CS512 Assignment 5: report

Gady Agam
Illinois Institute of Technology

Abstract

1. Problem statement

This assignment is about optical flow.

My program has implemented Lucas-Kanade and affine-flow algorithm. They all base on OFCE(optical flow constraint equation), but the idea is a little different, making the time consuming and the accuracy different.

2. Proposed solution

Since these two algorithm all base on OFCE, let's talk about it first.

The idea of OFCE is easy: assuming that the intensity of two corresponding points in two images (have small different, two frames in a video) are the same, we have:

$$I(x(t), y(t), t) = C$$

I is the image; x, y are the coordinate of a point; t is time; C represents a constant. We find its derivative base on time, then:

$$\frac{d}{dt} I(x(t), y(t), t) = 0 \rightarrow I_x * x_t + I_y * y_t + I_t = 0$$

I_x, I_y are the derivative of image; x_t, y_t are the observed 2D motion on image we want to find; I_t is the derivative of image base on time: we can imagine that images are in a stack, the simplest "derivative of image base on time" is just the difference between two neighbor images. Same as I_x and I_y, we want to do smoothing before finding the derivative to increase the robustness. So we do gaussian smoothing base on time (the kernel is 1D now, because we only smoothing on time dimension, not on x and y dimension) before calculate the difference.

There's a problem in OFCE. Since there's only one equation but with two unknowns, it will make the calculated observed motion vector be the production onto the gradient vector. So we want to give more restriction condition or look for other informations.

Lucas-Kanade algorithm is the one that looks for other informations. What it does is: for each point, consider its neighbors. Add up the squares of all left part of OPCE equations of its neighbors, and try to find its (x_t, y_t) (or (V_x, V_y)) which minimize the result. Finally we will get:

$$\begin{bmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{bmatrix} \begin{bmatrix} V_x \\ V_y \end{bmatrix} = \begin{bmatrix} -\sum I_x I_t \\ -\sum I_y I_t \end{bmatrix}$$

It's in Ax=B form. We can solve it. One problem is that matrix A maybe singular. This will happen when the point is not likely a corner point. So when implementing it we need to consider this. And Lucas-Kanade algorithm should show the result only on points that "likely" to be a corner.

Lucas-Kanade algorithm assume that the motion of neighbors are fix(basic the same). But when rotation happens, this assumption is not correct. So comes affine-flow algorithm. This algorithm will take rotation/affine into consideration.

Affine-flow algorithm is a little different from Lucas-Kanade algorithm. The base formula of them (the above formula) are the same, but the meaning of (V_x, V_y) is different. Instead of writing (V_x, V_y), it writes (u, v).

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} a_1 \\ a_4 \end{bmatrix} + \begin{bmatrix} a_2 & a_3 \\ a_4 & a_5 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

[a1; a4] is the translation matrix and [a2, a3; a4, a5] is the rotation matrix. They also have a scale parameter. Notice that Lucas-Kanande algorithm only consider [a1; a4] but not the rotation matrix. Now we need to find 6 parameters instead of 2 parameters in Lucas-Kanande algorithm. To estimate more parameters, we need to enlarge the neighborhood size that we look into. Also, the A matrix in Ax=B equation will be 6*6 size, so the calculation will be more complex than Lucas-Kanande algorithm. The program runs slow when using affine-flow algorithm.

3. Implementation details

The algorithms seem to be easy to implement because the equations is easy, but I still faced several problems that spend me a lot of time.

First let's talk about the color of motion vectors. Since I don't know how to draw an arrow in OpenCV, I decide to use different color to represent the directions of the vectors. The color gradually changes when the direction of the vector changes. If one vector is reliable, ignore the tone of color, one of (R, G, B) is 255. My color system is kind of like Munsell Color System, but maybe a little simpler.

Remember that we need to do smoothing on time dimension to find I_t. Instead of doing smoothing + difference, what I do is just convolute the image with “derivative of gaussian”.

If a point is not likely to be a corner, its matrix A in Lucas-Kanade will be singular, then the solution of the observed motion maybe unreliable. What we do is consider “reliability”: if the smallest eigenvalue of A^TA matrix of a point is relatively small comparing to other points, we think it has low reliability. Because if a point is not likely to be a corner, the elements of A matrix can be small, making the smallest eigenvalue also small. Also, if a point is a “weak” corner, A matrix is still small, smallest eigenvalue is relatively small, resulting that its reliability is still small. There maybe has some other factors that we use this comparison to determine reliability, but I haven't figure out them. If a point has a small reliability value, we let its intensity be small, vice versa. So the vector of a point which has large reliability value will be vivid. Since the “reliability value” is just a relative value, all of them is not larger than 1(we divided them by the largest “smallest eigenvalue”). So, we let user control a constant (“confidence parameter” track bar) that multiple to this “reliability value”. The default value is 20, means that as long as a smallest eigenvalue is not smaller than 0.05 times of the largest “smallest eigenvalue”, it will be 255 vivid.

Actually I lie a bit above. If we only let the color intensity of motion rely on the “reliability value” we will get a noise result(the result image will still has many small vectors even if no motion happens). This is because that the input image has some noise. Although we have smooth it base on time dimension, the noise is still remained. I tried to smooth the image more, but resulting that I can't do that, because the motion is also smoothed and the result out will be worse. So, except the “reliability value”, the intensity also relies on the length of the motion vector(simply multiply this two values). Now the result is nice.

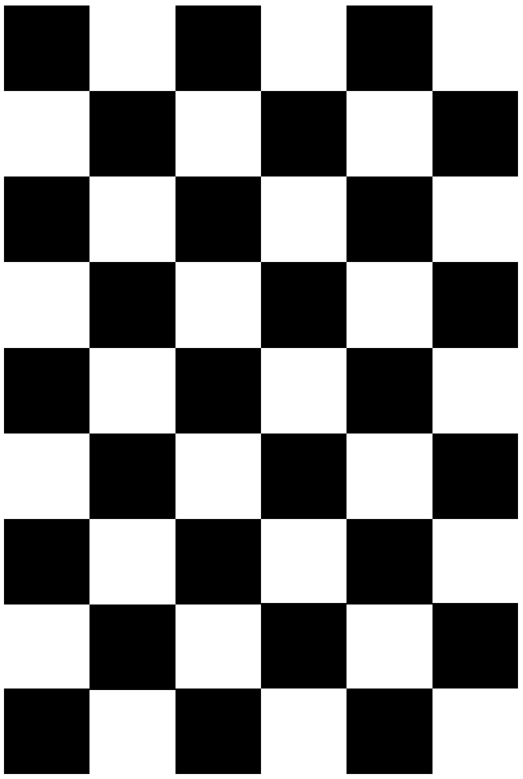
In the situation of affine-flow algorithm, the above method to calculate “reliability value” is not good enough. Because the matrix A of affine-flow algorithm is also related to coordinate x and y, a point which is not likely to be a corner may still has some large elements in A matrix, which may(?not sure, just guess) makes its smallest eigenvalue relatively large. So for affine-flow algorithm, besides the above step, we also do: if the ratio of (smallest-eigenvalue/second-smallest eigenvalue) of A^TA matrix is smaller than a threshold, we will ignore this point. The threshold can be controlled by the user with the “singular threshold” track bar.

There are five track bars in the program:

A) gaussian size(-11): the gaussian kernel size when we smooth on time dimension.
B) confidence parameter(-300): the constant we multiple to “reliability value” when effect the intensity of motion vectors' color.
C) interval size(-101): the density of the motion vectors that calculated. For example, “1” means that calculate motion vector for each pixels; “2” means that calculate motion vector for every other pixels. If set it to small value, the program may run slow.
D) window size(-101): the neighbor size that we look into in two algorithms.
E) singular threshold(-300): if the ratio of (smallest-eigenvalue/second-smallest eigenvalue) of A^TA matrix of a point is smaller than this threshold, we will ignore this point. The real value is divided by 300.

4. Results and discussion

We use the below chessboard to help showing the result, because its corners are relatively obviously.



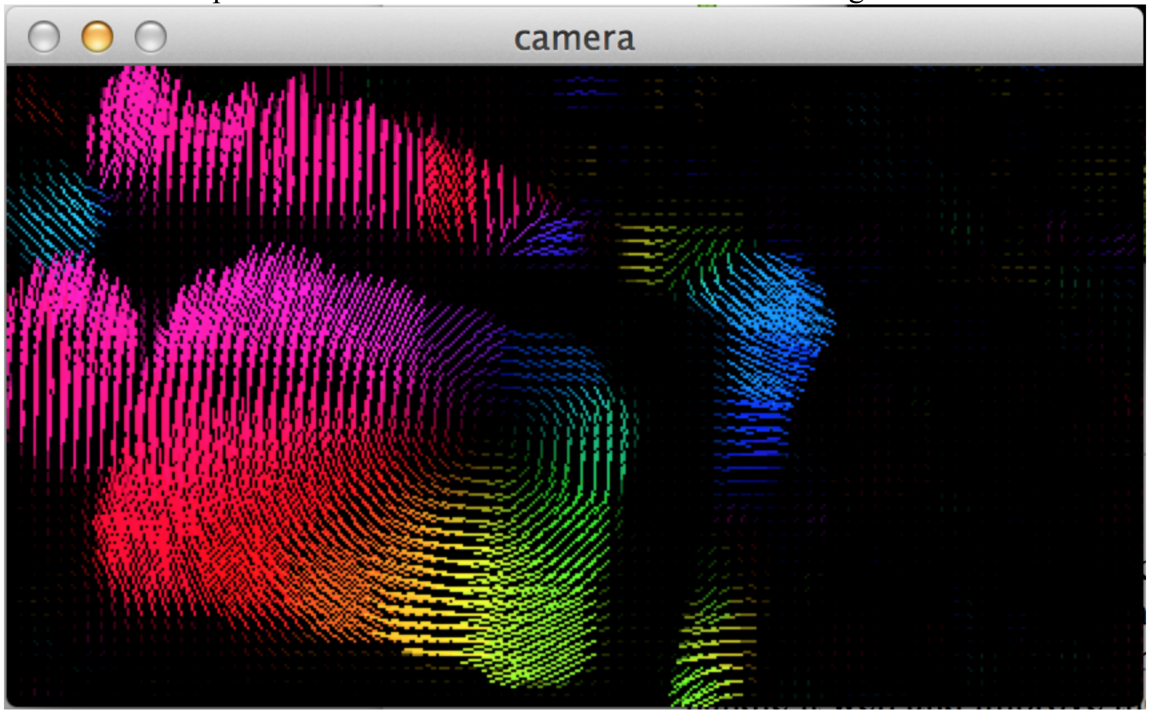
Here's the default value of each parameters:

A) gaussian size: 5
B) confidence parameter: 20
C) interval size: 5
D) window size: 30

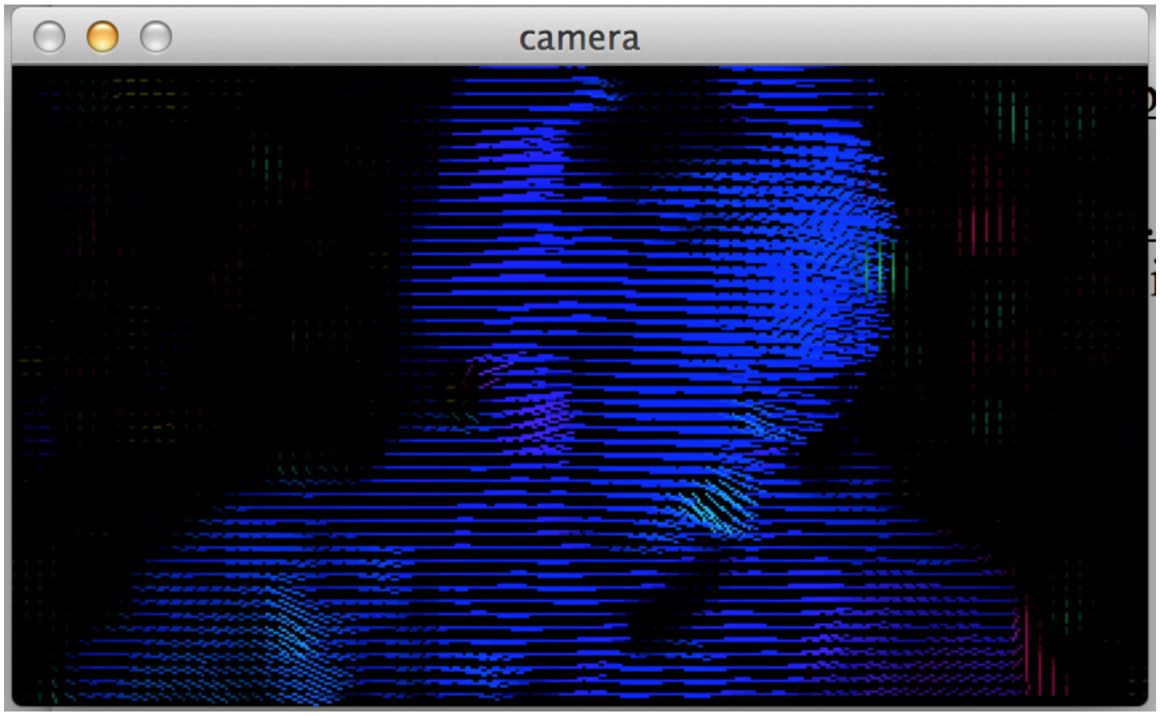
E) singular threshold: 0

To speed up the program, we reduce the images that captured by camera by 3*3=9 times.

Here's the OpenCV function which use Gunnar Farneback's algorithm:

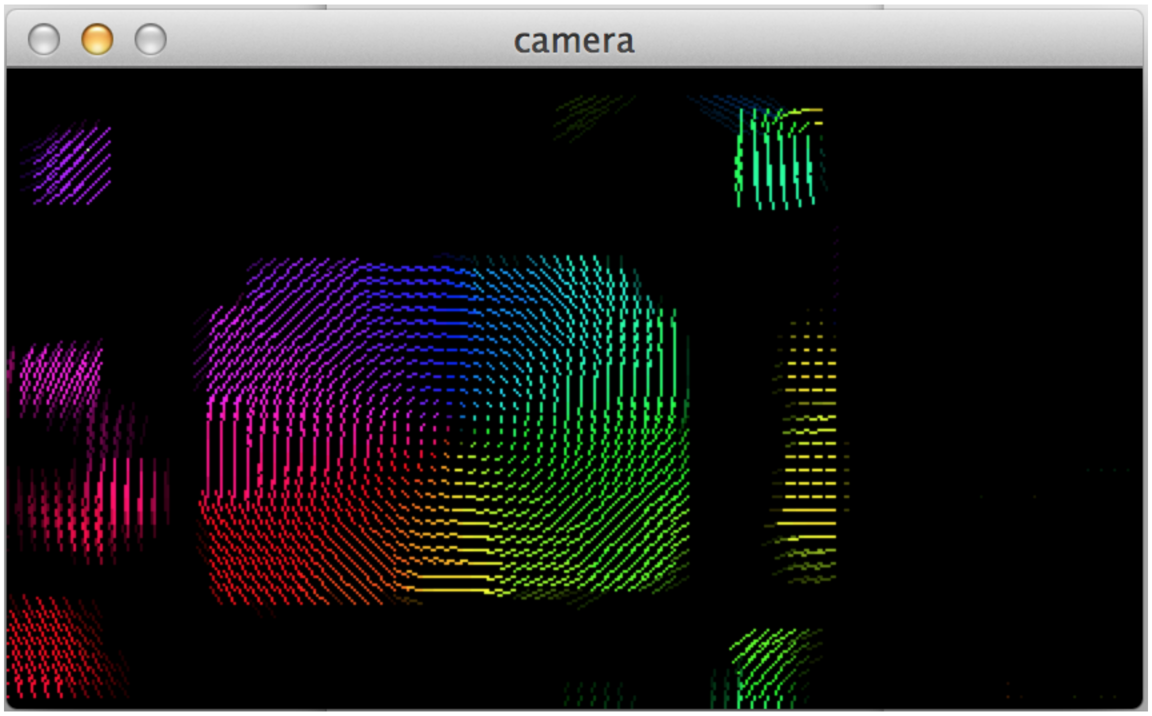


Here's the rotation of the chessboard.



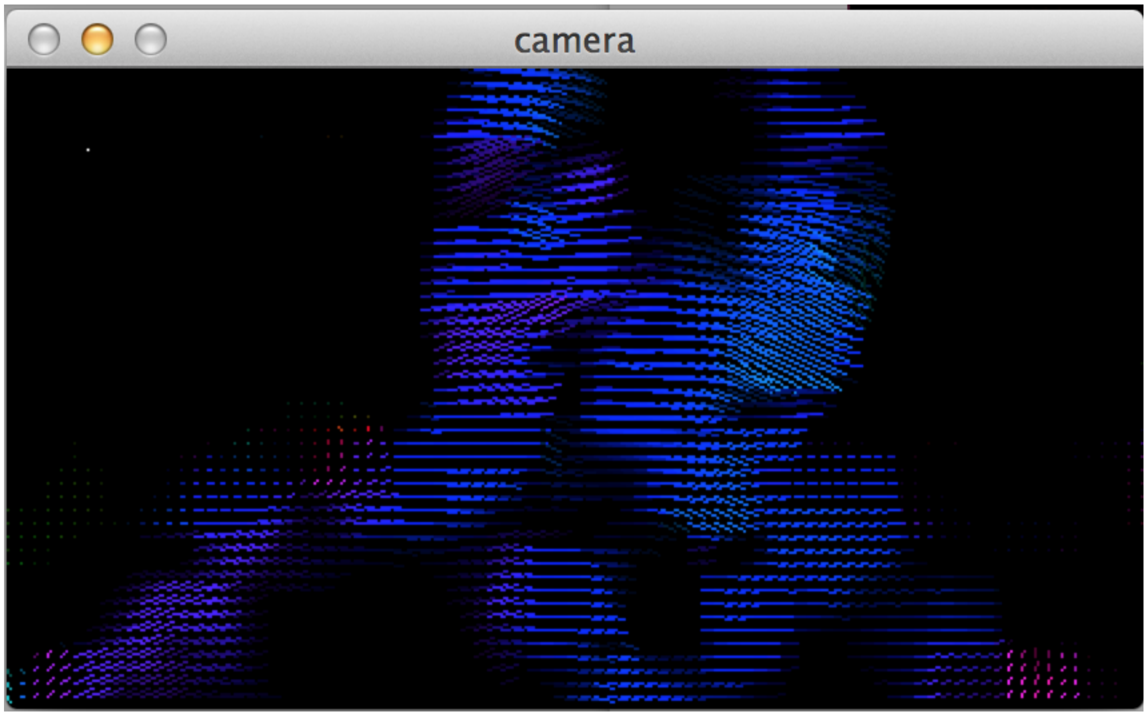
Here's I am moving to the right.

Then we show the result of my implementation:

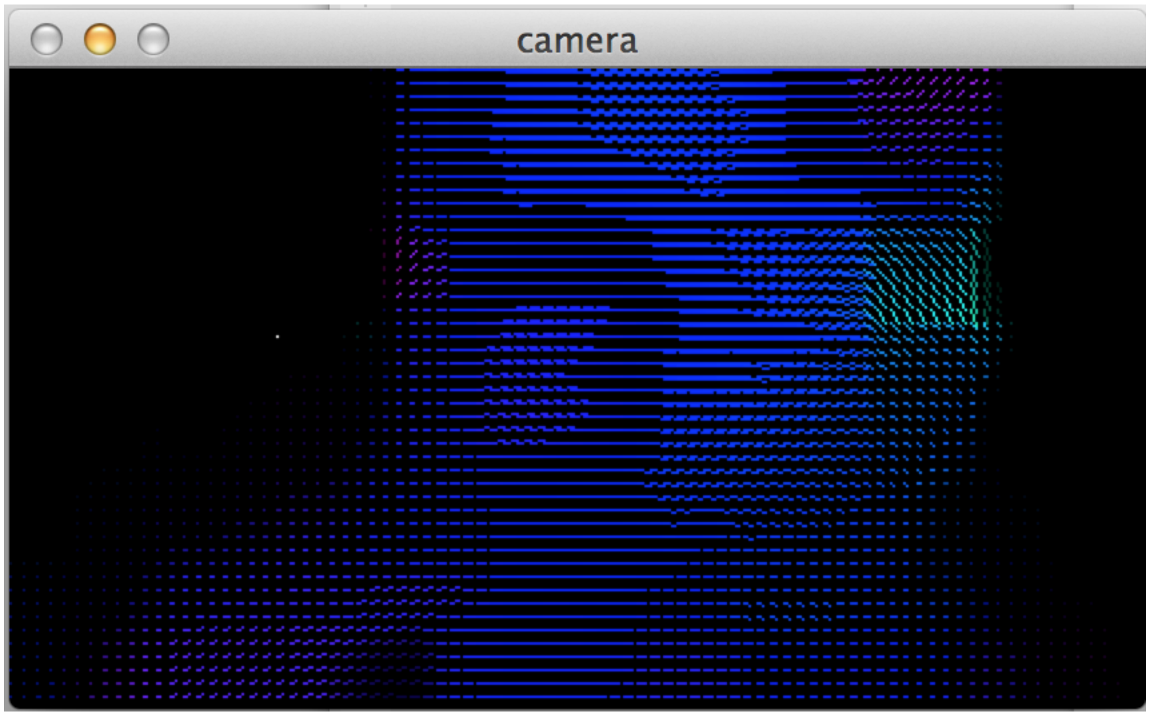


Rotation by Lucas-Kannade algorithm. All parameters are default. We can see that it looks pretty nice.

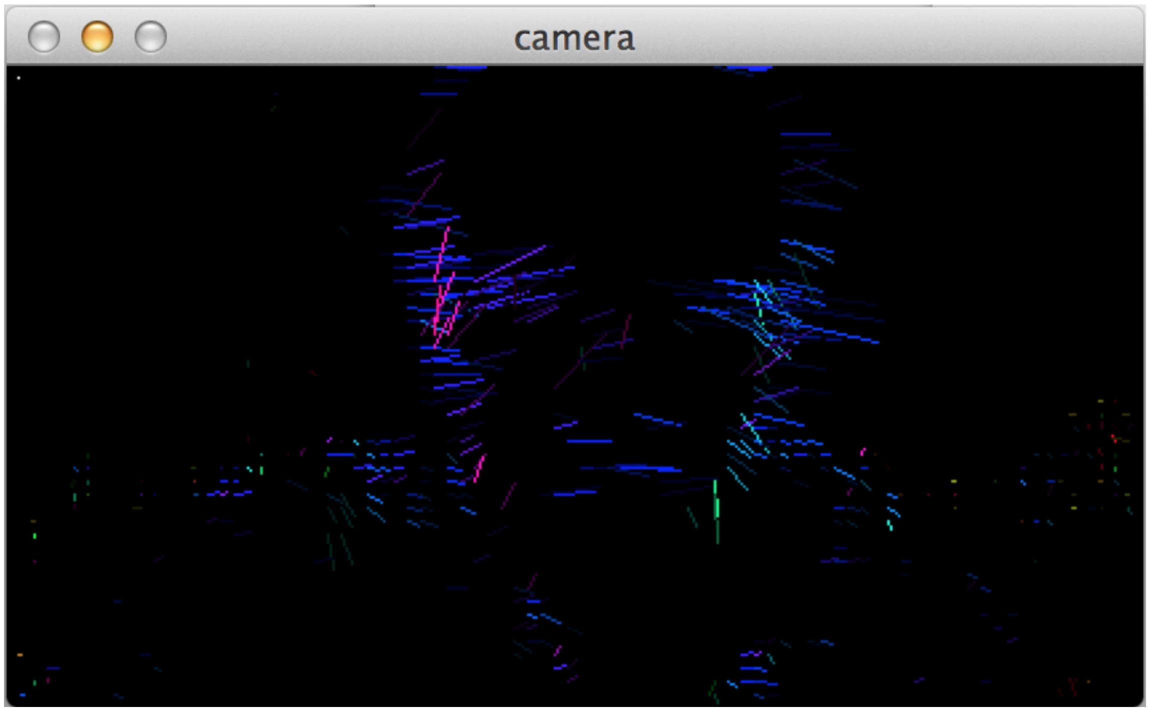
Since it's really hard to rotate the chessboard on a certain center using one hand and press 'p' using other hand, the rest test will only base on "my" motion.



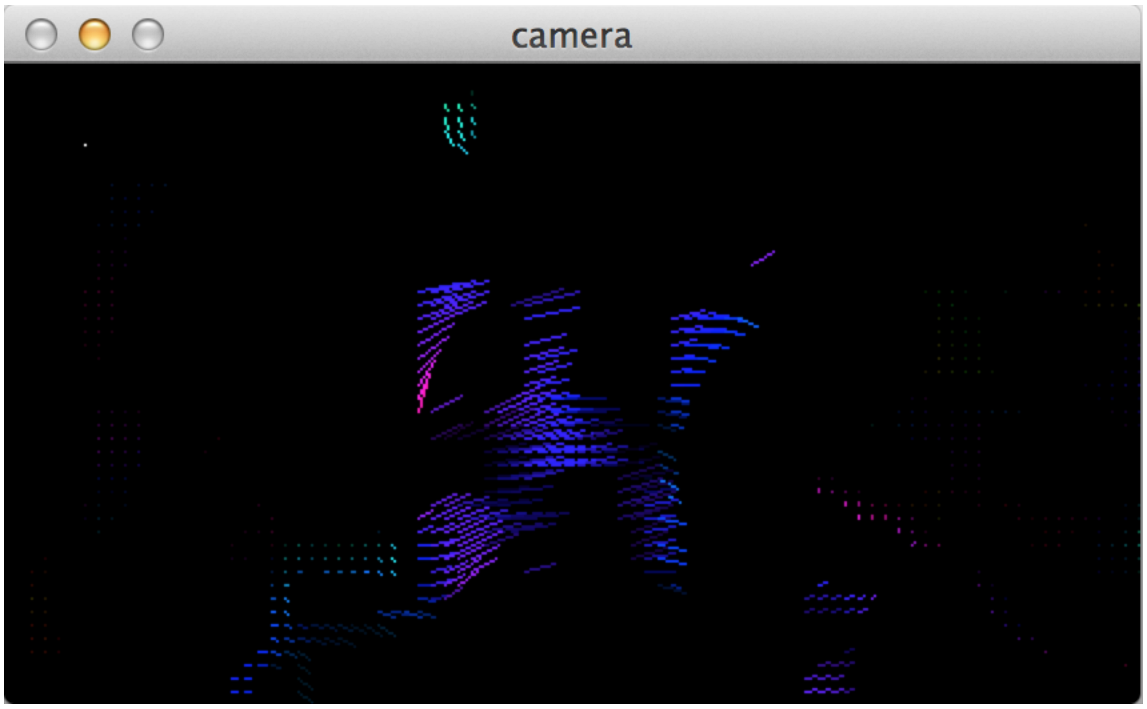
I am moving left by Lucas-Kannade algorithm. All parameters are default value. We can see that for may face and cloth, because they're basically the single color, so the motion is not detect, which is good. I'm wearing glass so the eyes part maybe has many "corner".



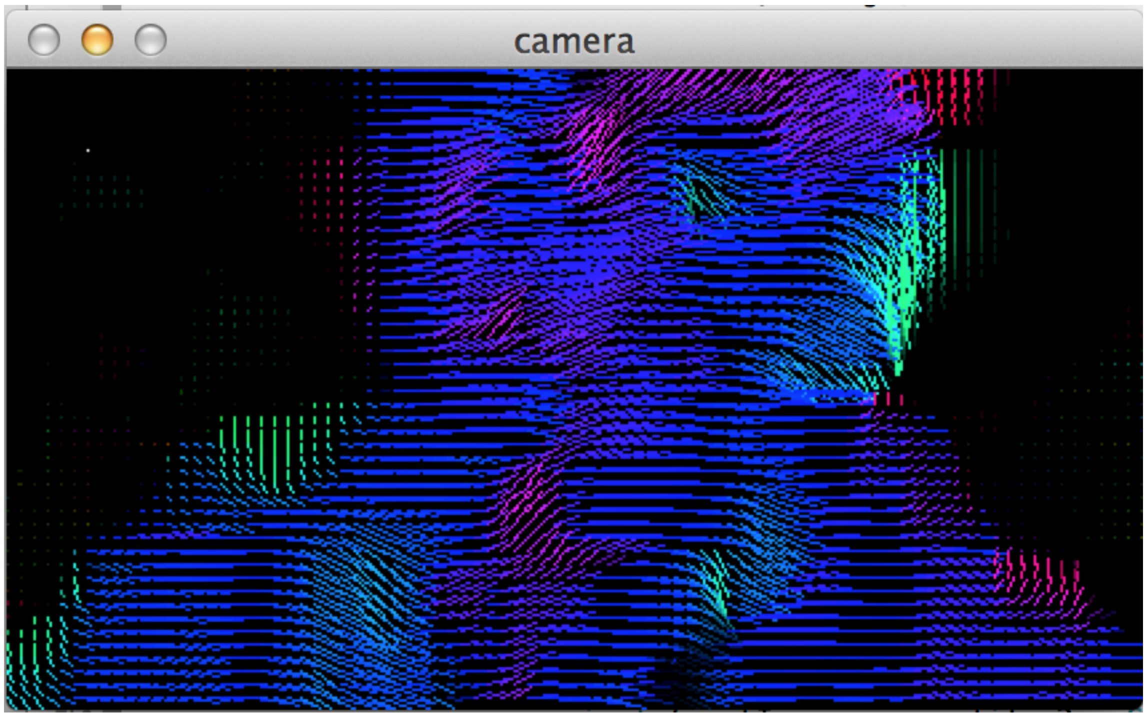
Same as previous, but the window size=101. The white point on the left shows how large the window is(from the white point to left-up corner of the image). We can see that the motion vectors are smoothed and more fixed. Of cause it will happen when it looks into larger size of neighborhood.



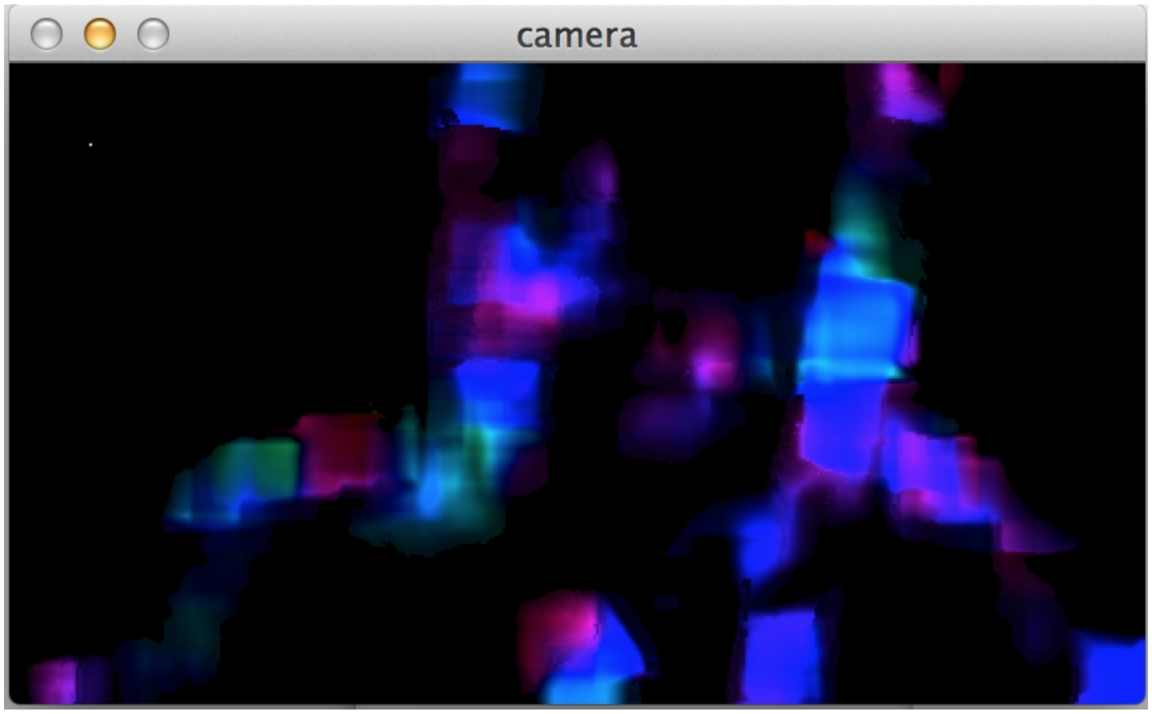
Same as previous, but the window size=5. We can see that the result is more noise because the neighborhood size is small.



I am moving left by Lucas-Kannade algorithm. All parameters are default value, except the "singluar threshold"=0.2. We can see that only the points that more likely to be a corner is showed, other are ignored. Well in the middle is my nose.



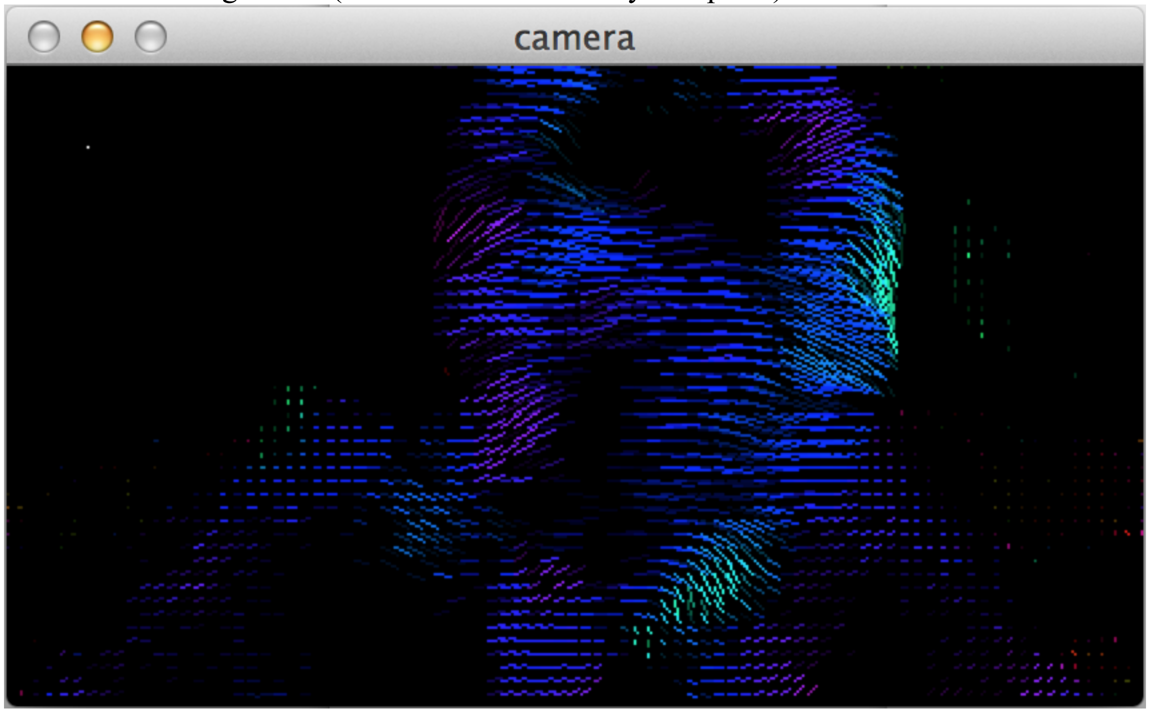
I am moving left by Lucas-Kannade algorithm. All parameters are default value, except the "confidence parameter"=300. We can see that the intensity of the points which not likely to be corners are increased(points on face && cloth), although they seems correct. But the intensity of the noise in the background is also increased(it may not very clear here, but under certain light condition and run the program by yourself, you will see it).



I am moving left by Lucas-Kannade algorithm. All parameters are default value, except the "interval size"=1. There's nothing need to say about it, just to show you how it can be. But the program becomes much slower if I do this.

The gaussian size is not affect the result that obviously, so I don't show you the result.

Actually I spend most of the time on the affine-flow algorithm. It run much shower then Lucas-Kanade algorithm (3-4 times slower on my computer). Here are its results:



I am moving left by affine-flow algorithm. All parameters are default value. We can see that: comparing to the L-K algorithm, some points are missing. That's because these points'

reliability value are small. Because the A matrix of the algorithm is more complex(rely on x, y coordinate) and the elements in it can be much larger, the smallest eigenvalues change a lot between different A^TA. So, the ratio(reliability value) can be small for some points although they may be nice corners. If we don't care about the discontinuance, the result is not bad. But because of its complex A matrix, there are some other problems:
(1) since this algorithm takes longer time to calculate, on my computer it will calculate 5 frames per second. So you have to move slow to test this algorithm.
(2) the "singular threshold" and "reliability value" all rely on the eigenvalues, but the complexity makes the eigenvalues changes wildly. This will makes us harder to distinguish singular matrix or outliers, and can cause more outliers or noise in the result.
(3) if a formula is more complex, it's more likely to introduce noise or enlarge the noise in it. Although I can't say exactly what kind of noise it introduces, but through the result I think it introduces some.

Summary: From the results we can see: although the Lucas-Kannade algorithm doesn't consider rotation of the points, but it still can "show" a quiet nice result in rotation. It's because in the rotation center the motion vector is small, so even if the vector is wrong it's not obvious; far from the rotation center, the direction of the motion vectors are more likely to be fixed, so the algorithm can show a nice rotation. Although affine-flow algorithm is better than Lucas-Kannade algorithm in theory, it's harder to handle in practice because of its more complex matrix(or formula). This complex will cause different kinds of problems, making the result seems not even better than Lucas-Kannade algorithm in some aspects.

5. References