Marker-based Augmented Reality model report

How to run the project

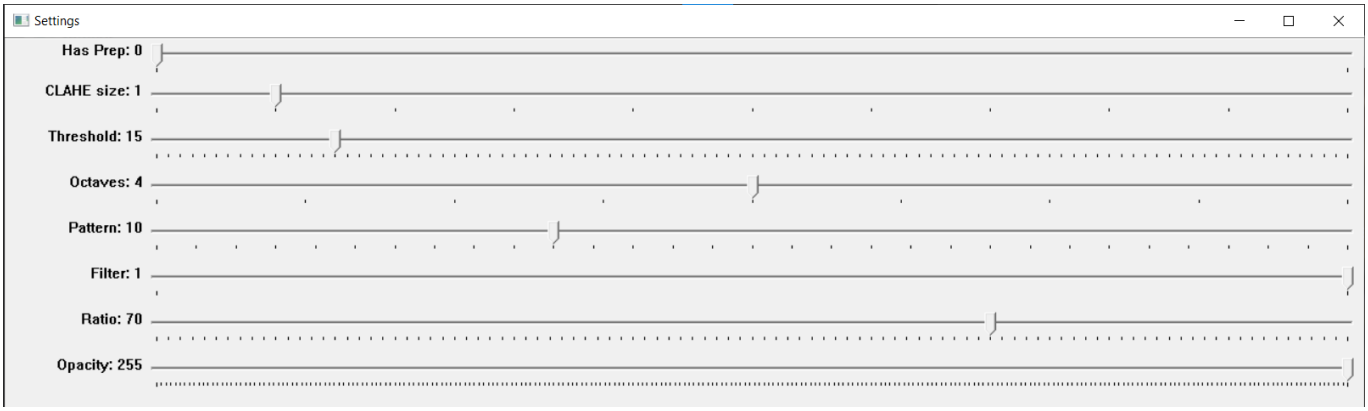
This project use **opencv-4.4.0** and **opencv_contrib-4.4.0** to execute it. In order to build it, you should modify the corresponding option of project's property.

An optimal solution executable is ready, in order to run it:

1. go to the **Release** folder: cd MarkerBasedAugmentedReality/x64/Release
2. put into this folder the sample image and target image
3. run with following syntax: ./MarkerBasedAugmentedReality.exe -i [sample_image_name] -i [target_image_name]

There are several trackbars for the optimal solution:

1. **Has Prep**: whether to do sample image preprocessing(CLAHE)
2. **CLAHE Size**: CLAHE tile grid size, should be greater than 1
3. **Threshold**: BRISK's threshold for keypoints's detection
4. **Octaves**: BRISK's octaves number
5. **Pattern**: BRISK's pattern scale, should be greater than 1
6. **Filter**: 0 for **Score Matching**, 1 for **KNN Matching**
7. **Ratio**: ratio for matching filter
8. **Opacity**: augmentation effect's opacity



If you want to build the solution with full parameters with possibility of changing the feature descriptor with trackbars, you need to switch on the **TEST_MODE** in code and build it. In this case, the program may crash down because some descriptors are not working with some matchers.

Problem statement

In this project, I need to design and implement an OpenCV solution. The solution relies on keypoint-based image descriptor to match between the image of a structured planar marker and its instance in webcam stream. In order to have a optimal solution under webcam stream utilization, there are several factors to consider about.

1. **Computation Time** This factor is the most important one because in the solution, the detection and matching task should be finished less than 1 frame. The minimum value should be 30 frames per second so that the streaming is fluent enough, that's is to say, the maximum computation time should be less than 0.03 second for each frame.
2. **Robustness** A robust solution is that the detecting and matching result is generally good for any kind of image inputs. In this project, a robust solution should not have much jittering when the instance is stable.
3. **Rotation & Scale Invariance** It is normal that the instance in webcam will be rotated or shown in different size. Therefore, a feature detector that has rotational variance or scale variance should not be considered.
4. **Light Condition** Sometimes the situation that the lighting is not ideal will happen, no matter for sample image or for webcam usage.
5. **Motion Blur** A common situation for webcam is that the object will move slowly or fast sometimes. The quality of detection should be considered even the object is moving. In this project, I choose a hard book cover with rich details and another with much less interest points as objects to test.

I choose **SIFT**, **SURF**, **ORB**, **AKAZE** and **BRISK** as the feature detectors and carry out the experiments. The **Harris Corner Detector**[1] and **Shi-Tomasi Corner Detector**[2] is eliminated because of their scale-variant feature.

Theoretical study

SIFT(Scale Invariant Feature Transform)

SIFT[3] is scale-invariant, which is also good at handling illumination change as well. It uses DoG as an approximation of LoG to calculate local extrema(potential keypoint) over scale[4]. SIFT provides distinctiveness, robustness and invariance to common image transformations such as rotation and scale[5]. It is widely accepted as one of highest quality options currently available, promising distinctiveness and invariance to a variety of common image transformations – however, at the expense of computational cost[6].

The parameters contain **nfeatures** which is the number of best features to retain, **nOctaveLayer** which is the number of layers in each octave, **contrastThreshold** that is used to filter out weak features, **edgeThreshold** that filters out edge-like features, and **sigma** that denotes the sigma of the Gaussian applied to the input image. Lowe has given the optimal value for these values in his paper.

SURF(Speeded-Up Robust Features)

SURF[7] approximates LoG with Box Filter convolution calculation, which is faster and is suitable for real-time detecting task. SURF is good at handling images with blurring and rotation, but not good at handling viewpoint change and illumination change.[8] Its implementation is in opencv_contrib repository, in non-free part. Its parameters contain **hessianThreshold** that denotes the threshold for hessian keypoint detector used

in SURF, **nOctaves** and **nOctaveLayers** like that of **SIFT**, as well as two flag **extended** and **upright** which denote using extended descriptor and up-right or rotated features respectively.

ORB(Oriented FAST and Rotated BRIEF)

ORB[9] is a binary feature descriptor that came from *OpenCV Lab* and is basically a fusion of FAST[10] keypoint detector and BRIEF[11] descriptor with many modifications to enhance the performance and scale as well as rotational invariance[12]. It employs the efficient Hamming distance metric for matching and these features make it suitable for real-time feature detection and description.

One of its parameters is *nFeatures* which denotes maximum number of features to be retained.

AKAZE

AKAZE[13] is the accelerated version of KAZE[14] which uses non-linear scale space to find features. It uses a binary descriptor that exploits gradient information. It is proved to be an improvement in repeatability and distinctiveness compared to previous algorithms.

BRISK(Binary Robust Invariant Scalable Keypoints)

BRISK[6] is another binary feature descriptor. It is different from the descriptors like BRIEF and ORB, by having a hand-crafted sampling pattern. BRISK sampling pattern is composed out of concentric rings. In sampling pattern, long pairs are used in BRISK to determine orientation and short pairs are used for the intensity comparisons that build the descriptor, as in BRIEF and ORB[15]. The parameters contains **thresh** which is **AGAST** detection threshold score, **octaves** that is detection octaves and **patternScale**.

Brute Force Matcher

Brute-Force matcher takes the descriptor of one feature in first set and is matched with all other features in second set using some distance calculation. And the closest one is returned[16]. The two parameters possible when creating pointer is **normType**, which defines the computation space. The second parameter, **crossCheck**, is a boolean and its default value is false. If set true, it will check if the pairs in each detector is the same point.

FLANN-based Matcher

FLANN stands for Fast Library for Approximate Nearest Neighbors. It contains a collection of algorithms optimized for fast nearest neighbor search in large datasets and for high dimensional features. It works faster than BFMatcher for large datasets[17].

Matching methods & filtering methods

Each matcher has **match()** and **knnMatch()** to do keypoint matching. The former will find the best match for each descriptor from a query set or the sample image. **knnMatch()** will return the best k matches for each descriptor.

There will be false matching and the next step is to filter the matches. For **knnMatch()**, Lowe proposed in his paper[3] to use a distance ratio test to try to eliminate false matches. The distance ratio between the two nearest matches of a considered keypoint is computed and it is a good match when this value is below a

threshold, and this value is recommended to be 0.7[18]. For **match()** we can sort with scores and filter those with lower scores by applying a ratio of matches to retain.

Alignment & Augmentation

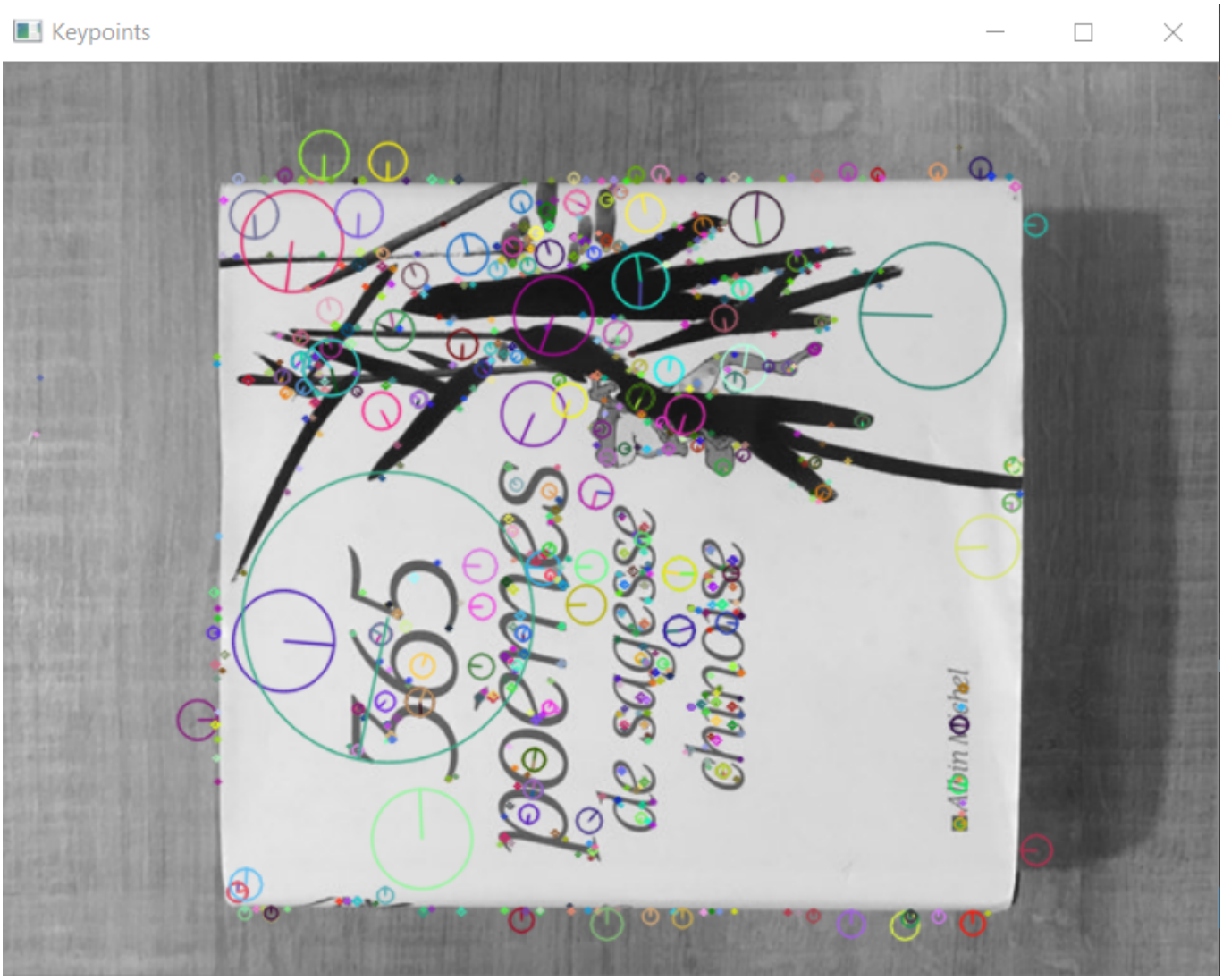
In order to replace the matched image with target image, the perspective transformation of matched image in frame image should be found. In OpenCV, the routine **findHomography()** will compute and return the perspective transformation H between them. We can specify in this method the **RANSAC** robust algorithm to try different random subset in order to estimate the homography matrix[19]. The **warpPerspective()** applies a perspective transformation to an image. By putting the destination image with target image, this can perform image morphing.

Experiment

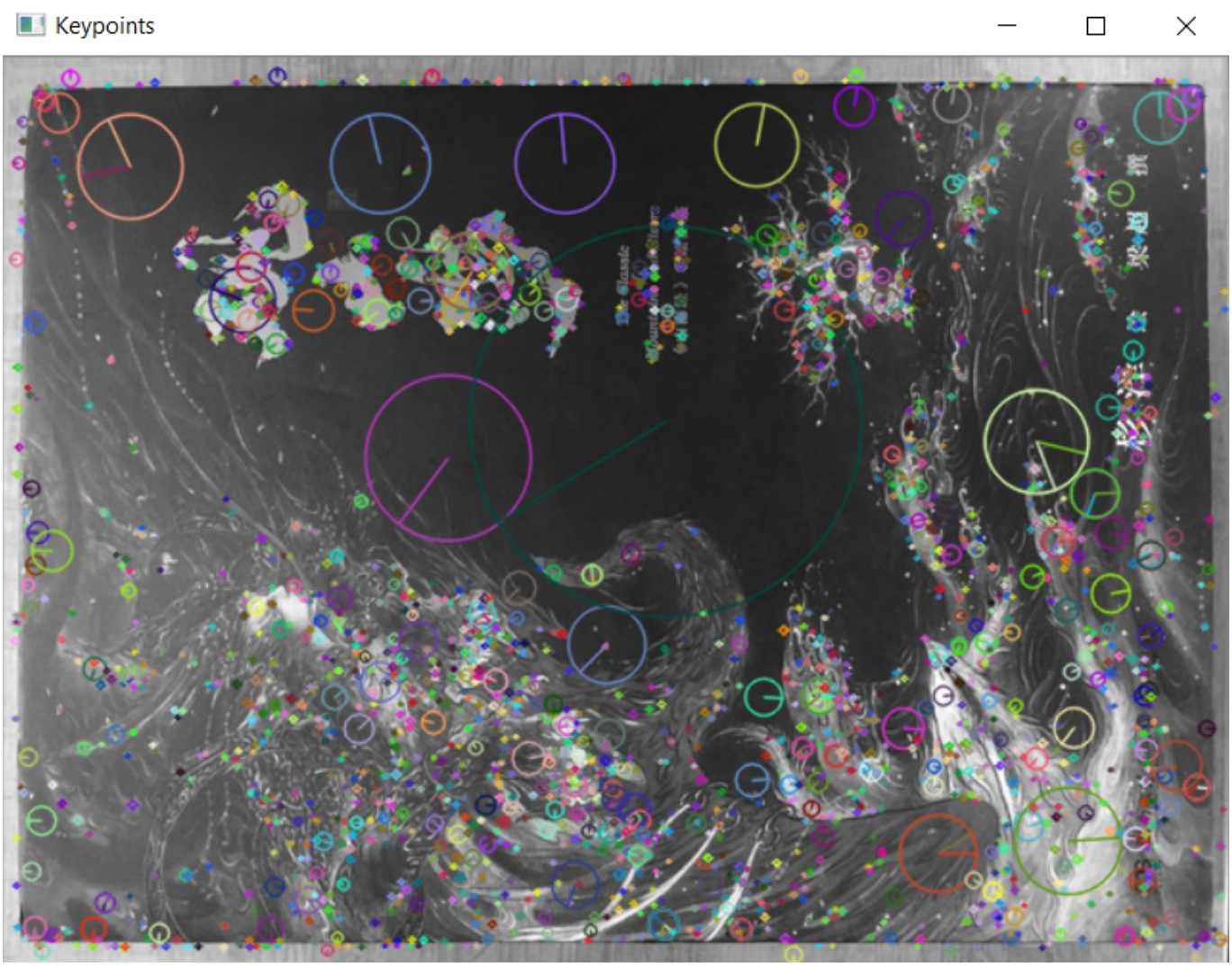
To have a optimal solution, I firstly evaluate the time for detection and matching operations of algorithms above. Each detector pointer is created with default parameters which are supposed to be the optimal ones.

performance

The first experiment is to test the **computational time** of detection for each feature extractor. My CPU is Intel i5-9300H with clockspeed of 2.40 GHz. The calculation time of **detect()** function is tested. In source code you can find the **KPTest()** code implementation. For this test, the variable is the sample image differentiated in lighting and texture(detail).



SIFT keypoints with simple detail sample under light environment



SIFT keypoints with rich detail sample under light environment

```
64/Release$ ./MarkerBasedAugmentedReality.exe -i sample_rich_dark.jpg -i target_h.jpg
Calculating...
surf took me 147 clocks (0.147000 seconds).
keypoints numbers: 862.
sift took me 838 clocks (0.838000 seconds).
keypoints numbers: 510.
orb took me 241 clocks (0.241000 seconds).
keypoints numbers: 500.
akaze took me 880 clocks (0.880000 seconds).
keypoints numbers: 130.
brisk took me 249 clocks (0.249000 seconds).
keypoints numbers: 478.
```

performance comparison with rich detail sample under dark environment

```
64/Release$ ./MarkerBasedAugmentedReality.exe -i sample_simple_dark.jpg -i target_h.jpg
Calculating...
surf took me 268 clocks (0.268000 seconds).
keypoints numbers: 1182.
sift took me 1186 clocks (1.186000 seconds).
keypoints numbers: 899.
orb took me 418 clocks (0.418000 seconds).
keypoints numbers: 500.
akaze took me 1244 clocks (1.244000 seconds).
keypoints numbers: 332.
brisk took me 434 clocks (0.434000 seconds).
keypoints numbers: 697.
```

performance comparison with simple detail sample under dark environment

```
64/Release$ ./MarkerBasedAugmentedReality.exe -i sample_rich_light.jpg -i target_h.jpg
Calculating...
surf took me 155 clocks (0.155000 seconds).
keypoints numbers: 2704.
sift took me 800 clocks (0.800000 seconds).
keypoints numbers: 2276.
orb took me 249 clocks (0.249000 seconds).
keypoints numbers: 500.
akaze took me 843 clocks (0.843000 seconds).
keypoints numbers: 1023.
brisk took me 310 clocks (0.310000 seconds).
keypoints numbers: 3999.
```

performance comparison with rich detail sample under light environment

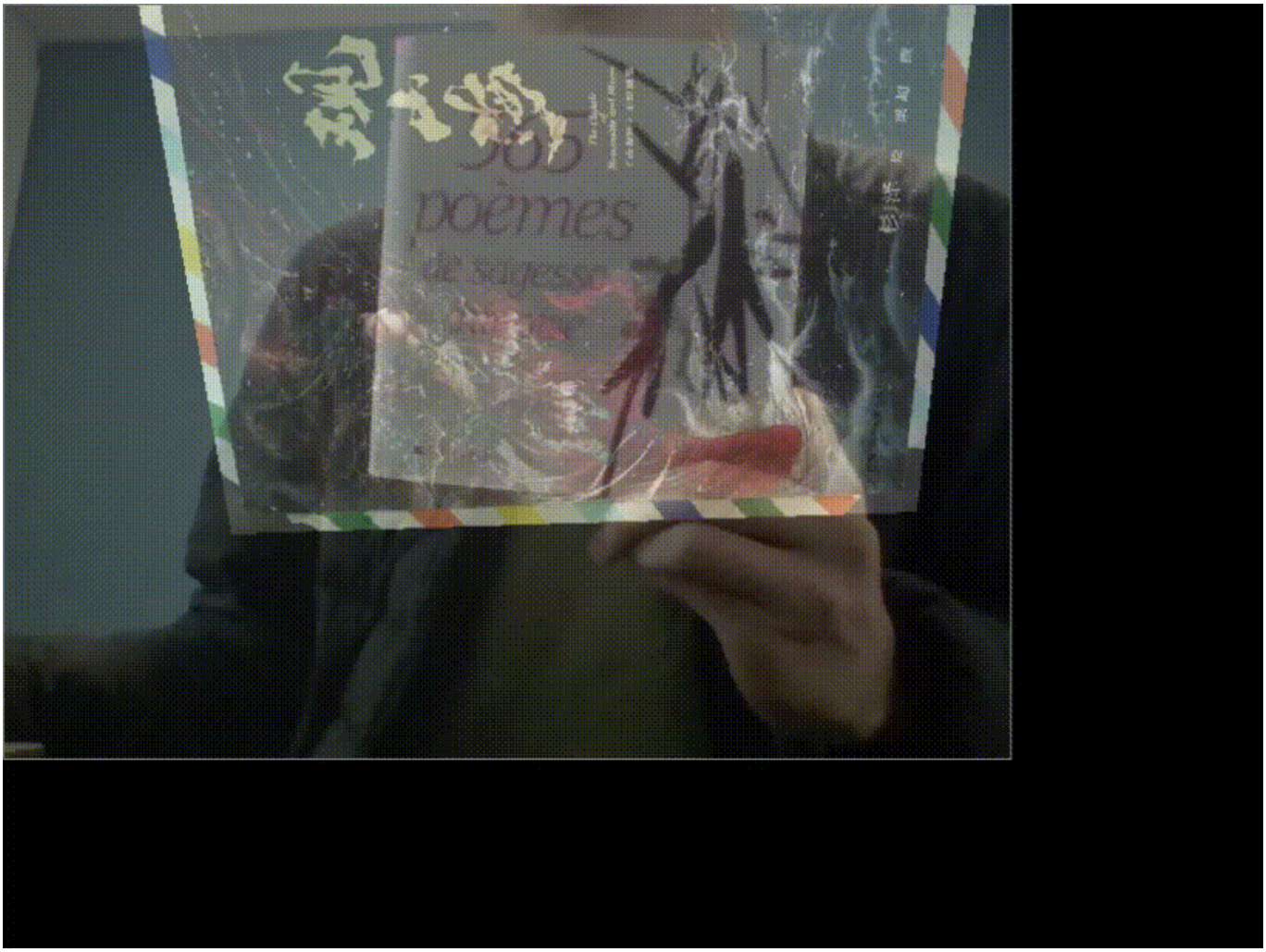
```
64/Release$ ./MarkerBasedAugmentedReality.exe -i sample_simple_light.jpg -i target_h.jpg
Calculating...
surf took me 153 clocks (0.153000 seconds).
keypoints numbers: 1972.
sift took me 853 clocks (0.853000 seconds).
keypoints numbers: 773.
orb took me 244 clocks (0.244000 seconds).
keypoints numbers: 500.
akaze took me 895 clocks (0.895000 seconds).
keypoints numbers: 1027.
brisk took me 263 clocks (0.263000 seconds).
keypoints numbers: 1617.
```

performance comparison with simple detail sample under light environment

From the test above, we can find that the detection time of **SURF**, **ORB** and **BRISK** are at the same level of speed and are much faster than **SIFT** and **AKAZE**. Another observation is that **SURF** find the most keypoints among them.

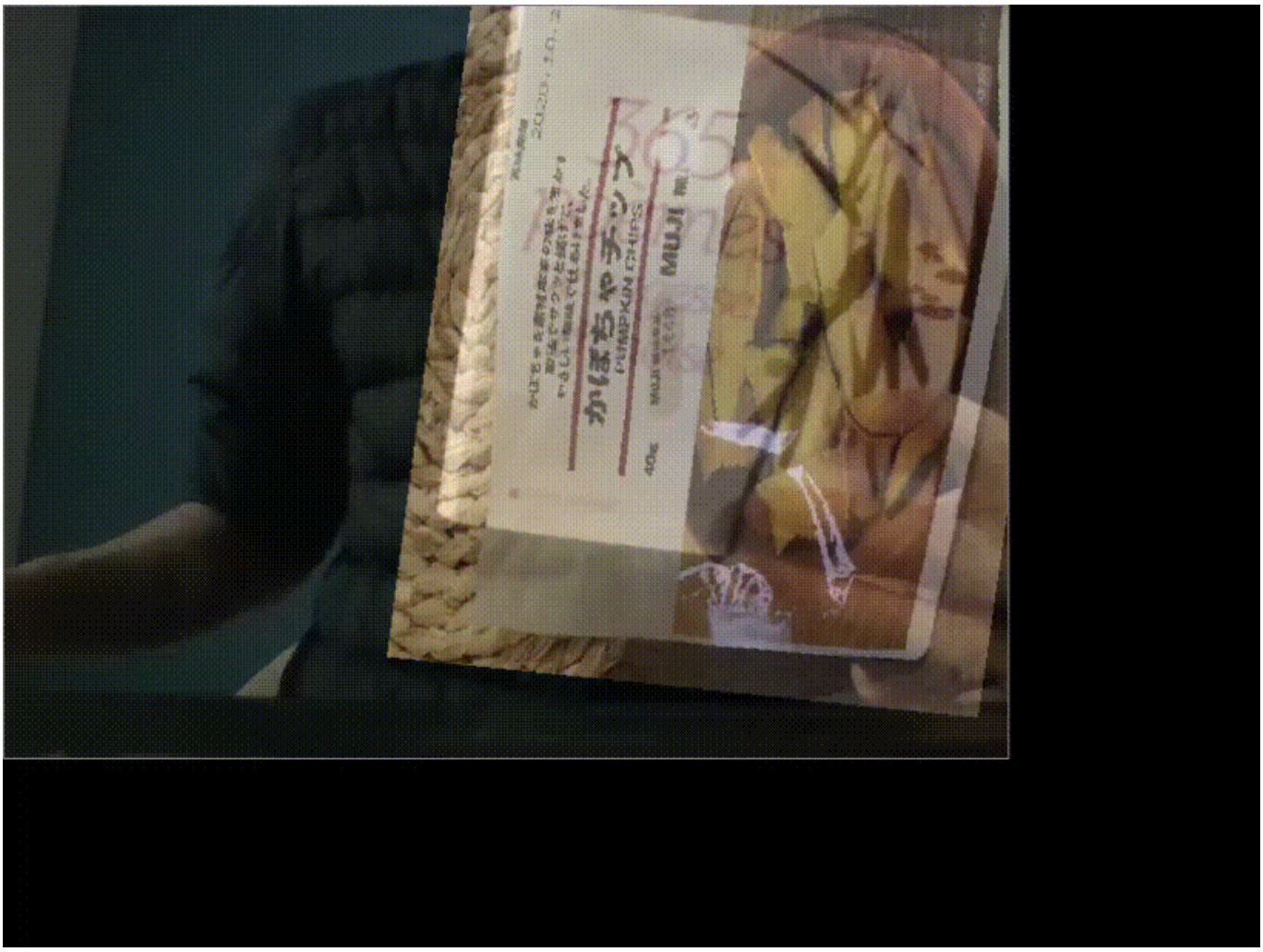
The second test is on matching. For binary string-based feature extractors(ORB, BRISK and AKAZE) I used **BruteForce-Hamming** matcher, and for floating-point ones I used **FLANN** matcher, because it is supposed to be faster than Brute Force based matcher for big dataset. The sample image is the one with simple detail under light environment. The matching filter is **kNN** filtering method where k = 2. The geometric transform is generated by routine **findHomography** with **RANSAC**.

```
AKAZE descriptors matching took me 0 clocks (0.000000 seconds).
The matcher is BruteForce-Hamming.
The matcher filter is KNN filtering.
keypoints1 numbers: 332.
keypoints2 numbers: 298.
matching numbers: 52.
matching rate: 0.174497.
average rate: 0.164124
```



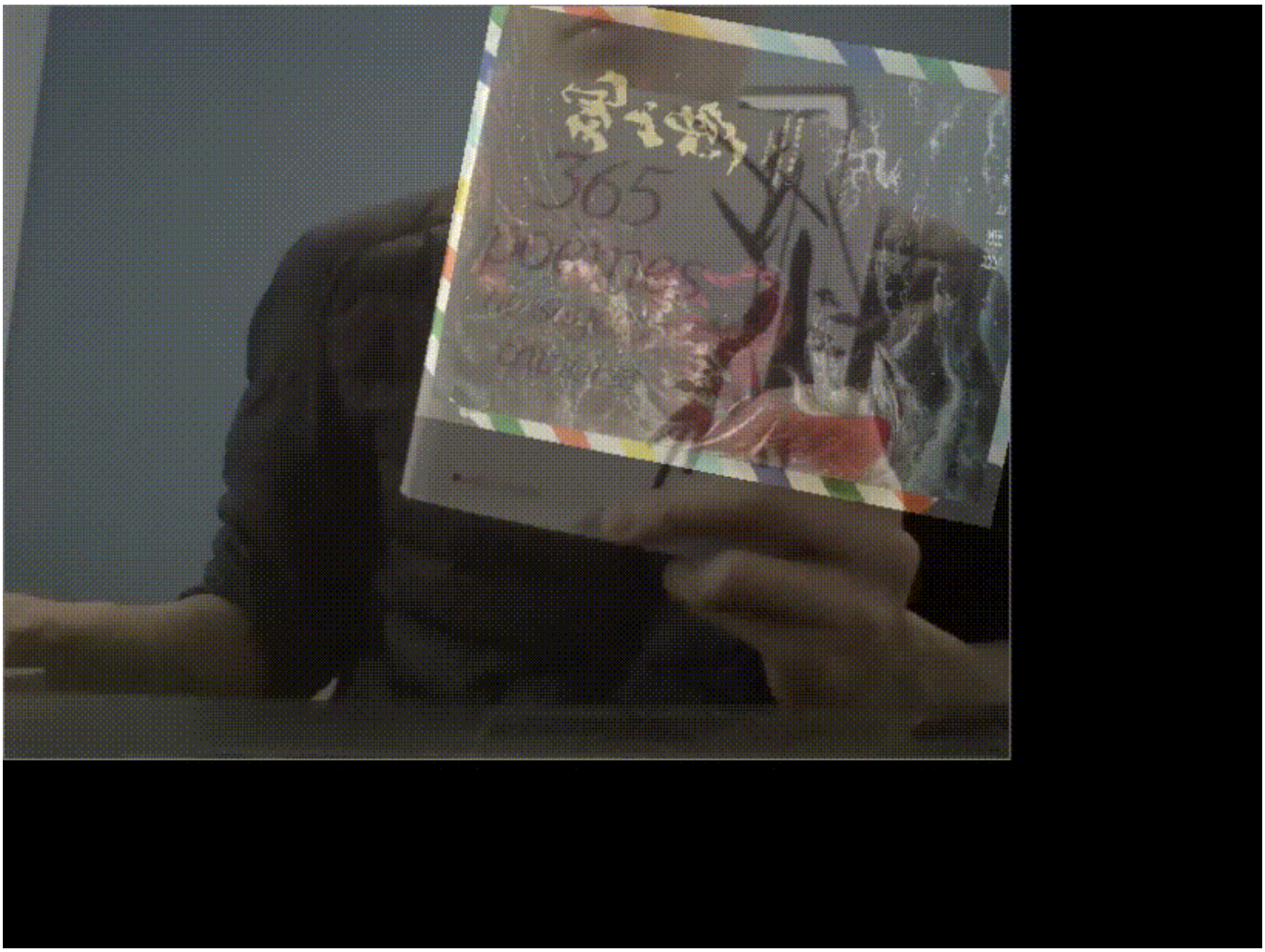
AKAZE performance with default parameters

```
BRISK descriptors matching took me 1 clocks (0.001000 seconds).
The matcher is BruteForce-Hamming.
The matcher filter is KNN filtering.
keypoints1 numbers: 697.
keypoints2 numbers: 365.
matching numbers: 56.
matching rate: 0.153425.
average rate: 0.123764
```



BRISK performance with default parameters

```
ORB descriptors matching took me 1 clocks (0.001000 seconds).
The matcher is BruteForce-Hamming.
The matcher filter is KNN filtering.
keypoints1 numbers: 500.
keypoints2 numbers: 500.
matching numbers: 38.
matching rate: 0.076000.
average rate: 0.072396
```

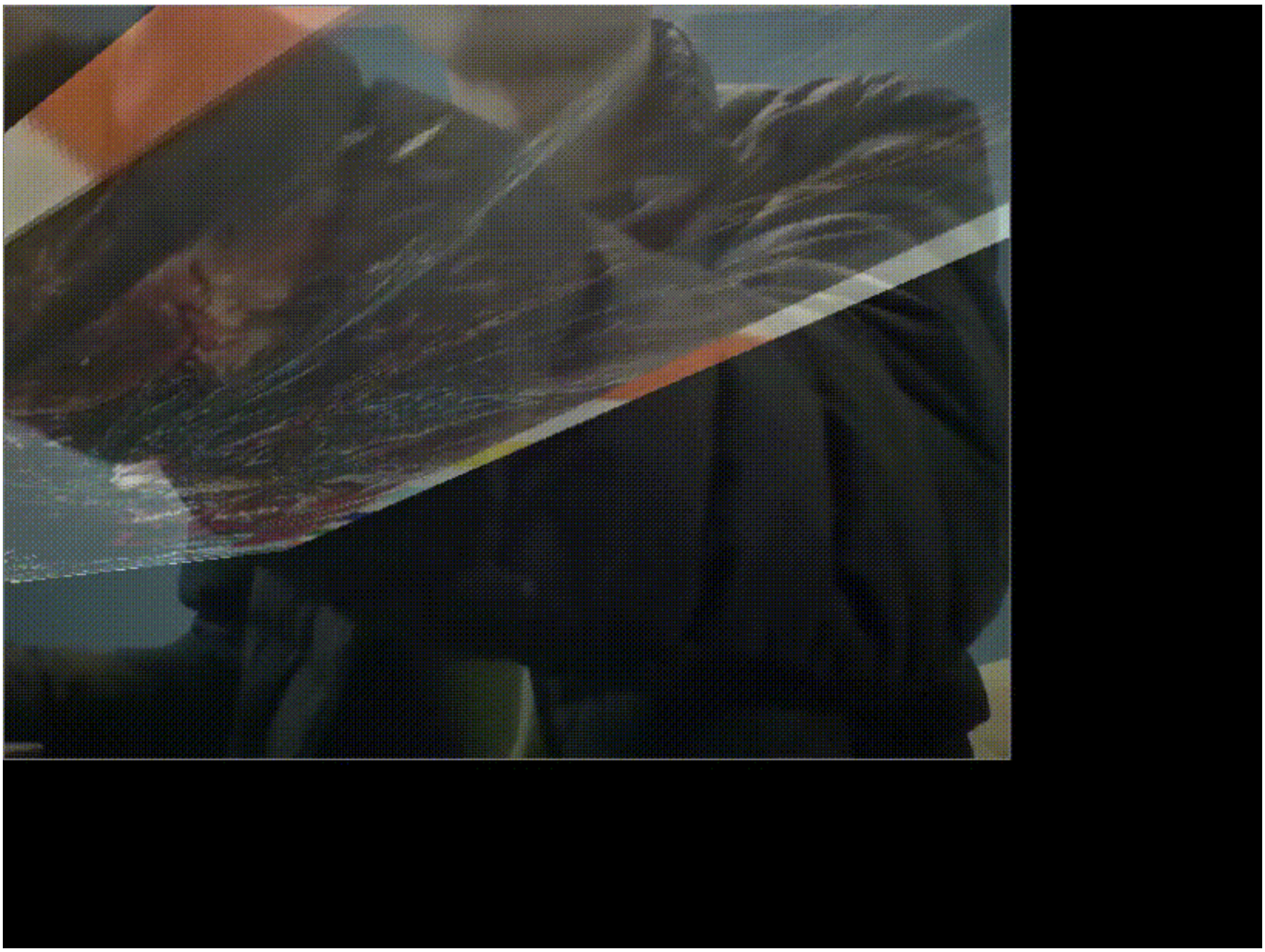


ORB performance with default parameters

```
SIFT descriptors matching took me 13 clocks (0.013000 seconds).
The matcher is FLANN based.
The matcher filter is KNN filtering.
keypoints1 numbers: 899.
keypoints2 numbers: 552.
matching numbers: 185.
matching rate: 0.335145.
average rate: 0.300418
```

SIFT performance with default parameters

```
SURF descriptors matching took me 19 clocks (0.019000 seconds).
The matcher is FLANN based.
The matcher filter is KNN filtering.
keypoints1 numbers: 1182.
keypoints2 numbers: 1223.
matching numbers: 247.
matching rate: 0.201962.
average rate: 0.190617
```



SURF performance with default parameters

From this test we can find that binary string-based descriptors match much faster than that of floating-point-based ones. From observation, I found that **SIFT** and **SURF** will influence the fluency of webcam streaming. However, the matching quality(matching rate) is also better. For binary string-based extractors, the **BRISK** and **AKAZE** are equally better than **ORB** in terms of stability. They all have the problem of jittering when moving the instance quickly. With **ORB**, the matching output will jitter even I hold the book in hand without motion. After this experiment, I decide:

1. Abandon **ORB** because of jittering.
2. Test with parameters of **SIFT** and **SURF** to make the amount of matching less, in order to have less calculation time. According to theory, **SURF** should be three times faster than **SIFT**.
3. Try to improve the matching quality to make sure that when I hold nothing, it will not do wrong matching.

Rotation & scale

The next experiment is about rotational invariance and scale invariance. At a closer distance, the rotation will not influence much the result of all the descriptors. However, when I move the object away, their behavior differs. For **SIFT** it is almost the same, but for **SURF** it is not as robust as before, with some jittering and wrong

matching. **BRISK** and **AZAKE** is not as good as before. But **BRISK** is more robust than **AKAZE** without much wrong matching.

Varying parameters of descriptors

At this stage, I have a general idea of the characteristics of each descriptor. I divide them in to two group:

1. SIFT and SURF
2. BRISK and AKAZE

For the 1st group, I try varying their parameters to limit the calculation time to an acceptable value(0.16s). At the same time, they should have similar quality of matching without too much jittering.

I test with **SURF** by varying the value of **hessianThreshold**, ,**nOctaves** and **nOctaveLayers**. By observation, the augmentation of **hessianThreshold** can be useful to augment the matching rate and performance time. However, it suffers from the jittering problem even the object is stable. What's worse is that it cannot detect object, if the object is not faced to camera.

```
SURF descriptors matching took me 191 clocks (0.191000 seconds).
hessian threshold: 100.
nOtaves: 4.
nOtavesLayers: 3.
keypoints1 numbers: 2014.
keypoints2 numbers: 1005.
matching numbers: 178.
matching rate: 0.177114.
average rate: 0.188801
```

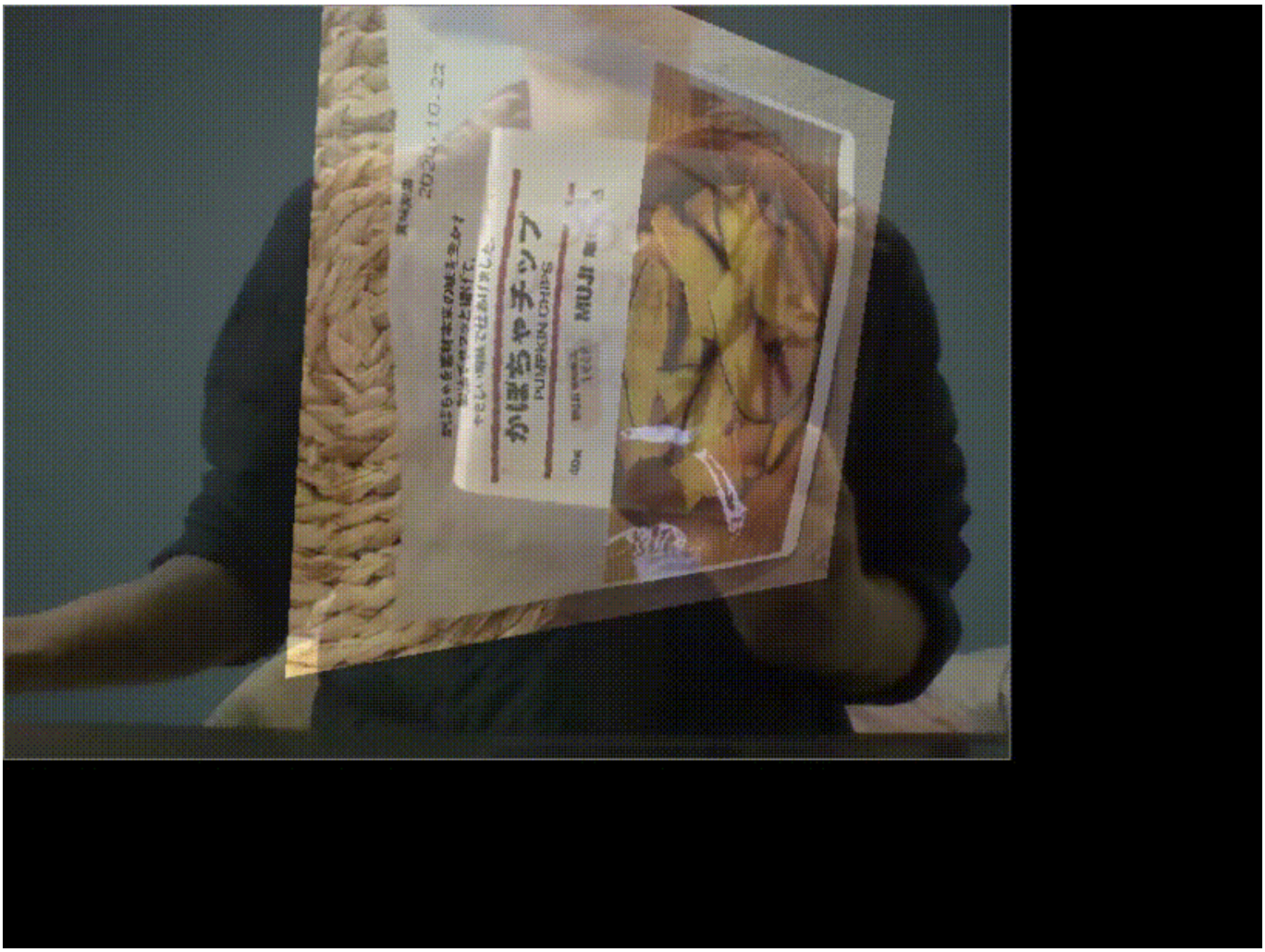
SURF defaut parameters

```
SURF descriptors matching took me 133 clocks (0.133000 seconds).
hessian threshold: 500.
nOtaves: 4.
nOtavesLayers: 3.
keypoints1 numbers: 1286.
keypoints2 numbers: 389.
matching numbers: 112.
matching rate: 0.287918.
average rate: 0.344852
```

```
SURF descriptors matching took me 104 clocks (0.104000 seconds)
hessian threshold: 1200.
nOtaves: 4.
nOtavesLayers: 3.
keypoints1 numbers: 827.
keypoints2 numbers: 104.
matching numbers: 38.
matching rate: 0.365385.
average rate: 0.550666
```

```
SURF descriptors matching took me 84 clocks (0.084000 seconds).
hessian threshold: 1200.
nOtaves: 4.
nOtavesLayers: 2.
keypoints1 numbers: 678.
keypoints2 numbers: 180.
matching numbers: 45.
matching rate: 0.250000.
average rate: 0.322996
```

```
SURF descriptors matching took me 96 clocks (0.096000 seconds).
hessian threshold: 1200.
nOtaves: 3.
nOtavesLayers: 3.
keypoints1 numbers: 818.
keypoints2 numbers: 106.
matching numbers: 48.
matching rate: 0.452830.
average rate: 0.323766
```

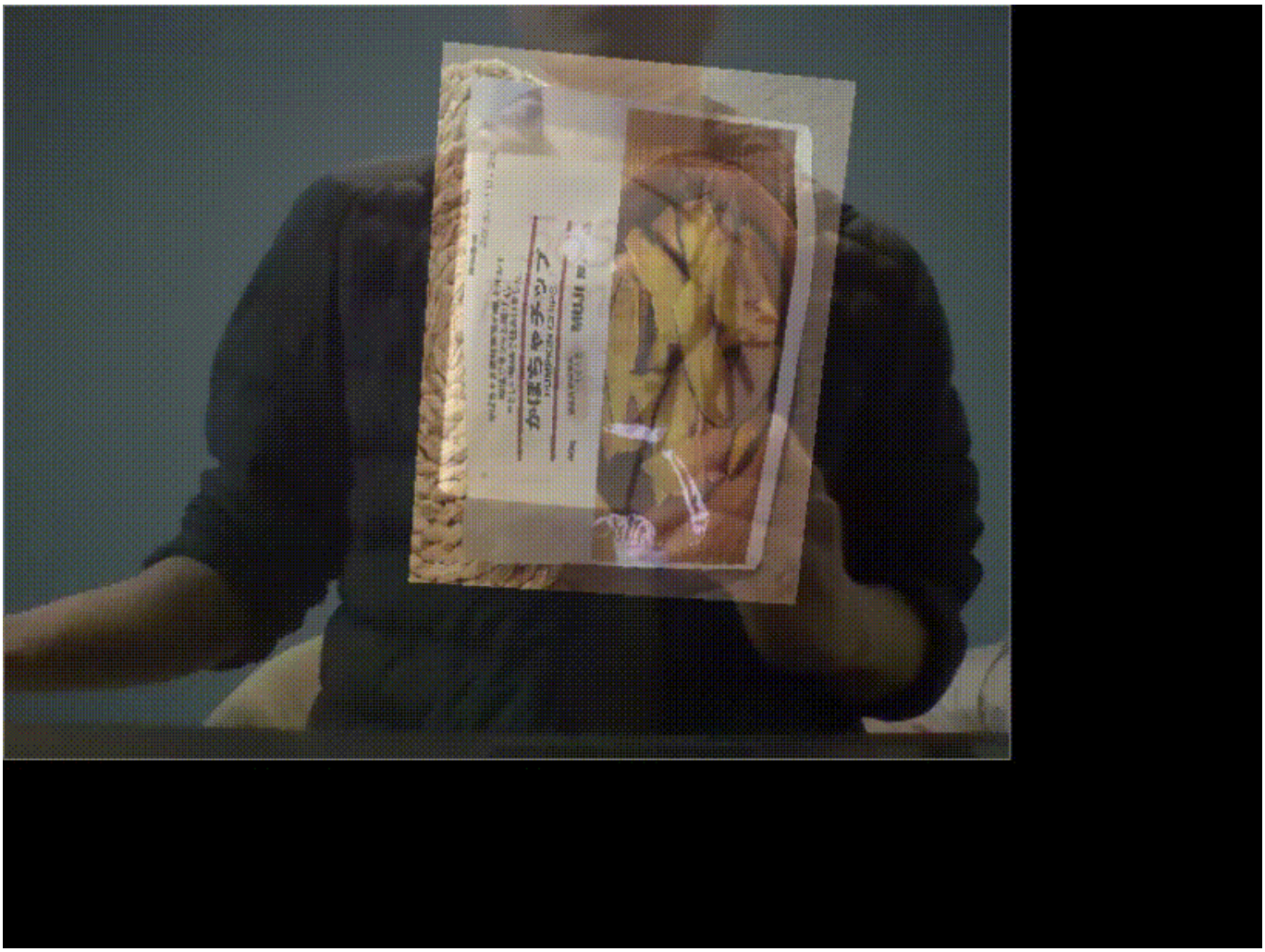


hessianThreshold: 1200, nOctaves: 3, nOctaveLayers: 3

For **SIFT** descriptor, I use **Brute Force L2** matcher because I find that it is faster. There are several parameters for **SIFT**: **nFeatures**, **nOctaveLayers**, **contrastThreshold**, **edgeThreshold** and **sigma**. By varying them I find that the default value is the most robust choice. The change of threshold will make it a bit faster, but it starts suffering jittering.

```
SIFT descriptors matching took me 128 clocks (0.128000 seconds).
nfeatures: 0.
nOtavesLayers: 3.
contrast threshold: 0.040000.
edge threshold: 10.000000.
sigma: 1.600000.
keypoints1 numbers: 773.
keypoints2 numbers: 416.
matching numbers: 60.
matching rate: 0.144231.
average rate: 0.167851
```

SIFT defaut parameters



```
SIFT descriptors matching took me 122 clocks (0.122000 seconds)
nfeatures: 0.
nOtavesLayers: 3.
contrast threshold: 0.040000.
edge threshold: 10.000000.
sigma: 1.600000.
keypoints1 numbers: 1253.
keypoints2 numbers: 216.
matching numbers: 53.
matching rate: 0.245370.
average rate: 0.189256
```

```
SIFT descriptors matching took me 113 clocks (0.113000 seconds).
nfeatures: 0.
nOtavesLayers: 3.
contrast threshold: 0.080000.
edge threshold: 10.000000.
sigma: 1.600000.
keypoints1 numbers: 620.
keypoints2 numbers: 145.
matching numbers: 22.
matching rate: 0.151724.
average rate: 0.224371
```

By observation, I find that the computation time of **SIFT** is similar to **SURF**, while with a better quality, so I decide to abandon **SURF** for the solution.

For the second group of **BRISK** and **AZAKE**, I try improving their detection and matching quality by varying their parameters.

For **BRISK**, its parameters are **threshold**, **octaves** and **patternScale**. While decreasing **threshold** or increasing **octaves**, the keypoints detected increase which make the matching quality better. Even though the computation time increase to about 0.1s, but for experience it is unrecognizable. With modified parameters, **BRISK** can achieve a stable matching even though the object is not faced to camera at all.

```
BRISK descriptors matching took me 47 clocks (0.047000 seconds).
threshold: 30.
Otaves: 3.
pattern scale: 1.000000.
keypoints1 numbers: 1617.
keypoints2 numbers: 434.
matching numbers: 35.
matching rate: 0.080645.
average rate: 0.119788
```

BRISK defaut parameters

```
BRISK descriptors matching took me 69 clocks (0.069000 seconds).
threshold: 15.
Otaves: 3.
pattern scale: 1.000000.
keypoints1 numbers: 2115.
keypoints2 numbers: 891.
matching numbers: 137.
matching rate: 0.153760.
average rate: 0.144168
```

```
BRISK descriptors matching took me 87 clocks (0.087000 seconds).
threshold: 15.
Otaves: 4.
pattern scale: 1.000000.
keypoints1 numbers: 2530.
keypoints2 numbers: 786.
matching numbers: 59.
matching rate: 0.075064.
average rate: 0.041261
```

The jittering problem of **AKAZE** is much worser than **BRISK**, which is proved by tests above. Therefore, I will not go in deeper for **AKAZE** in the following experiments.

Descriptor Matchers & Matching filters

When doing experiments above, I used Brute-Force Hamming descriptor matcher, so whether it is possible to use other metrics to get a significant improvement ? The answer is no[20]. But for **SIFT** descriptor, there are options like **FLANN** and **Brute-Force**. The normType **BF-L2** are also suitable for **SIFT**.

The performance of **BF-L2** and **FLANN** is also related to the matching methods and matching filters. Each descriptor matcher has **match()** function and **knnMatch()** function, which correspond to best matching and k-nearest-neighbor matching respectively. To filter out the bad matching, they can chooses **Score filtering** and **Ratio filtering** respectively.

Here I would like to compare the performance with different matchers without filtering methods for **SIFT** and **BRISK** firstly.

```
SIFT descriptors matching took me 132 clocks (0.132000 seconds).
current matcher is FLANN based.
keypoints1 numbers: 912.
keypoints2 numbers: 830.
matching numbers: 912.
matching rate: 1.000000.
average rate: 1.000000
```

```
SIFT descriptors matching took me 138 clocks (0.138000 seconds).
current matcher is BruteForce (L2 norm).
cross check: False
keypoints1 numbers: 912.
keypoints2 numbers: 791.
matching numbers: 912.
matching rate: 1.000000.
average rate: 1.000000
```

```
SIFT descriptors matching took me 127 clocks (0.127000 seconds).
current matcher is BruteForce (L2 norm).
cross check: True
keypoints1 numbers: 912.
keypoints2 numbers: 726.
matching numbers: 281.
matching rate: 0.308114.
average rate: 0.312669
```

```
BRISK descriptors matching took me 122 clocks (0.122000 seconds)
current matcher is BruteForce (L2 norm).
cross check: False
keypoints1 numbers: 2530.
keypoints2 numbers: 1684.
matching numbers: 2530.
matching rate: 1.000000.
average rate: 1.000000
```

```
BRISK descriptors matching took me 103 clocks (0.103000 seconds).
current matcher is BruteForce-Hamming.
cross check: False
keypoints1 numbers: 2530.
keypoints2 numbers: 1828.
matching numbers: 2530.
matching rate: 1.000000.
average rate: 1.000000
```

```
BRISK descriptors matching took me 114 clocks (0.114000 seconds).
current matcher is BruteForce-Hamming.
cross check: True
keypoints1 numbers: 2530.
keypoints2 numbers: 1633.
matching numbers: 604.
matching rate: 0.238735.
average rate: 0.237216
```

The result shows that the computation time of **FLANN** and **BF-L2** for **SIFT** is similar, for this sample image. However, with **crossCheck** activated, the matching quality is much better. For **BRISK**, **BF-Hamming** is more quickly than **BF-L2**.

The next step is to combine match functions as well as match filtering to compare.

```
SIFT descriptors matching took me 154 clocks (0.154000 seconds).
current matcher is BruteForce (L2 norm).
cross check: True
matching filter is best matching & Score filtering.
keypoints1 numbers: 912.
keypoints2 numbers: 380.
retained match ratio: 0.700000.
matching numbers: 102.
matching rate: 0.111842.
average rate: 0.194688
```

```
SIFT descriptors matching took me 147 clocks (0.147000 seconds).
current matcher is BruteForce (L2 norm).
cross check: False
matching filter is KNN matching & Ratio filtering.
keypoints1 numbers: 912.
keypoints2 numbers: 718.
retained match ratio: 0.700000.
matching numbers: 155.
matching rate: 0.169956.
average rate: 0.170395
```

```
SIFT descriptors matching took me 129 clocks (0.129000 seconds).
current matcher is FLANN based.
matching filter is best matching & Score filtering.
keypoints1 numbers: 912.
keypoints2 numbers: 630.
retained match ratio: 0.700000.
matching numbers: 638.
matching rate: 0.699561.
average rate: 0.699561
```

```
SIFT descriptors matching took me 136 clocks (0.136000 seconds).
current matcher is FLANN based.
matching filter is KNN matching & Ratio filtering.
keypoints1 numbers: 912.
keypoints2 numbers: 526.
retained match ratio: 0.700000.
matching numbers: 142.
matching rate: 0.155702.
average rate: 0.175953
```

```
BRISK descriptors matching took me 80 clocks (0.080000 seconds).
current matcher is BruteForce-Hamming.
cross check: True
matching filter is best matching & Score filtering.
keypoints1 numbers: 2530.
keypoints2 numbers: 661.
retained match ratio: 0.700000.
matching numbers: 166.
matching rate: 0.065613.
average rate: 0.128487
```

```
BRISK descriptors matching took me 86 clocks (0.086000 seconds).
current matcher is BruteForce-Hamming.
cross check: False
matching filter is KNN matching & Ratio filtering.
keypoints1 numbers: 2530.
keypoints2 numbers: 1392.
retained match ratio: 0.700000.
matching numbers: 165.
matching rate: 0.065217.
average rate: 0.067101
```

When setting ratio to be 0.7, the computation time is similar between four combinations for **SIFT**, which are generally 1 times slower than that of **BRISK**. In this case, I will provide **BRISK** as the feature descriptor for the final solution, based on the fact that it is faster than **SIFT** and that its robustness is similar or even better than **SIFT**. Another fact is that user can still use this solution by adjusting the threshold even they have no a good machine.

Final solution

As final solution, I set **BRISK** as feature descriptor and **BF-Hamming** as descriptor matcher. The parameters are the creation option for pointer to **BRISK** and **Matcher filter** as well as augmentation effect's opacity.

At this moment I would like to do an evaluation for its performance with response to different situations. The default parameters are following which is the optimal parameters that I conclude after several tests:

```
> Keypoints Detector | BRISK
> Threshold          | 15
> Octaves            | 4
> Pattern Scale      | 1
> Descriptor Matcher | BruteForce-Hamming
> Matches Filtering  | KNN matching & Ratio filtering
> Matches Rate       | 0.7
```

The experiment under normal situation will be also recorded and provided as video at the end of this document.

Calculation Time

When holding a book without movement, the calculation time is from 80ms to 90ms for my computer with CPU Intel i5-9300H with clockspeed of 2.40 GHz, under a good lighting situation and a simple sample as well as an simple background.

```
BRISK descriptors matching took me 90 clocks (0.090000 seconds).
BRISK descriptors matching took me 88 clocks (0.088000 seconds).
BRISK descriptors matching took me 89 clocks (0.089000 seconds).
BRISK descriptors matching took me 87 clocks (0.087000 seconds).
BRISK descriptors matching took me 86 clocks (0.086000 seconds).
BRISK descriptors matching took me 85 clocks (0.085000 seconds).
BRISK descriptors matching took me 85 clocks (0.085000 seconds).
BRISK descriptors matching took me 90 clocks (0.090000 seconds).
BRISK descriptors matching took me 84 clocks (0.084000 seconds).
BRISK descriptors matching took me 85 clocks (0.085000 seconds).
```

Rotation & Scale

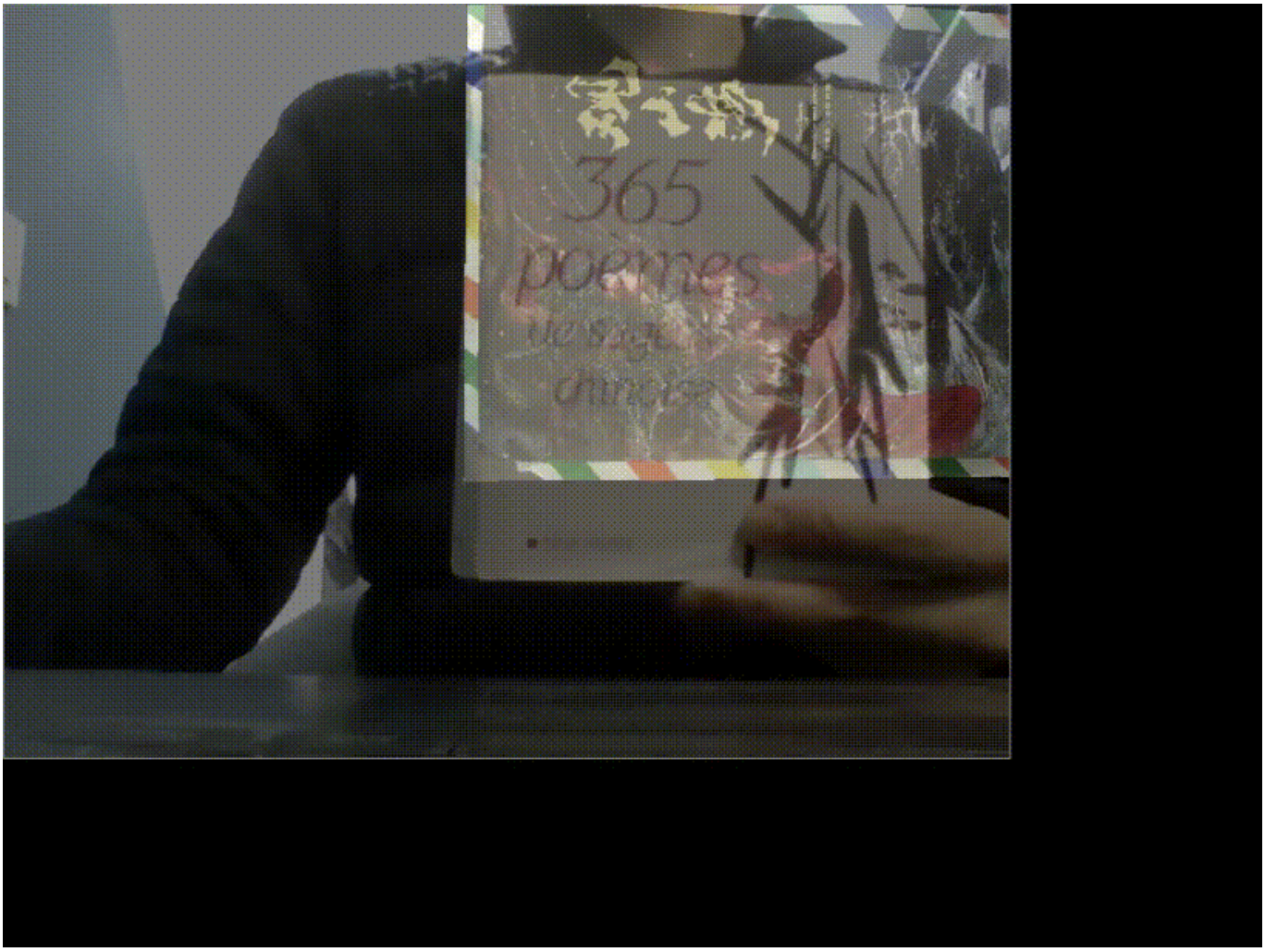
Without rotation, the safe distance from object to camera is about 50cm. From 50 to 70cm, there will be visible jittering. After that the object cannot be matched successfully.

With 180 degree of rotation along the normal direction, the safe distance is about 40cm. From 40 to 80cm, there will be visible jittering. After that the object cannot be matched successfully.

At a distance of 25cm from camera, if the object is rotated along the normal direction to the ground, within 45 degree it is matched successfully without too much jittering. From 45 to 50 degree, it will become unrecognizable.

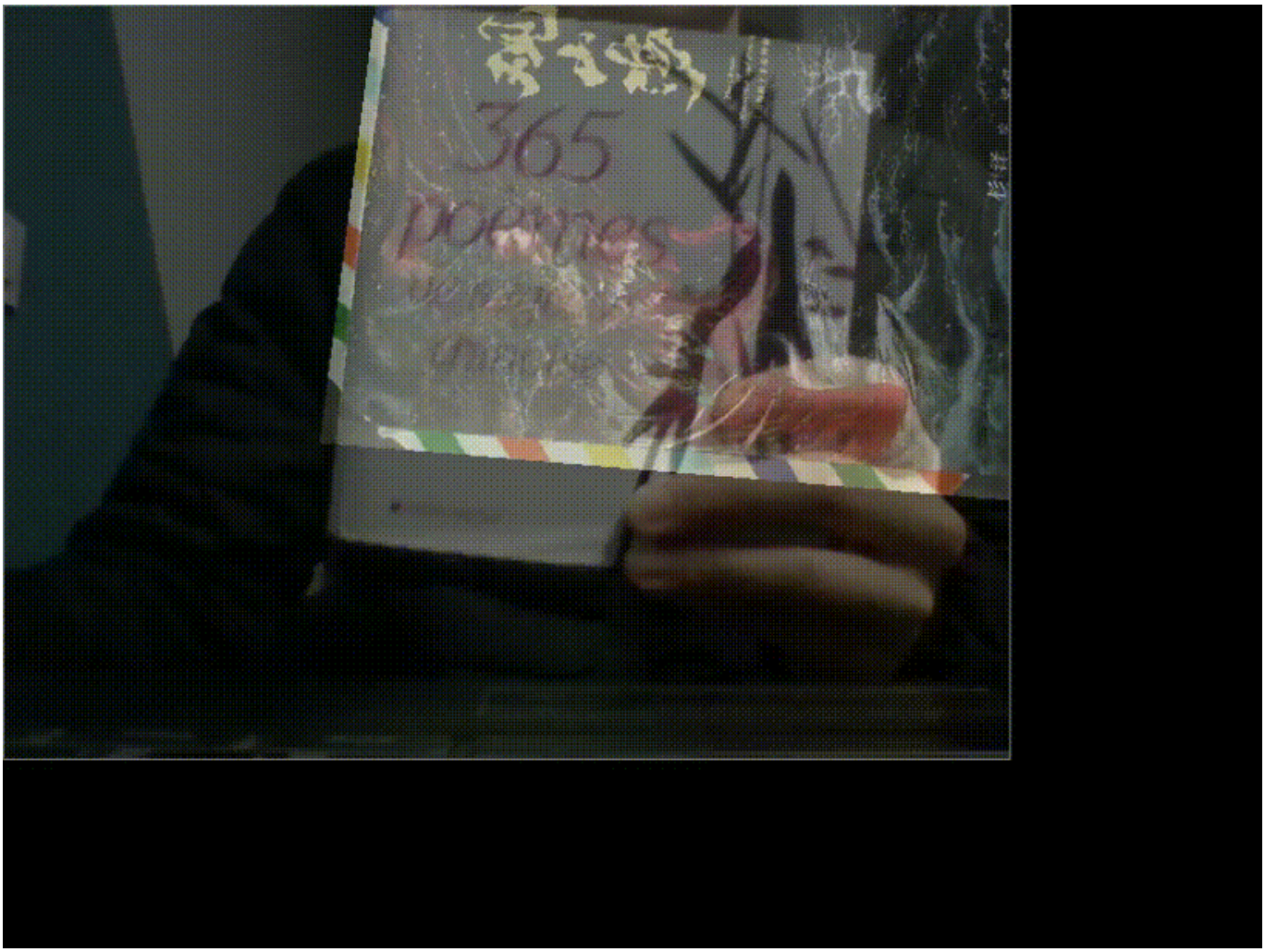
Motion Blur

With quick motion, the matching will not be successful but will be done once the object becomes stable. With slow motion, the matching will not be interrupted.



Lighting

The lighting situation will not influence too much but will result in certain jittering, even there is no light in room, with only screen's light. It also depends on the object itself. If it is black with many details, the matching will be greatly influenced, but is still can be identified.



Background Complexity

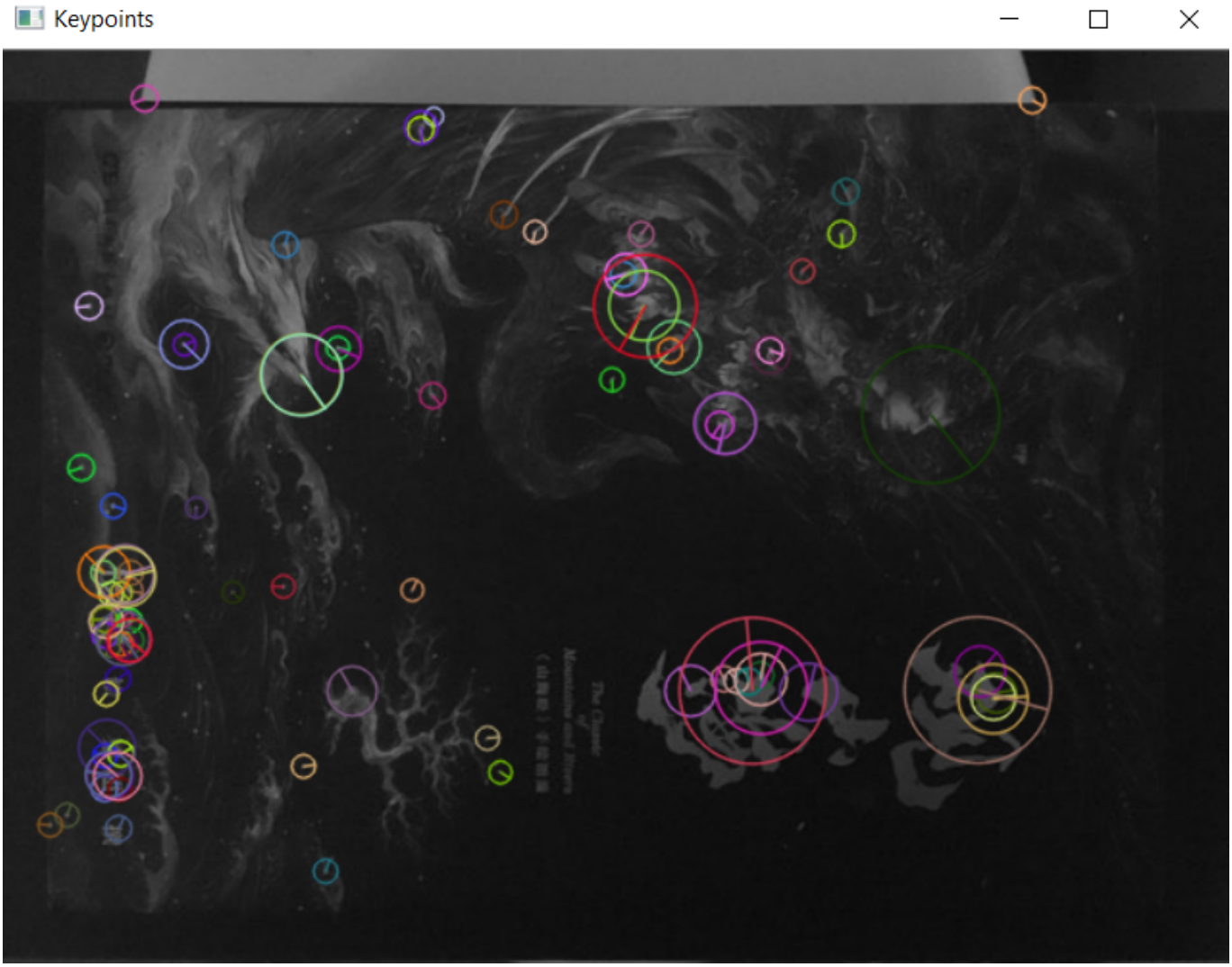
With comparison of putting many objects on background, the matching speed and quality will not be influenced.

Sample Images variation

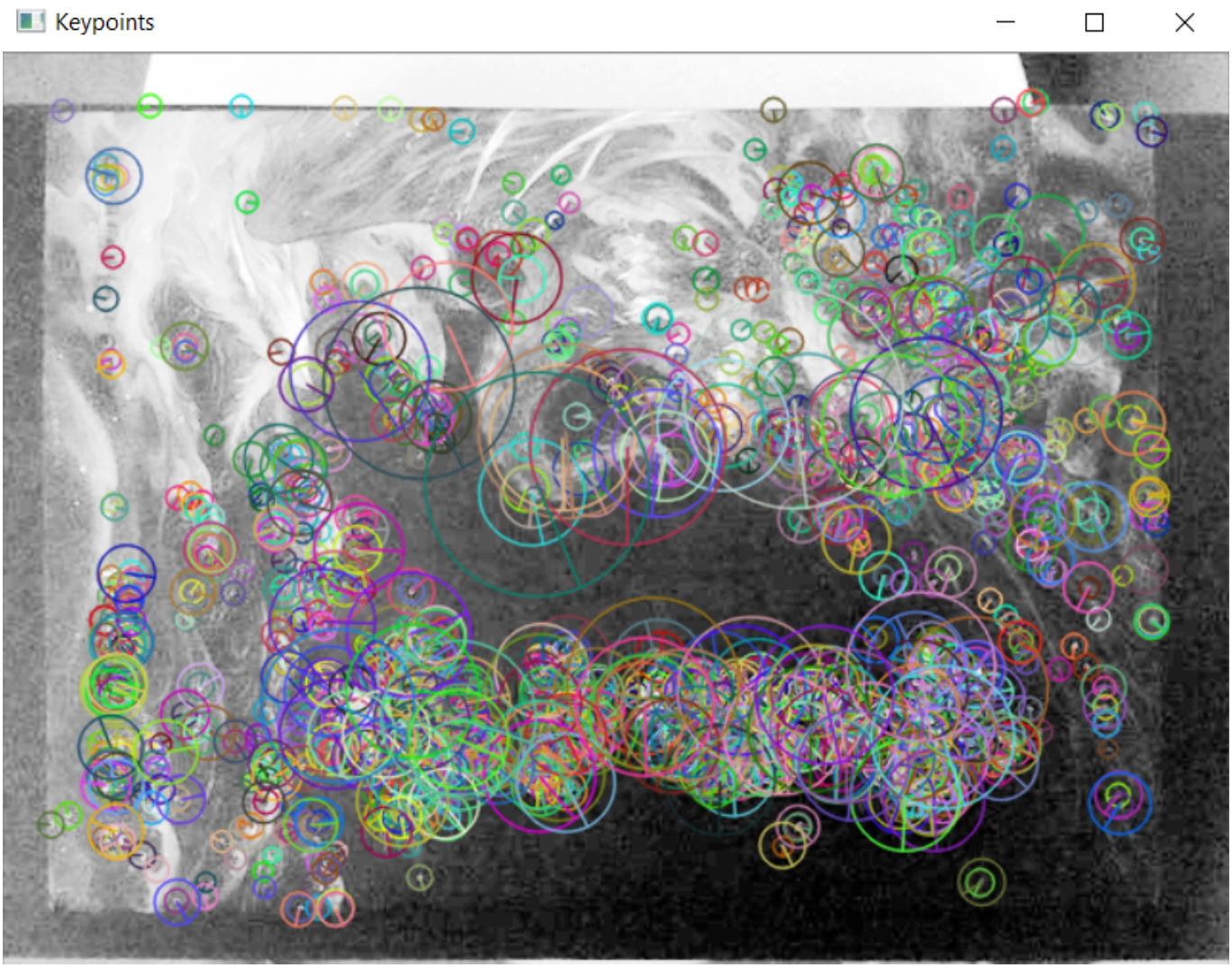
The sample image will influence the performance. By applying an cover with many details, the computation time will even be doubled but the matching quality is still robust.

Preprocessing

Some sample images need to be preprocessed, so I add **CLAHE** pipeline to adjust the contrast of luminosity of image.



without CLAHE preprocessing



with CLAHE preprocessing

I have tested gaussian blurring as preprocessing but it will not help the detecting and matching. In most cases, it will erase the details and make the keypoints much less than original image. So I do not use it for final solution.

limitations

The first limitation is that, when there is no correct target in scene, the solution will still try matching. The second limitation is that, the computation time will vary according to sample image. However, this can be improved by increasing the threshold to make detected keypoints less.

Conclusion

After doing this project, I have gained a better understanding on state-of-the-art descriptors theoretically and practically. I also learned a basic way to carry out a scientific and quantitative way to do benchmark test of algorithms and the way to document it. Finally, I learned how to evaluate a solution's performance.

References

1. OPENCV -- Harris Corner Detection
2. OPENCV -- Shi-Tomasi Corner Detector & Good Features to Track
3. G. D. Lowe. Distinctive image features from scale-invariant keypoints. IJCV, pages 91–110, November 2004
4. OPENCV -- Introduction to SIFT (Scale-Invariant Feature Transform)
5. A. Pieropan, M. Björkman, N. Bergström and D. Kragic, Feature Descriptors for Tracking by Detection: a Benchmark, Jul 2016, online
6. S. Leutenegger, M. Chli and R. Siegwart, "Brisk: Binary robust invariant scalable keypoints", Proc. Int. Conf. Computer Vision, pp. 2548-2555, 2011.
7. Bay, H., Ess, A., Tuytelaars, T., Gool, L.V.: Surf: Speeded Up Robust Features. Computer Vision and Image Understanding 10, 346–359 (2008)
8. OPENCV -- Introduction to SURF (Speeded-Up Robust Features)
9. Ethan Rublee, Vincent Rabaud, Kurt Konolige, Gary R. Bradski: ORB: An efficient alternative to SIFT or SURF. ICCV 2011: 2564-2571.
10. Edward Rosten and Tom Drummond, “Machine learning for high speed corner detection” in 9th European Conference on Computer Vision, vol. 1, 2006, pp. 430–443.
11. Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua, "BRIEF: Binary Robust Independent Elementary Features", 11th European Conference on Computer Vision (ECCV), Heraklion, Crete. LNCS Springer, September 2010.
12. OPENCV -- ORB (Oriented FAST and Rotated BRIEF)
13. KAZE Features. Pablo F. Alcantarilla, Adrien Bartoli and Andrew J. Davison. In European Conference on Computer Vision (ECCV), Fiorenze, Italy, October 2012. bibtex
14. Fast Explicit Diffusion for Accelerated Features in Nonlinear Scale Spaces. Pablo F. Alcantarilla, Jesús Nuevo and Adrien Bartoli. In British Machine Vision Conference (BMVC), Bristol, UK, September 2013. bibtex
15. A tutorial on binary descriptors – part 4 – The BRISK descriptor
16. OPENCV -- Feature Matching
17. OPENCV -- Feature Matching
18. OPENCV -- Feature Matching with FLANN
19. Camera Calibration and 3D Reconstruction
20. E. Bostanci, "Is Hamming distance only way for matching binary image feature descriptors?", Electronics Letters, vol. 50, no. 11, pp. 806-808, May 2014.