]]])
b = np.array([[v[0][0]],
              [v[0][1]],
              [v[1][0]],
              [v[1][1]],
              [v[2][0]],
              [v[2][1]],
              [v[3][0]],
              [v[3][1]]])
h = np.dot(np.linalg.inv(A), b)
H = np.array([[h[0, 0], h[1, 0], h[2, 0]],
              [h[3, 0], h[4, 0], h[5, 0]],
              [h[6, 0], h[7, 0], 1]])
return H
```

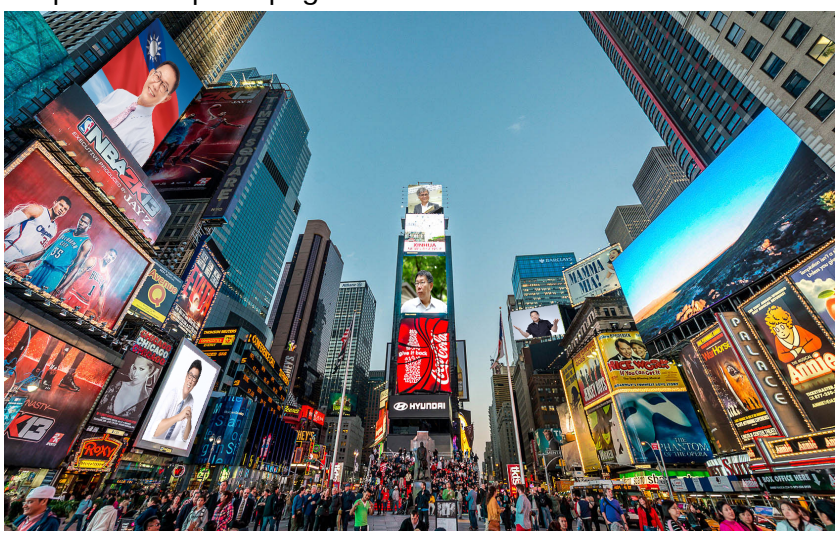
Background image: ./input/times_square.jpg

With 5 images projected in the background image.





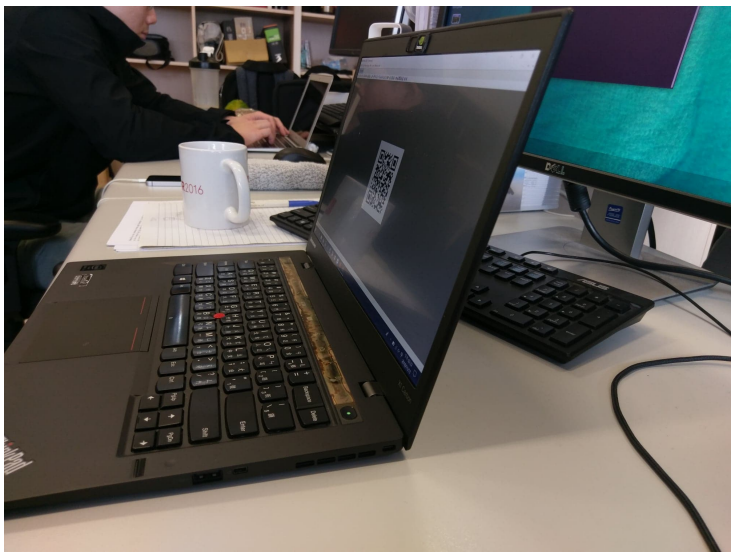
Output result: part1.png



Part 2: Unwarp the screen

Implemented **interpolation** during **backward warping**.

Source image: ./input/screen.jpg



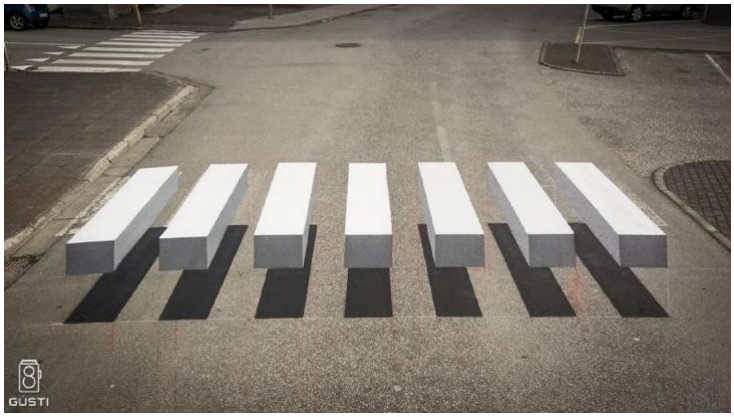
Output Result: part2.png

decoded link: http://media.ee.ntu.edu.tw/courses/cv/18F/

Part 3: Unwarp the 3D illusion

Implemented **backward warping** of the specific area in the image to obtain the top view.

Source image: ./input/crosswalk_front.jpg



Output Result: part3.png

We cannot obtain the result from this procedure, because some other factors will effect the view of imaging. Factors such as

- camera view causes different viewing angle
- distance between two points can be distorted
- relationship between objects cannot be recovered
- information are not sufficient from only one view

Thus, we cannot simply obtain parallel bars from the top view only with the projection matrix.

Bonus: Simple AR

With the help of the homography matrix, we can implement a simple augmented reality (AR). Besides, we need more techniques that help us find some features in the video matching the input image. Some techniques such as **SIFT, SURF, FAST, ORB**, etc can provide us a way to detect important features in the image. Each of the them has their advantages and disadvantages to efficiency and accuracy of detecting the interest points.

SIFT (Scale-Invariant Feature Transform)

The goal of SIFT is to detect interest points in the image. SIFT follows the steps below:

Keypoint Detection: It detects the keypoints by using Difference-of-Gaussian (DOG), convolved with different scales of Gaussian filters.
Orientation Normalization: Calculate local orientation and gradients and count the orientations by sample points near the keypoint.
Keypoint Descriptor: 128 dimensional descriptor from calculating histogram of the orientation.

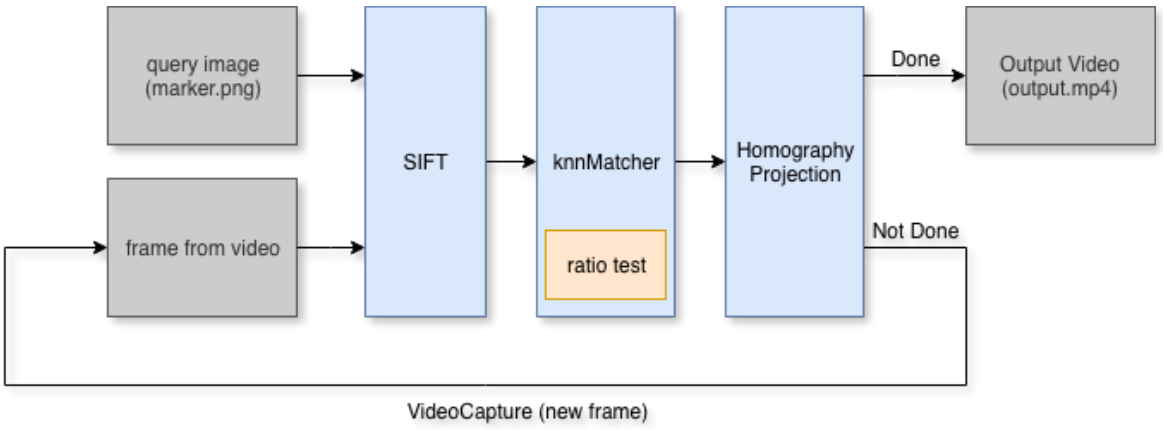
However, SIFT is not the most efficient way to obtain interest points, but the interest points is more accurate. Thus, we only need to compute how close are the descriptors between two images. They are possibly the same point if they are highly related in their descriptors.

BFMatcher.knnMatcher

In order to know which descriptor matches, we need a matcher. Brute force matcher (*BFMatcher*) simply calculates the distance between a descriptor from the query image to all descriptors of the target image, and returns the closest one. *knnMatcher* is to find top k closest descriptors that match the query descriptor.

Overall Algorithm

The overall algorithm flow is shown in the figure below.



First, I applied SIFT algorithm as mentioned above, using *BFMatcher.knnMatcher* for matching. I chose **k = 2** for the *knnMatcher* and applied a ratio test[1] as below.

$$D_{query} \begin{cases} keep & ,if\ d(D^m_{frame}, D_{query}) < 0.85 \times d(D^n_{frame}, D_{query}) \\ discard & ,if\ d(D^m_{frame}, D_{query}) \geq 0.85 \times d(D^n_{frame}, D_{query}) \end{cases}$$

D represents the descriptor of the query image or the frame, and d represents the distance between the descriptor. *m is the most closest descriptor and n* is the second one. If the distance of *m* is larger than 0.85 of the distance of *n*, it seems that the descriptor *m* is not that unique, so we eliminated it for more reasonable projection. After detecting the interest points, we can compute the homography projection matrix based on these interest points.

Output video link: https://www.dropbox.com/s/kqnaiyu2xqgn84e/output_kp.mp4?dl=0

---

[1] Reference: Feature Matching, https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_feature_homography/py_feature_homography.html