A Review on Cinemagraph Auto Compose Using OpenCV

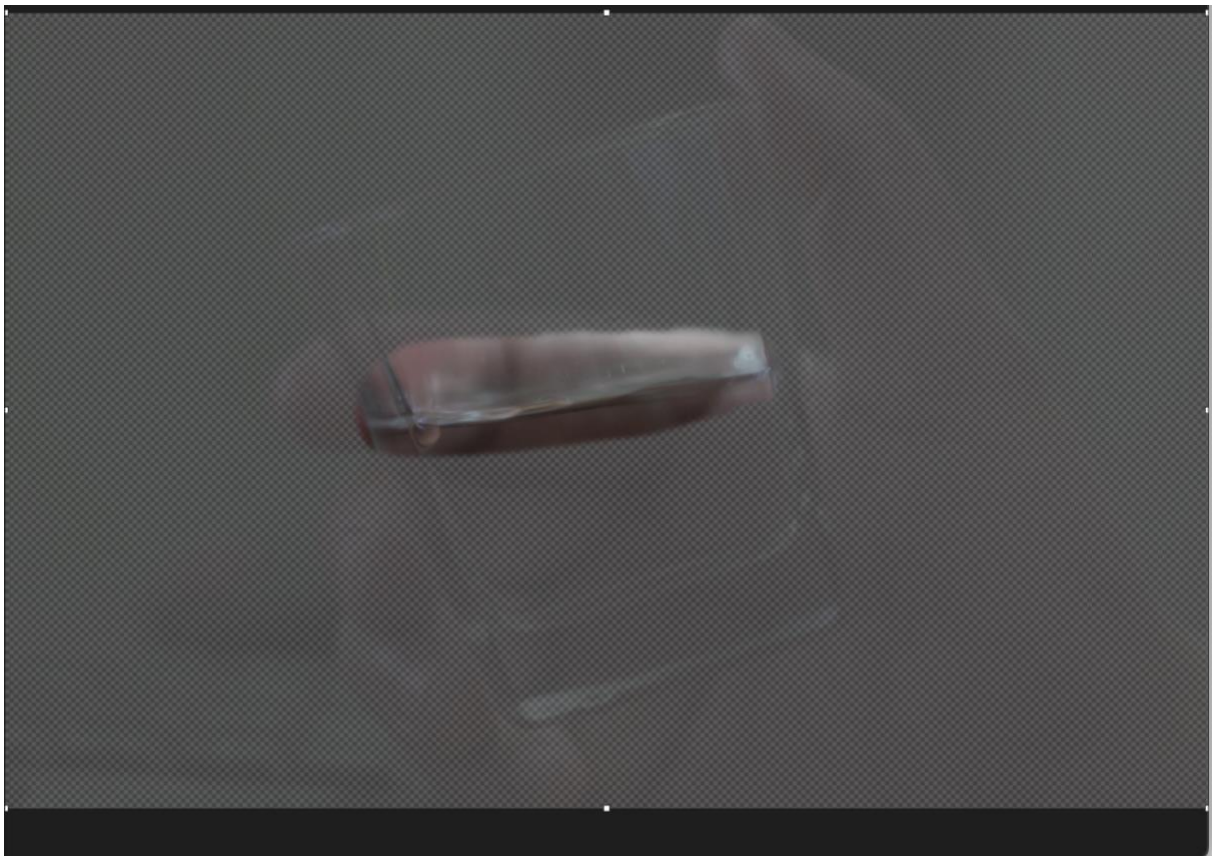
Yuankai Zhu, 21134118

Keywords : Computer Vision, Cinemagraph, OpenCV, Template Matching, Canny Edge Detection

Abstract : "Cinemagraph" is a novel filming technique, and its filming methods and steps are relatively cumbersome, so it has not yet become widely popular in the field of photography. This article aims to study the feasibility of using computer vision-related technology to minimize the tedious work of the process. I was developing in iOS environment using Xcode and debugging on an iOS device. I have tried two different methods from OpenCV for template matching and edge detection. Currently, the results are far from the expected targets with only barely usable results in some special cases.

Introduction: Cinemagraphs are still photographs in which a minor and repeated movement occurs, forming a video clip. The first cinemagraph was recorded in 2011 by Kevin Burg and Jamie Beck[i]. It creates an illusion that time has been frozen for a portion of the scene. The traditional production process for Cinemagraph involves using a tripod and high-speed shutter to take a series of photos of a pre-designed scene, typically around 10 frames depending on the subject. Then, each frame is edited to extract the elements that will move in the final product, and these elements are individually pasted back onto the first frame. Finally, the frames are combined to create a video or gif. This workflow is not particularly brain-burning, but the operational steps and workload are tedious and heavy.

Here is a demonstration of how to make a cinemagraph in traditional way.

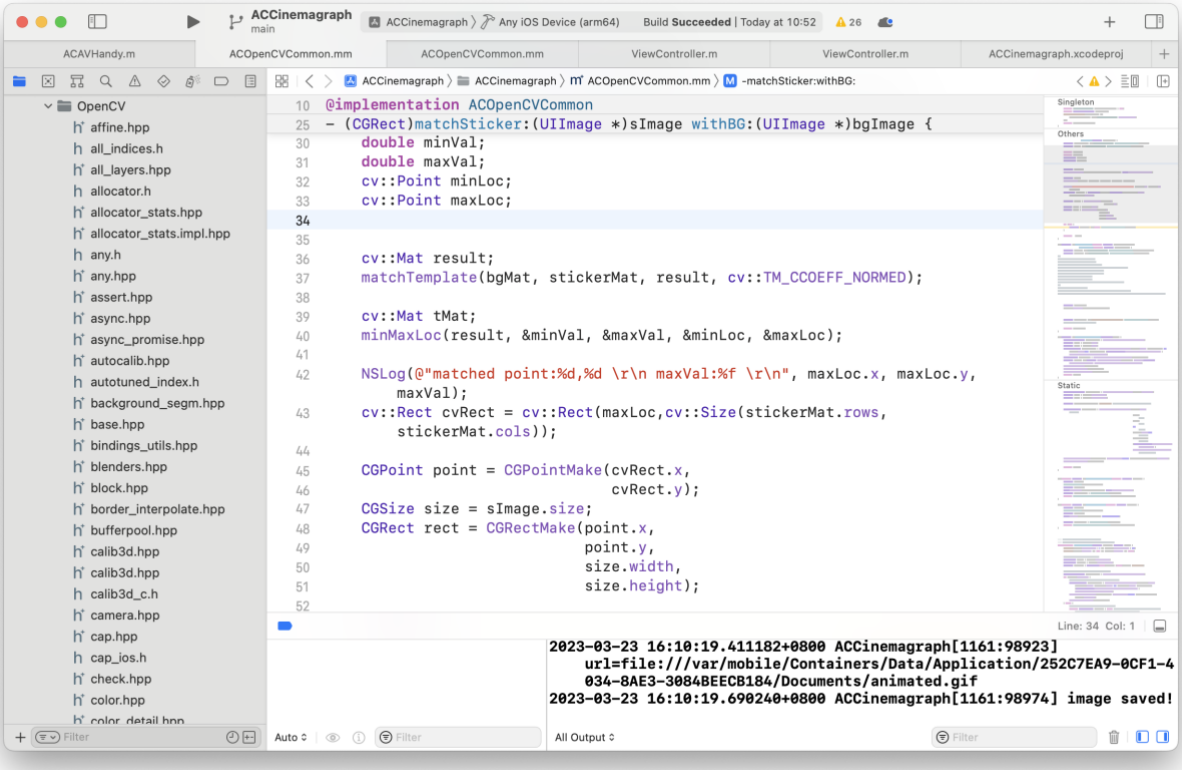


Yuankai, Zhu. (2014). A glass of water demo [Image]

As you can see, there are 9 frames in total. The first frame is used as base frame with all the pixels in it. And for the rest I just keep the portion that I want it animated and paste it on the first frame and frame by frame. From my personal experience, in some case, even when using a tripod, the portions that I want to be animated can slightly deviate from the base frame. In the example above, my hand was unable to remain completely still due to my breathing and heartbeat. So, this research is about the feasibility of removing the process of manual editing and aligning pixels. For human vision this is a super easy thing. For computer, I thought it's not that complex in the beginning. I thought each frame is not significant different. It's all about matching. But I underestimated its complexity. Here is my research process and current research results.

The development process:

With the idea of matching. I searched related function in OpenCV. I choose OpenCV because I think there will be more practical codes and examples on it since it's one of the most widely used framework. I was using iOS development environment because I'm more familiar with it. I imported OpenCV as a framework of my demo project.



First, I tested that OpenCV has been imported correctly. Verify this by testing a simple function that converts an image to a grayscale matrix.

```
+ (nonnull UIImage *)cvtColorBGR2GRAY:(nonnull UIImage *)image {
    cv::Mat bgrMat;
    UIImageToMat(image, bgrMat);
    cv::Mat grayMat;
    cv::cvtColor(bgrMat, grayMat, cv::COLOR_BGR2GRAY);
    UIImage *grayImage = MatToUIImage(grayMat);
    return RestoreUIImageOrientation(grayImage, image);
}
```

Then I tried matchTemplate() [ii] function provided by OpenCV.

1. **method=TM_SQDIFF**

$$R(x,y) = \sum_{x',y'} (T(x',y') - I(x+x',y+y'))^2$$

2. **method=TM_SQDIFF_NORMED**

$$R(x,y) = \frac{\sum_{x',y'} (T(x',y') - I(x+x',y+y'))^2}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}}$$

3. **method=TM_CCORR**

$$R(x,y) = \sum_{x',y'} (T(x',y') \cdot I(x+x',y+y'))$$

4. **method=TM_CCORR_NORMED**

$$R(x,y) = \frac{\sum_{x',y'} (T(x',y') \cdot I(x+x',y+y'))}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}}$$

5. **method=TM_CCOEFF**

$$R(x,y) = \sum_{x',y'} (T'(x',y') \cdot I'(x+x',y+y'))$$

where

$$T'(x',y') = T(x',y') - 1/(w \cdot h) \cdot \sum_{x'',y''} T(x'',y'')$$
$$I'(x+x',y+y') = I(x+x',y+y') - 1/(w \cdot h) \cdot \sum_{x'',y''} I(x+x'',y+y'')$$

6. **method=TM_CCOEFF_NORMED**

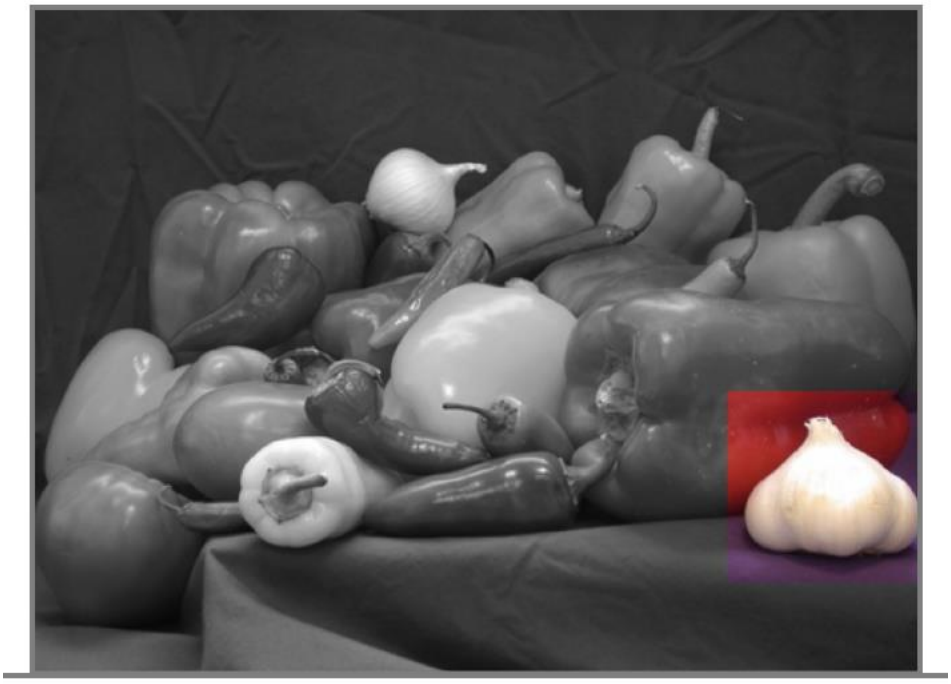
$$R(x,y) = \frac{\sum_{x',y'} (T'(x',y') \cdot I'(x+x',y+y'))}{\sqrt{\sum_{x',y'} T'(x',y')^2 \cdot \sum_{x',y'} I'(x+x',y+y')^2}}$$

It simply slides the template image over the input image (as in 2D convolution) and compares the template and patch of input image under the template image. Several comparison methods are implemented through 6 different equations.

I converted the examples provided in the development documentation into Objective-C code.

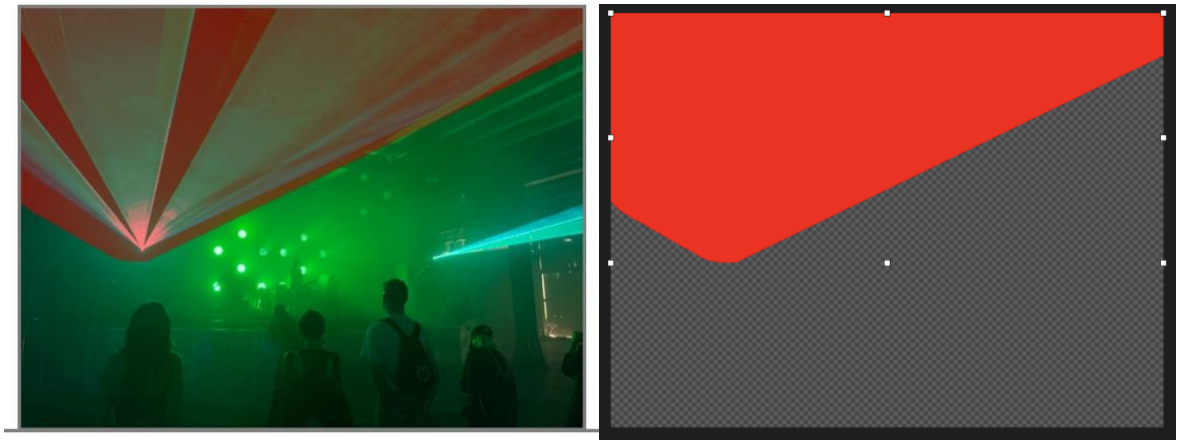
```
- (CGRect)matchSticker:(UIImage *)sImage
               withBG:(UIImage *)bgImage {
    cv::Mat stickerMat = [ACOpenCVCommon cvMatFromUIImage:sImage];
    cv::Mat bgMat = [ACOpenCVCommon cvMatFromUIImage:bgImage];

    double minVal;
    double maxVal;
    cv::Point minLoc;
    cv::Point maxLoc;


    cv::Mat result;
    matchTemplate(bgMat, stickerMat, result, cv::TM_CCOEFF_NORMED);

    cv::Mat tMat;
    minMaxLoc(result, &minVal, &maxVal, &minLoc, &maxLoc);

    NSLog(@"matched point:%d,%d \r\n maxVal:%f \r\n", maxLoc.x, maxLoc.y, maxVal);
    cv::Rect cvRect = cv::Rect(maxLoc,cv::Size(stickerMat.rows, stickerMat.cols));

    CGPoint point = CGPointMake(cvRect.x,
                                cvRect.y);
    CGSize size = sImage.size;
    CGRect rect = CGRectMake(point.x,
                             point.y,
                             size.width,
                             size.height);

    //UIImage *resImg = [self UIImageFromCVMat:result];

    return  rect;
}
```

I used a pre-cropped portion of the original test image as the template for matching it with the test image. Afterwards, I draw it through objc onto a greyscale version of the test image to demonstrate its effectiveness.



This indicates that the code is functioning properly and that the console output confirms that the maximum match value is 0.999299.

I'll skip the details of the other codes I wrote for implementing the entire process, such as generating a mask image from user touch, cropping each frame, and exporting the image set to a video file.



I tested the codes by using a photo set that I personally filmed at a live show. The set includes 10 frames captured using the high speed shutter on my iPhone and taken by hand.

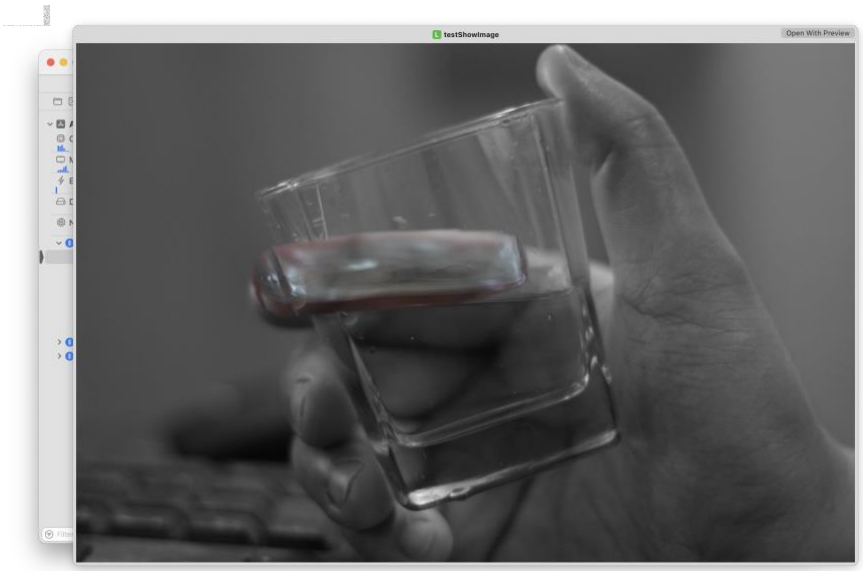
(original photo set) (processed photo set)

I have successfully created a gif/video that closely matches my desired outcome. As you can see the background on the right is completely still. The test results seem usable, but a minor shaking of the laser light source is noticeable.

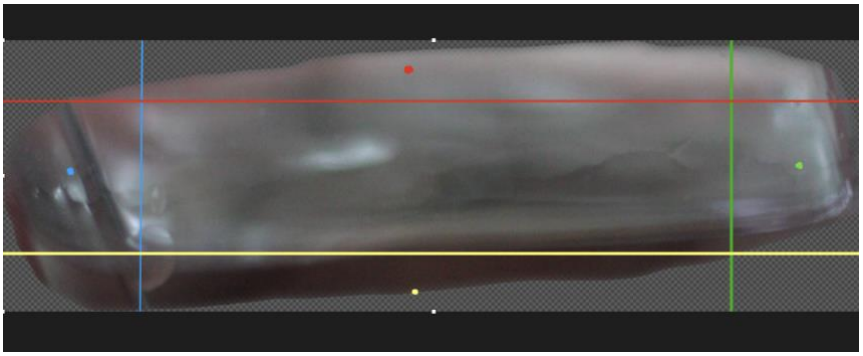
The matching result from the console shows : [0.738948, 0.701110, 0.652286, 0.667243, 0.683662, 0.752907, 0.744485, 0.764951], which is not very good.

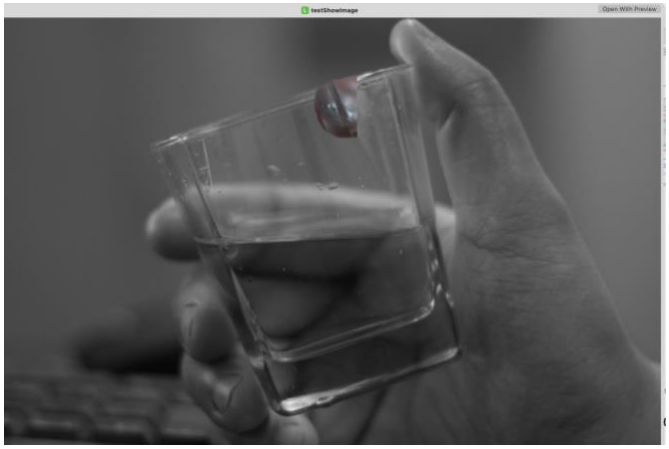
When I delved deeper, I realized that the first attempt was a stroke of luck. As I tested more photo sets, many failures occurred. Like the example photo set “A glass of water”, I didn’t use it for test because I already lost each original frame of the photo set. There is just a .pxd file with cropped layers. But when I tested some layers of it, I found the usability of matchTemplate() function didn’t fully meet my requirements.

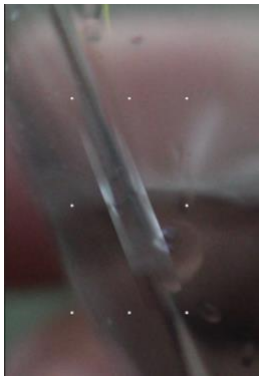
Failures:



From this test result “matched point: 888,949 maxVal: 0.543284”, I really can’t blame the algorithm, because it seems it is trying to match the cropped piece to my finger. I assume this is caused by most of the pixels in cropped piece has significant difference from the first frame. Only the edge of the glass seems the same thing as first frame. For this guess I tried to split the cropped image into four areas and will use the best match of them as the result.

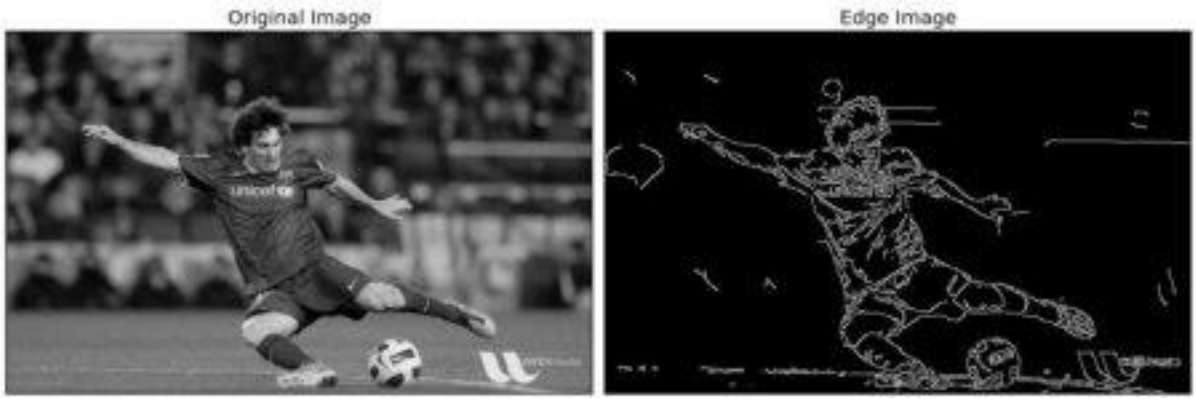






It failed again. Then I checked just the edge of the glass in photo editing software. I realized even the edge changed a lot from the computer perspective. And we can't expect users to mark such a tiny edge, otherwise it's pointless, why not to use photo editing software directly.

With this understanding I have to search for other solutions. It seems I need to figure out how to match edges. I found another function of OpenCV. **Canny Edge Detection**[iii].

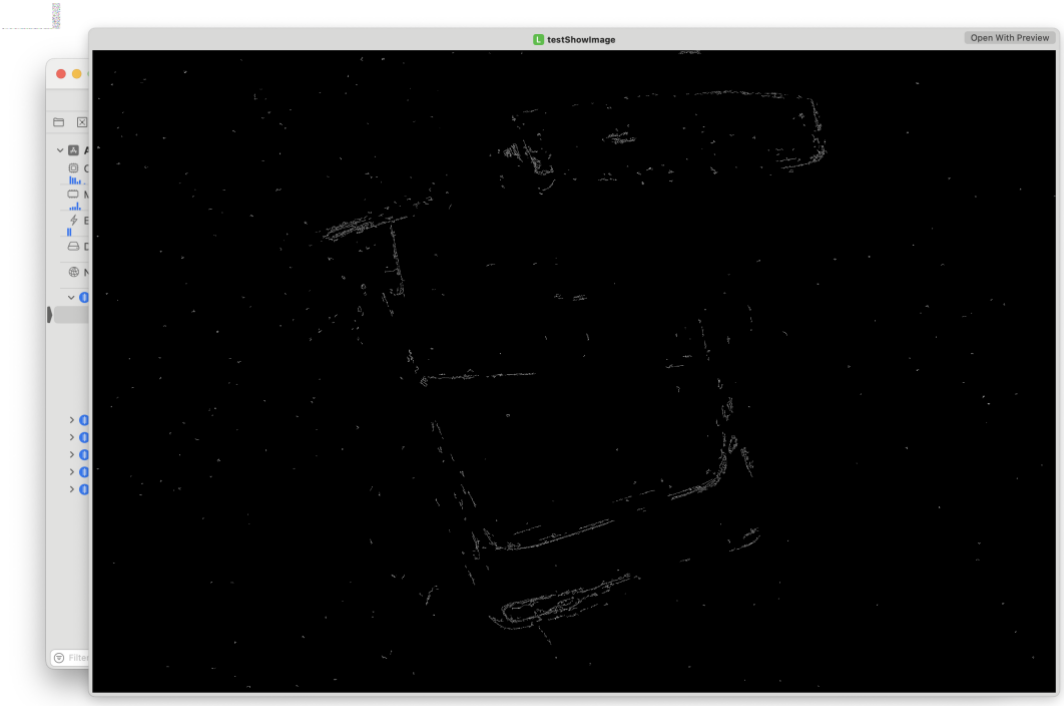


(Test image on OpenCV website)

It is a popular edge detection algorithm developed by John F. Canny[iv]. It is a multi-stage algorithm with "Noise Reduction", "Finding Intensity Gradient of the Image", "Non-maximum Suppression", "Hysteresis Thresholding".

After I translated the sample code into Objective-C code.

```
+ (nonnull UIImage *)cvtEdgeImage:(nonnull UIImage *)image {
    cv::Mat bgrMat;
    UIImageToMat(image, bgrMat);
    cv::Mat edgeMat;
    cv::Canny(bgrMat, edgeMat, 120.0, 40.0);

    UIImage *resImage = MatToUIImage(edgeMat);
    return RestoreUIImageOrientation(resImage, image);
}
```

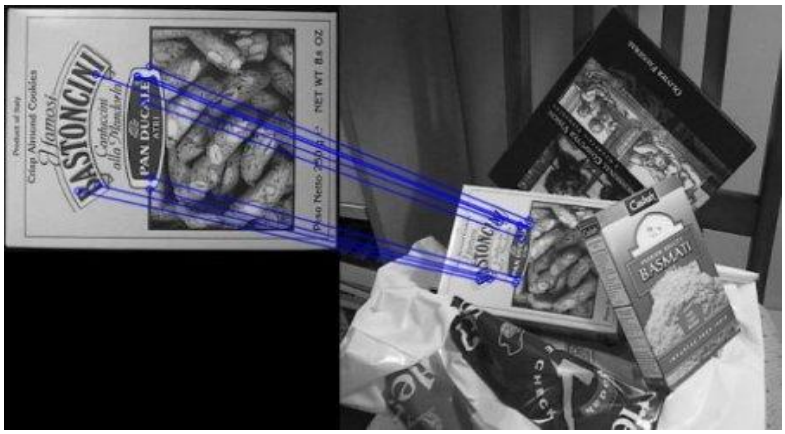


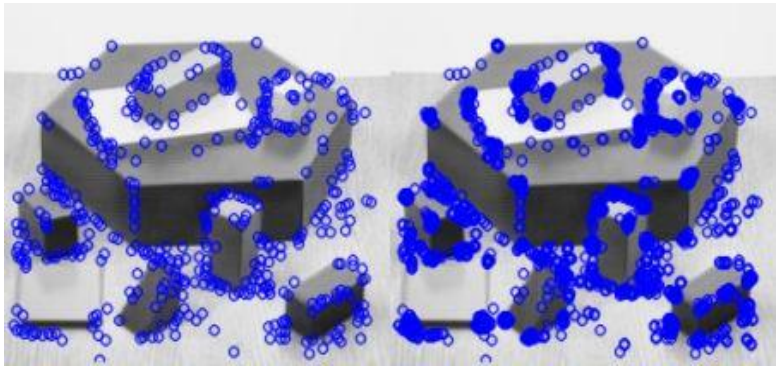
I’ve got a worse result at “matched point:1757,17 maxVal:0.027743”.

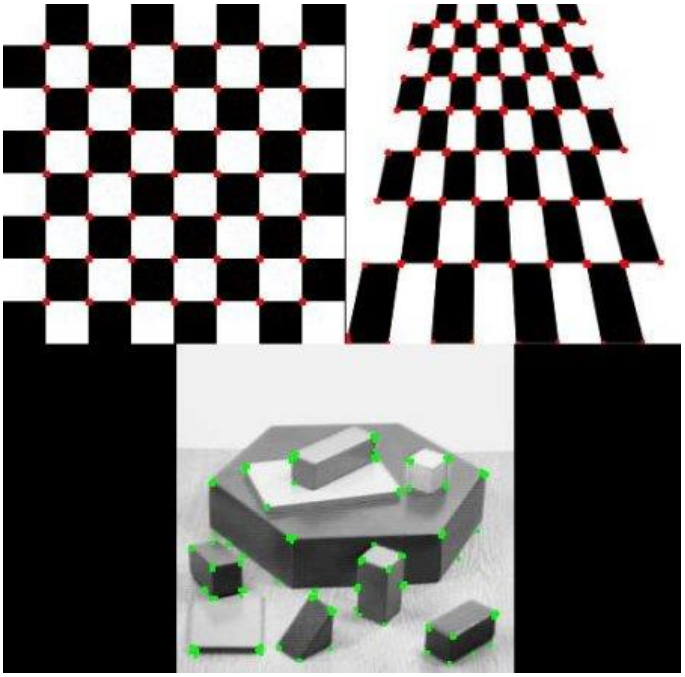
I really underestimated the complexity of this goal. What appears as an easy task to human vision can be a significantly challenging task for computer vision.

Future scope:

My next step is to research another series of functions **Feature Detection and Description** [v] of OpenCV.



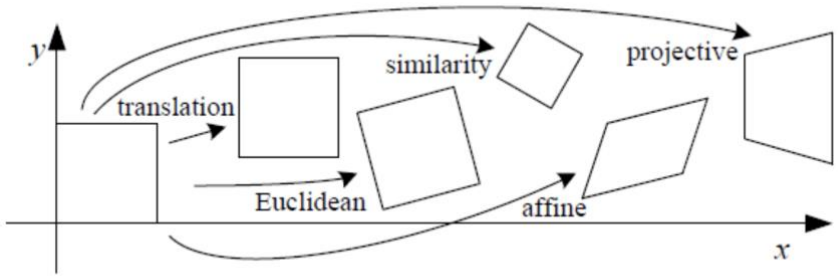




It includes “Feature Matching + Homography to find Objects”, “Harris Corner Detection”, “Shi-Tomasi Corner Detector & Good Features to Track”, “FAST Algorithm for Corner Detection” and so on.

And I'll have more basic understanding of computer vision, such as:

Basic Set of 2-D Transformation



Richard Szeliski, "Computer Vision: Algorithms and Application"[vi]

While writing this review and browsing the internet, I came across a fascinating algorithm called **KLT (Kanade-Lucas-Tomasi feature tracker[vii])**.

It is a common computer vision algorithm mainly used in object tracking, image matching, and other related fields.

The core idea of this algorithm is to calculate the global motion parameters based on the motion of pixels in a local region, which enables the target tracking and pose estimation tasks.

(Optical Flow[viii] test image on OpenCV website)

References:

[i] Cinemagraphs.com. (2023). About Retrieved from http://cinemagraphs.com/about

[ii] OpenCV. (n.d.). matchTemplate(). Retrieved March 25, 2023, from https://docs.opencv.org/4.x/df/dfb/group__imgproc__object.html#ga586ebfb0a7fb604b35a23d85391329be

[iii] OpenCV. (2020). Canny Edge Detection. Retrieved March 25, 2023, from https://docs.opencv.org/4.2.0/da/d22/tutorial_py_canny.html

[iv] Wikipedia contributors. (2023, March 24). John Canny. In Wikipedia, The Free Encyclopedia. Retrieved 06:00, March 25, 2023, from https://en.wikipedia.org/wiki/John_Canny.

[v] OpenCV. (2021). Feature Detection and Description. OpenCV 4.x documentation. Retrieved from https://docs.opencv.org/4.x/db/d27/tutorial_py_table_of_contents_feature2d.html

[vi] Szeliski, R. (2011). Figure 2.4 [Basic set of 2D planar transformations]. In Computer Vision: Algorithms and Applications. Springer.

[vii] Wikipedia contributors. (2022, January 23). Kanade–Lucas–Tomasi feature tracker. In Wikipedia. https://en.wikipedia.org/wiki/Kanade%E2%80%93Lucas%E2%80%93Tomasi_feature_tracker

[viii] OpenCV. (2018). Optical Flow. OpenCV 3.4.4 documentation. Retrieved from https://docs.opencv.org/3.4/d4/dee/tutorial_optical_flow.html