Thesis for the Degree of Master of Engineering

A Study on the Implementation of Video Stabilization for Real-time Application

by

Axel Paradis IRAKOZE

Department of IT Convergence and Application Engineering

The Graduate School

Pukyong National University

February 2019

A Study on the Implementation of Video Stabilization for Real-time Application

실시간 응용을 위한 비디오 안정화 구현에 관한 연구

Advisor: Prof. Jong Nam Kim

by

Axel Paradis IRAKOZE

A thesis submitted in partial fulfillment of requirements for the degree of

Master of Engineering

in Department of IT Convergence and Application Engineering
The Graduate School, Pukyong National University

February 2019

A Study on the Implementation of Video Stabilization for Real-time Application

A dissertation

by

Axel Paradis IRAKOZE

Approved by:

[illegible]

(Chairman) Prof. Bong Kee Sin

Young Bong Kim

(Member) Prof .Young Bong Kim

[illegible]

(Member) Prof. Jong Nam Kim

February 2019

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

MEMS:	Micro-Electro-Mechanical Systems
OIS:	Optical Image Stabilization
TBB:	Threading Building Blocks
CUDA:	Compute Unified Device Architecture
RAM:	Random Access Memory
CPU:	Central Processing Unit
MSER:	Maximally stable extremal region extractor.
SURF:	Speeded Up Robust Features
GFTT:	Good Features to Track

ABSTRACT 추상 (KOREAN 한국어)

비디오 안정화는 비디오에서 불안한 동작을 줄이는 기술입니다. 이 논문에서는 Optical Flow : Feature 추출, Lucas-Kanade 방법을 사용한 Optical Flow, 이미지 아핀 변환을 사용하여 비디오 안정화 단계를 설명합니다. 이 글에서는 비디오 안정화의 각 단계에 관련된 수학적 모델에 대해서도 논의 할 것입니다. 이 문서의 뒷부분에서는 고화질 비디오를 적용하는 동안 비디오 안정화 결과를 향상시키고 더 효율적으로 만드는 방법에 대해 언급했습니다.

이 논문에서는 먼저 비디오 안정화의 역사와 소프트웨어 기술에 대한 필요성이 기계적 해결책보다 어떻게 논의되는지에 대해 논의 할 것이다. 이 논문에서 다루는 모든 수학 공식을 다루는 Optical Flow 를 사용하여 비디오 안정화에 대해 논의 할 것입니다. 이 논문은 옵티컬 플로우를 사용한 비디오 안정화와 관련된 모든 수학적 모델에 대한 독자의 이해를 돕습니다.

이 논문에서는 다양한 이미지 해상도와 FPS 에 대한 우리 프로그램의 결과에 대해 논의한다. 우리는 또한 다른 비디오 안정화 알고리즘의 결과를 우리의 프로그램과 비교할 것입니다.

OpenCV 기능을 사용하여 옵티컬 플로우를 사용한 비디오 안정화를 성공적으로 구현할 수 있습니다. Optical Flow 기반 비디오 안정화는 순수한 소프트웨어 기반이므로 우리 프로그램의 성능은 시스템마다 다릅니다. 프로그램 속도를 높이기 위해 Intel TBB, Nvidia CUDA (Nvidia GPU 기반 시스템에서만)를 사용하여 프로그램에서 병행 을 사용할 수 있습니다.

ABSTRACT

Video Stabilization is the technique to reduce jittery motion in a video. In this thesis, we shall discuss the steps involved in video stabilization using Optical Flow: Feature extraction, Optical Flow using the Lucas-Kanade method, Image Affine transformation. In this thesis, we will also discuss mathematical models involved in each step of video stabilization. In the latter part of the thesis, we have mentioned methods to improve the video stabilization results and make it more efficient while applying to high-resolution videos.

In this thesis, first we shall discuss the history of video stabilization and how the need for software technique arose over mechanical solutions. Then we shall discuss video stabilization using Optical Flow, in which this thesis covers all the mathematical formulae involved. This thesis will give the reader a clear idea of all the mathematical models involved in video stabilization using optical flow.

In the latter part, this thesis discusses the result of our program on various image resolution and FPS. We shall also compare the results of other video stabilization algorithms with our program.

Video stabilization using optical flow can be successfully implemented by using OpenCV functions. As Optical Flow based video stabilization is purely software-based, the performance of our program varies from system to system. To speed up our program we can use parallelism in our program, by using Intel TBB, Nvidia CUDA(only on Nvidia GPU powered systems).

CHAPTER 1

INTRODUCTION

1.1 BACKGROUND

The human mind is designed to get a satisfaction out of the stable visual scenery and is disturbed by unstable and jittery visual scenery. Thus it becomes necessary to design a method to stabilize videos recorded by a jittery support. There are many techniques for video stabilization, in early stages of video stabilization involved a mechanical assembly integrated with cameras. But, with the development for powerful computational hardware, it is now possible to ditch these huge mechanical assemblies by introducing image processing software to compensate the jittery motion of camera support.

1.2 HISTORY

Video Stabilization method was first designed in 1991 through a mechanical stabilization housing. Primitive mechanical stabilization used 2 axis gyroscope, each for sensing motion in motion in two perpendicular axes. The output signal of these gyroscopes is analyzed by a microcontroller which actuates two perpendicular motors to compensate the motion sensed by the gyros [1]. Today there are mechanical video stabilizers available having 5 stabilization axes. The major disadvantage of this method was a large space and power requirement.

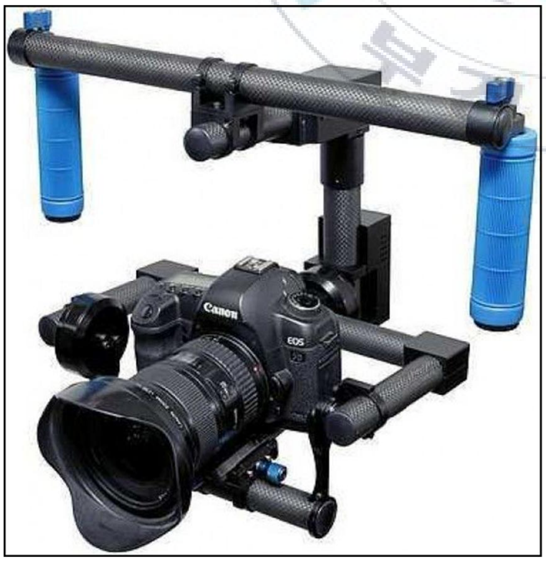


Fig 1. Dual axis mechanical video stabilization setup.

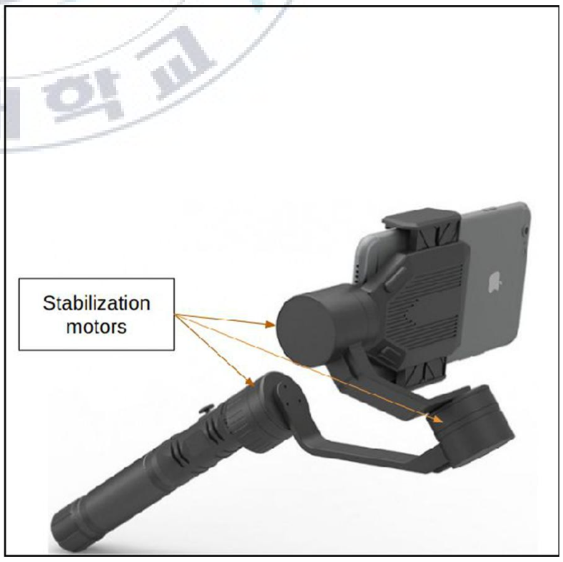
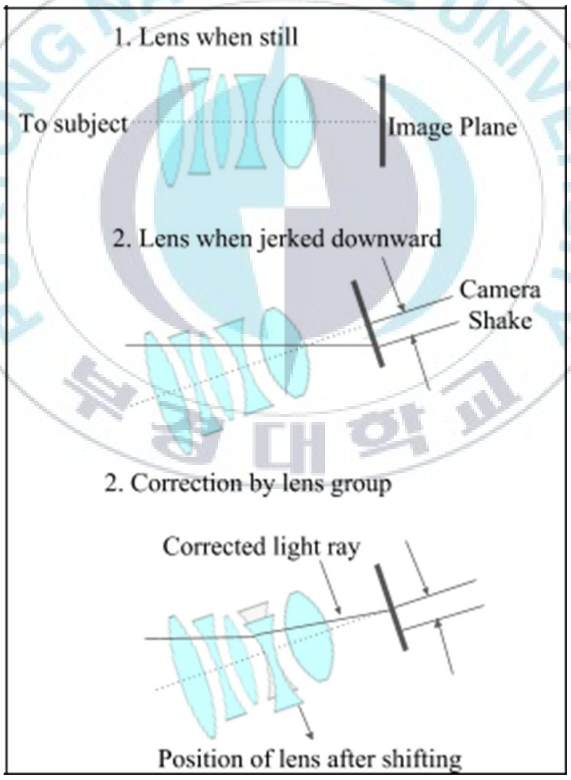


Fig 2. Reduced size of recent gimbal setup

Recent developments of mechanical video stabilization reduced the size of this bulky assembly due to diminishing semiconductor chip size and sensor size(MEMS). Mechanical video stabilization is also known as a gimbal. There is a major drawback of this techniques: external mechanical assembly requires significant space.

1.3 RECENT TECHNOLOGIES

Another method of video stabilization called as Optical Image Stabilization(OIS), this consists of a movable floating lens system. Similar to mechanical stabilization this method also uses gyroscope and accelerometer to sense motion change. The data from these sensors is analyzed by a microcontroller which commands the floating lens assembly so as to compensate the unintentional motion in yaw and pitch direction. Thus, OIS varies the actual Optical path of the incoming light to the sensor. Most of the OIS techniques do not correct rotation around the optical axis of the lens. The complex lens movement mechanism results in high costs.


1. Lens when still
To subject
Image Plane
2. Lens when jerked downward
Camera Shake
2. Correction by lens group
Corrected light ray
Position of lens after shifting
Fig 3. Optical Image Stabilization working.

Thus a method for stabilizing the video via software was developed. There are many techniques for digitally stabilizing videos. Here, we will be discussing corner matching method and apparent motion detection of these blocks across consecutive frames in a video stream via Optical Flow algorithm.

Picture blurring caused by hand jitter, a biological phenomenon occurring at a frequency below 20 Hz is more evident in higher resolution cameras thus OIS is widely used in smartphone cameras. Unlike software techniques for video stabilization, OIS also corrects lens motion blur. Fig 4. shows a comparison of two images, left one with OIS turned off and the right one with OIS turned on.

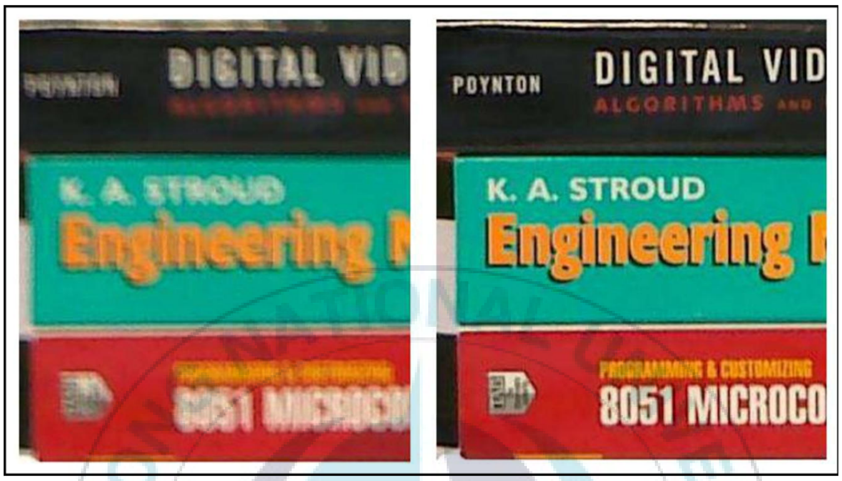


Fig 4. OIS turned off(left), OIS turned on(right)

1.4 PURPOSE

In this thesis, we will discuss a widely used video stabilization technique: by using Optical Flow. Optical Flow technique is a computationally intensive method, thus we will also discuss the methods to improve its performance by using parallel computing methods such as Intel TBB and widely known Nvidia CUDA[21] which is approximately 30 times faster than running image processing tasks on a single thread.

1.5 ORGANIZATION OF THE THESIS

Chapter 2 tells about the various methods of video stabilization in brief. In Chapter 3, we shall discuss in detail the mathematical calculations involved in video stabilization using Optical Flow. In Chapter 4, we have arranged results of our program in a graphical and tabular manner for easy understanding, in additions we shall also discuss the calculation we performed to obtain the FPS that our program can process at certain resolution and how we plotted the graphs, we have tested our program against various scenarios. In Chapter 5, we have explained what steps we took to create the video stabilization program. Chapter 6 concludes this thesis with Future scope for performance improvements of our program using parallel processing technologies.

CHAPTER 2

CONVENTIONAL METHODS OF VIDEO STABILIZATION

2.1 VIDEO STABILIZATION APPROACHES

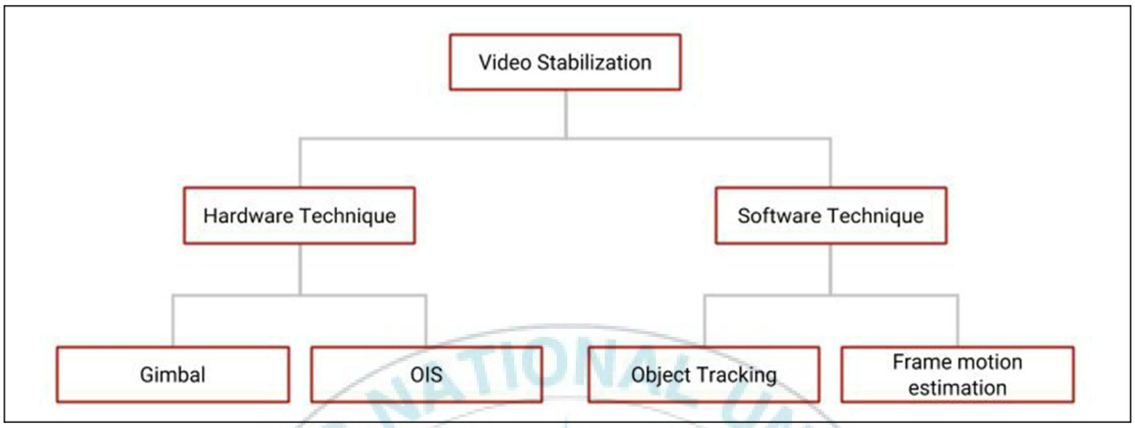


Fig 5. Video Stabilization approaches[20]

As discussed earlier in Chapter 1, about the Hardware techniques for video stabilization that they require space for mechanical assembly on the other hand software technique of video stabilization does not require physical space. There are two techniques for software video stabilization: Object Tracking and Frame Motion estimation.

Object Tracking technique consists of first detecting the object of interest and then keeping it stable throughout the video. Object Tracking does not fix jittery motion of the background. Usually, video stabilization using object tracking is not suitable if the object of interest is displacing through a large distance. There are three major algorithms for object tracking such as Meanshift, Camshift and Background Subtraction. We shall discuss each of these algorithms in brief.

2.2 OBJECT TRACKING BASED VIDEO STABILIZATION

2.2.A MEANSHIFT:

The intuition behind the Meanshift is simple. Consider you have a set of points, it can be a pixel distribution like histogram backprojection(It basically replace every pixel by its probability to occur in the image. It is not really useful to use it by hand, but this is used by other algorithms like Meanshift)[14]. You are given a small window (maybe a circle) and you have to move that window to the area of maximum pixel density (or the maximum number of points). It is illustrated in Fig 6.

The initial window is shown in the blue circle with the name "C1". Its original center is marked in blue rectangle, named "C1_o". But if you find the centroid of the points inside that window, you will get the point "C1_r" (marked in the small blue circle) which is the real centroid of the window. Surely they don't match.

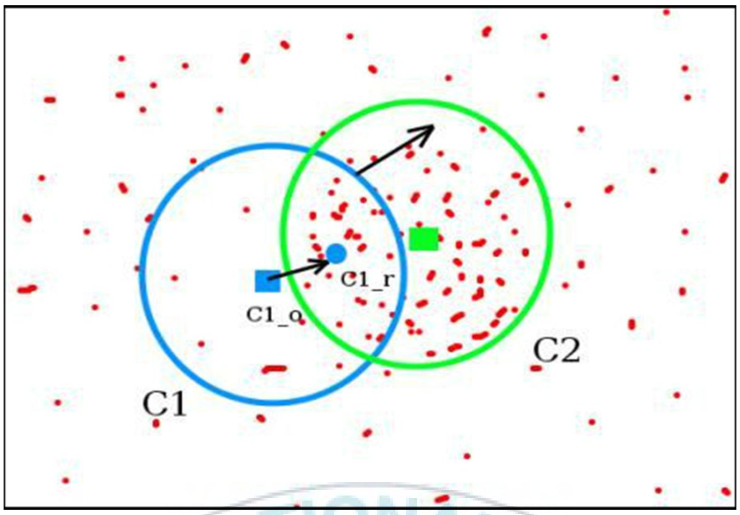


Fig 6. Meanshift centroid based window shift[16].

So move your window such that circle of the new window matches with the previous centroid. Again find the new centroid. Most probably, it won't match. So move it again, and continue the iterations such that center of the window and its centroid falls on the same location (or with a small desired error). So finally what you obtain is a window with maximum pixel distribution. It is marked with a green circle, named "C2".

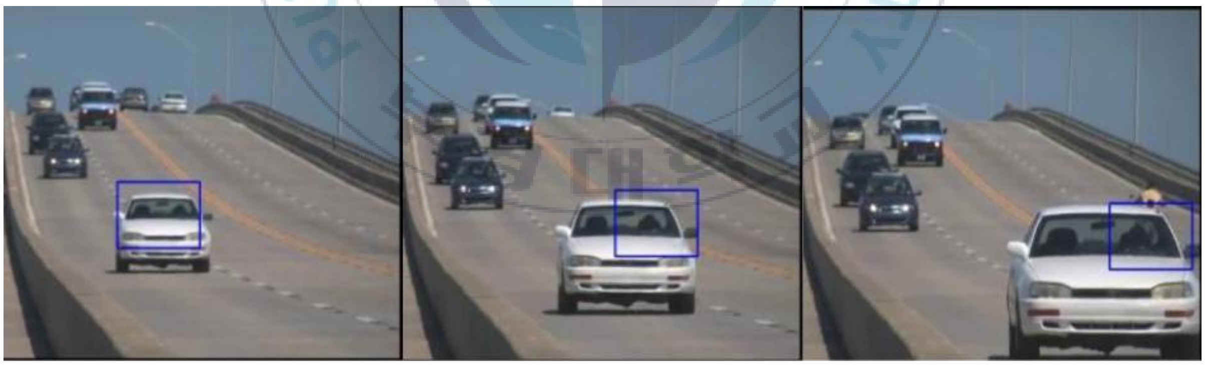


Fig 7. Meanshift based object tracking

Fig 7. shows object tracking using Meanshift algorithm but there is a major problem with this algorithm: window always has the same size when the car is farther away and it is very close to the camera. That is not good. We need to adapt the window size with the size and rotation of the target.

2.2.B CAMSHIFT:

Camshift also uses Meanshift algorithm in its initials steps, but to dynamically vary the object tracking frame size, Camshift updates the window size once Meanshift converges by calculating the zeroth moment of the existing window size as shown in equation Eq 1[15]. Camshift also calculated the orientation of the best fitting ellipse.

$$s = 2\,x\,\sqrt{\frac{M_{00}}{256}} \qquad \ldots \text{ Eq 1}$$

Camshift result is shown in Fig 8, as we can see the tracking window size varies according to the size of the region of interest.

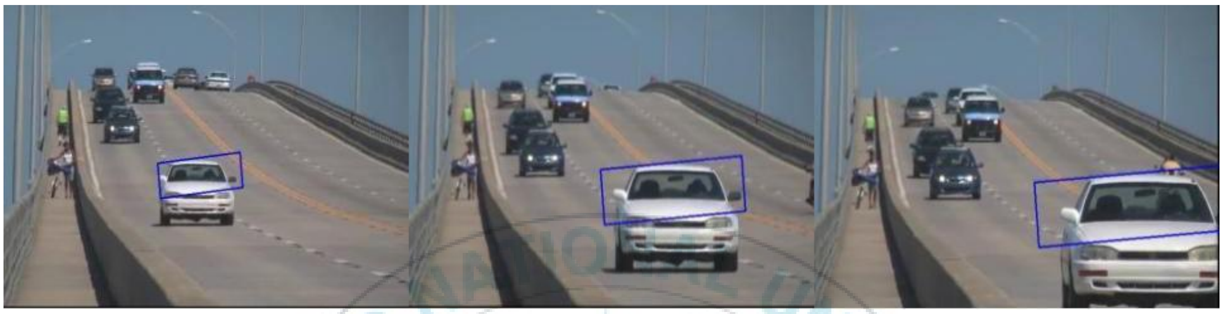


Fig 8. Camshift based object tracking

2.2.C BACKGROUND SUBTRACTOR

Background subtractor is a very widely used method for image segmentation, to separate foreground from background. Background subtractor can be used to track a moving object in a video and then apply video stabilization to this object such that it maintains its position throughout the stabilized video. Majorly there are four methods of background subtractor, each having the same basic algorithm with some modification, these four methods are - BackgroundSubtractorMOG, BackgroundSubtractorMOG2, BackgroundSubtractorGMG, BackgroundSubtractorKNN. As the focus of this thesis is not on video stabilization using object tracking we shall only discuss the basic idea behind background subtractor.

Background subtractor basically takes a series of frames and subtracts the consecutive frame, giving us a binary image denoting the moving foreground objects. Below is Fig 9 showing the binary image obtained after background subtraction.

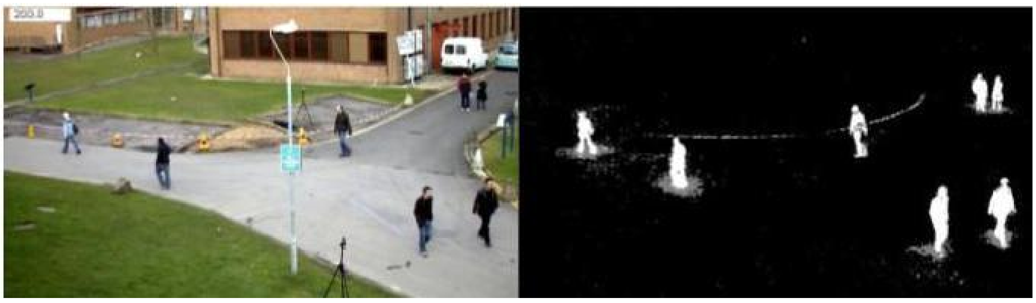


Fig 9. Left: Original Image; Right: Background subtracted binary image using the MOG2 technique

Most widely used technique due its better results is MOG2[17][18]. This algorithm selects the appropriate number of Gaussian distribution for each pixel. It provides better adaptability to varying scenes due to illumination changes etc. MOG2 also has the provision to detect the shadow of the moving object, in Fig 9, the shadow of foreground objects is denoted by gray color.

2.3. FRAME MOTION ESTIMATION BASED VIDEO STABILIZATION USING OPTICAL FLOW:

2.3.A PRINCIPLE:

Feature detection and minimizing the distance between identical feature across consecutive frames to produce steady video, having reduced jitter than the original video.

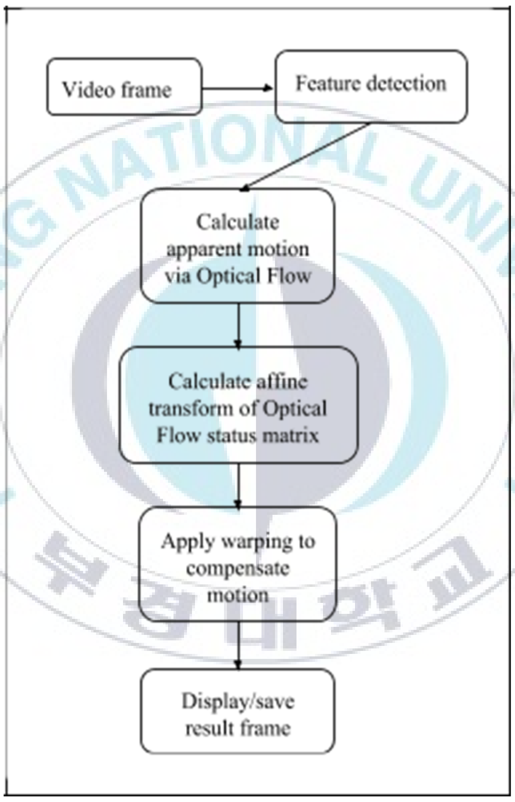

Video frame
Feature detection
Calculate apparent motion via Optical Flow
Calculate affine transform of Optical Flow status matrix
Apply warping to compensate motion
Display/save result frame

Fig 10. Video Stabilization using Optical Flow algorithm block diagram

First, a single video frame from a series of frames is taken, features in that frame are extracted, these features are the key points to the Optical Flow algorithm. Optical Flow algorithm analyses the motion of these detected features across consecutive video frames and determine the net motion of these features. To spatially stabilize the video we need to warp the image matrix such that these detected features will maintain their position throughout the video. In this thesis, we will focus on Optical Flow based video stabilization. We shall see the methodology of Optical Flow based video stabilization in the next chapter, Chapter 3.

2.4 BRIEF COMPARISON OF VARIOUS FEATURE DETECTION TECHNIQUES

Features in an image are of two types: local and global. Local features are extracted by segmenting the image into patches, while for global feature extraction, the whole image is considered. Some examples of local features are blobs, corners, and edges. Contours, HOG, Invariant Moments are some examples of global features[25].

For video stabilization application we shall deal with local features. In this section, we shall look at a comparison of algorithms to detect local features. For this comparison we shall consider the following aspects:

1. Noise immunity
2. Detection cost in time
3. Immunity against change in light intensity

After the comparison, we shall select the algorithm which will be most suitable for our needs for video stabilization. Below are the test images:

Fig 11. From left to right: 1. Mandril, 2. Barbara, 3. Lena, 4. Peppers

After the comparison we shall select the most relevant feature detection algorithm having the best performance on an average among the three test categories mentioned above.

For noise immunity, the algorithm having the least number of detected features in noisy image is better. For second test category, the algorithm having the lowest time to find features is better. For the third category, the algorithm having the least drift when light intensity change is considered to be better.

2.4.A NOISE IMMUNITY

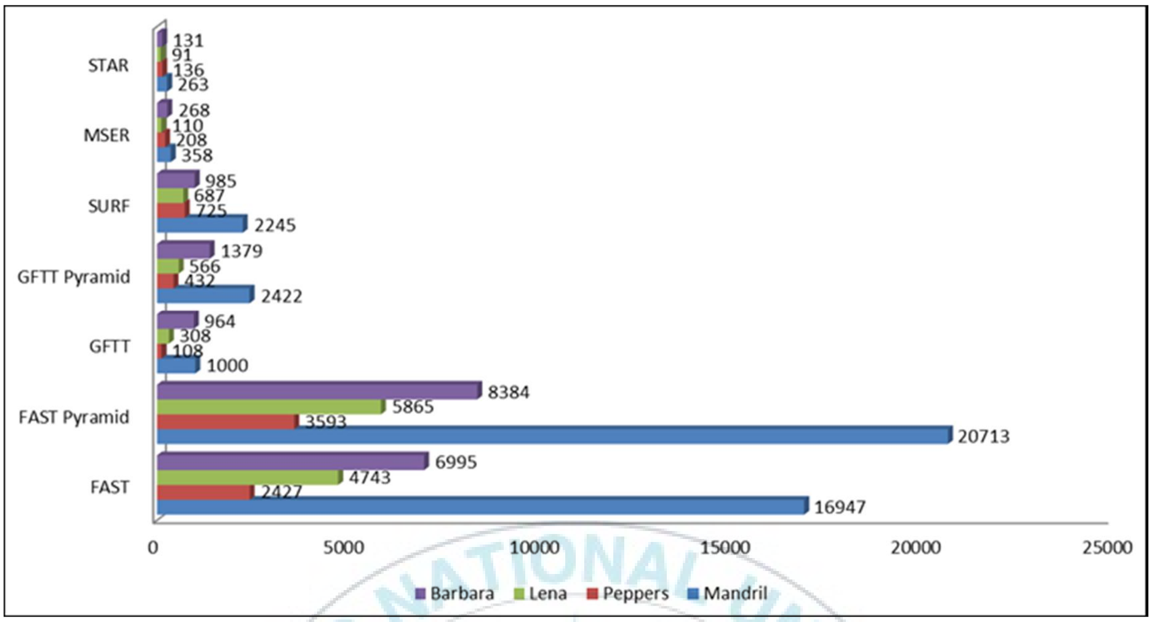


Fig 12. Comparison of the number of features detected on basis of noise immunity(lower is better)

From the graph, it is concluded that STAR has the highest immunity against noise while FAST/FAST pyramid performs very poor on a noisy image. While GFTT also performs very good against a noisy test image.

2.4.B DETECTION COST IN TIME

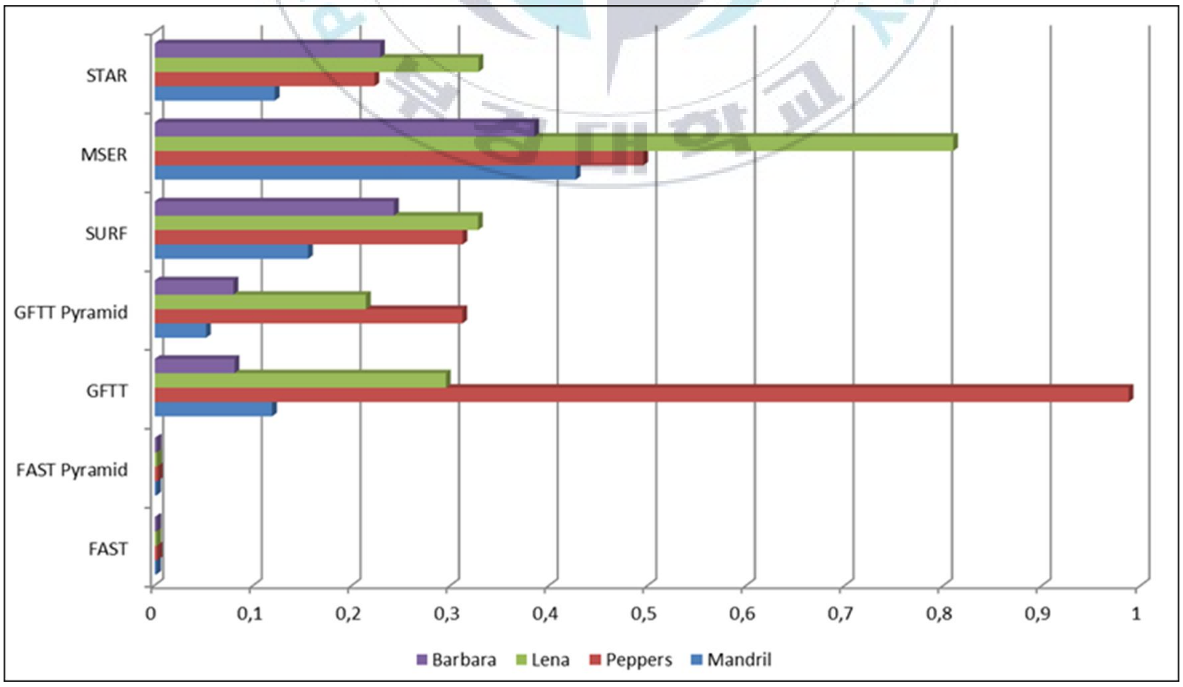


Fig 13. Comparison of the number of features detected on basis of computational cost in ms (lower is better)

From the above graphs, it is inferred that FAST takes least time to extract features from images. GFTT performs poorly for Peppers image. This time can still be optimized by reducing the highest number of detectable features and by increasing the distance between features.

2.4.C IMMUNITY AGAINST CHANGE IN LIGHT INTENSITY

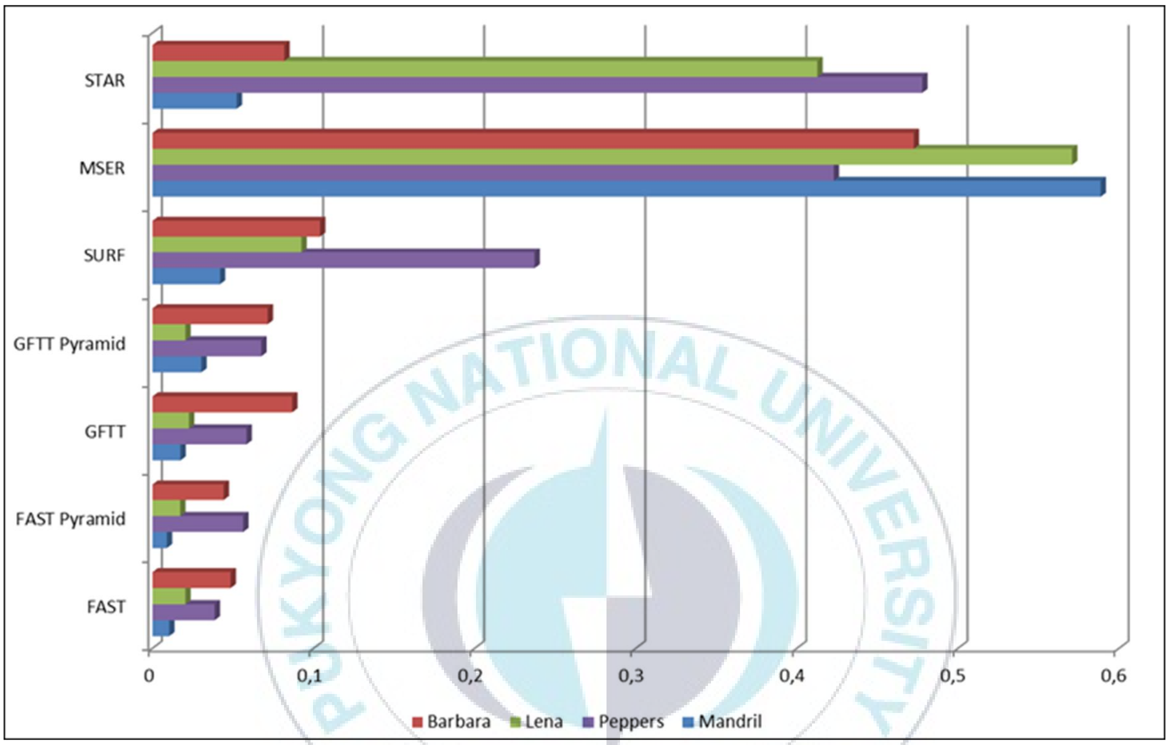


Fig 14. Comparison of the number of features detected on basis of immunity against light intensity change(lower is better)

This test can be considered as the most important test for deciding which features detector we shall use in our video stabilization application. From the graph above, it can be inferred that STAR has the highest drift on changing the light intensity while FAST has the minimum. While GFTT performance against light intensity change is also impressively good. For applying video stabilization on real time and outdoor video, it is better if the features detector has high resistance against light intensity change.

From the above comparison, we have selected GFTT as the most suitable algorithm for feature extraction in our video stabilization program, due to the following reasons:

1. Better immunity to noise
2. Invariance to light intensity change.
3. Relatively low computational cost

CHAPTER 3

METHODOLOGY

3.1 FEATURE DETECTION

Features in an image are the distinct regions in an image such as the pattern of colors, corners, edges and blobs which can be used to extract information from the image[4]. Good features should have the following properties[5]:

1. *Repeatability:* Irrespective of the angle of viewing the object of interest, good features are the one which is common to maximum viewing angles. Repeatability of features in an image also implies more robustness over uncertain image deformation.
2. *Distinctiveness:* A features should show maximum variation than image region in its vicinity to call it as a good feature.
3. *Locality:* A feature should not be affected by problems of occlusion caused by viewing the object of interest from different viewing angles.
4. *Quantity:* Number of features in an image should be ample enough to be tracked without being affected by uncertain losses in an image.

Here, we will focus on tracking "good" corners in video frames.

3.2 A BRIEF ON HARRIS CORNER DETECTION

Corners in an image can be defined as the regions in the image having a large variation in pixel value in all directions. In 1998 Chris Harris and Mike Stephens[6] came up with a mathematical representation of this idea. All possible directions around single pixel can be covered by making a shift of 45° (each pixel except border pixel has 8 neighbors) starting from 0° to 360° around a pixel, represented by the following equation Eq (2):

$$E(u, v) = \Sigma(x, y) \, . \, [I(x + u, y + v) - I(x, y)]^2 \qquad \ldots (2)$$

Where $\Sigma(x, y)$ is called the window function, it gives weight to each pixel being scanned, $I(x + u, y + v)$ gives the shifted intensity, u is the shift in the x-direction and v is the shift in y-direction, $I(x, y)$ is the Intensity of the pixel being scanned.

For corner detection, we need to maximize this function. Applying Taylor series to this function gives us equation Eq (3):

$$E(u, v) = [u\ v]\ M\ [u\ v]^T \qquad \ldots(3)$$

Where M is represented by equation Eq(4):

$$M = \Sigma w(x, y)\ .\ \begin{bmatrix} IxIx & IxIy \\ IxIy & IyIy \end{bmatrix} \qquad \ldots (4)$$

Then Chris Harris and Mike Stephens came up with a score function which determines whether a window can contain a corner or not, this function is given in equation Eq(5):

$$R = det(M) - k(trace(M))^2 \qquad \ldots.(5)$$

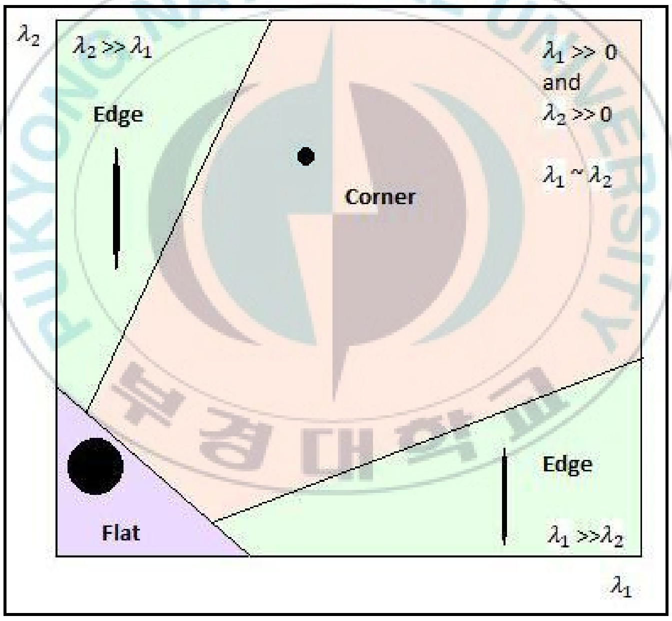


Fig 15. Representation of the Harris corner model output based on the value of λ_1 and λ_2 [8]

Where det(m) = $\lambda_1\lambda_2$, trace(M) = $\lambda_1 + \lambda_2$ and λ_1 and λ_2 are the eigen values of *M*.

Thus the eigen values of *M* decide whether the region is flat, corner or edge, Fig 11 shows the representation of the effect of these eigen values on the results.

3.3 MODIFICATION TO HARRIS CORNER DETECTION

In 1994, J. Shi and C. Tomasi made a small modification to Harris corner detection algorithm and simplified the algorithm. They modified the equation Eq(5) for Harris corner detection and simplified it to equation Eq(6):

$$R = min(\lambda_1, \lambda_2) \qquad \ldots(6)$$

Thus the representation in figure Fig 15 is modified to figure Fig 16:

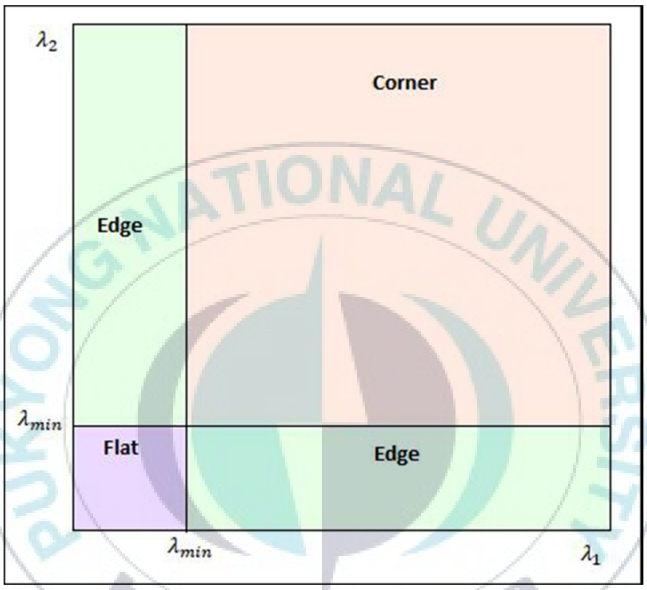


Fig 16. Good Features to Track output based on eigen values of M [8]

This algorithm is proven to have better results than its predecessor[7]. Thus here, we will be using this algorithm to track features in a video frame.

OpenCV function of this method is given by:

*cv::void **goodFeaturesToTrack**(InputArray **image**, OutputArray **corners**, int **maxCorners**, double **qualityLevel**, double **minDistance**, InputArray **mask**=noArray(), int **blockSize**=3, bool **useHarrisDetector**=false, double **k**=0.04)*

3.4 OPTICAL FLOW

Optical Flow assumes the following two conditions:

1. Pixel densities of an object do not change across consecutive video frames.
2. Neighboring pixels of a moving object has similar motion as the moving object.

Equation eq(7) represents the first assumption[9] in mathematical form:

$$I(x, y, t) = I(x + dx, y + dy, t + dt) \qquad \ldots(7)$$

where I represent the intensity of a pixel, x,y are spatial coordinates of the pixel, t is the temporal advancement of the pixel in video frames, dx, dy, dt represents a small change in x, y, t respectively. On application of Taylor series expansion to first assumption equation Eq(7) then removing the common terms and dividing the whole equation with d_t we get equation Eq(8)[9]:

$$f_x u + f_y v + f_t = 0 \qquad \ldots(8)$$

Where,

$$f_x = \frac{\delta f}{\delta x} \;;\quad f_y = \frac{\delta f}{\delta y}$$

$$u = \frac{dx}{dt} \;;\quad v = \frac{dy}{dt}$$

Equation Eq(8) is called the Optical Flow equation but this equation is under-constraint (one equation two unknown variables) and cannot be solved. There are several methods to solve this equation, one of the method is proposed by B. D. Lucas and T. Kanade.

Lucas-Kanade method took into consideration the second assumption that is optical flow should be equal for all neighboring pixel of a single pixel so 9 equations were obtained from 3x3 pixel matrix(8 pixels surrounding one pixel). Lucas-Kanade system is over constraint system(9 equations, two variables), thus solvable.

After applying the least square fit method[9][10] we obtain equations eq(9) and eq(10):

$$\Sigma f^2_{xi} u + \Sigma f_{xi} f_{yi} v = -\Sigma f_{xi} f_{ti} \qquad \ldots(9)$$

$$\Sigma f_{xi} f_{yi} u + \Sigma f^2_{yi} v = -\Sigma f_{yi} f_{ti} \qquad \ldots(10)$$

For small motions, this algorithm works perfectly but fails when large motion occurs and gives arbitrary results because derivatives used in the method assume a small change in motion. This problem is solved by implementing the pyramid method to Lucas-Kanade method. Pyramid method scales down the number of pixel motion for the derivatives to give correct results[10][11]

OpenCV function for Optical Flow:

void calcOpticalFlowPyrLK(InputArray prevImg, InputArray nextImg, InputArray prevPts, InputOutputArray nextPts, OutputArray status, OutputArray err, Size winSize=Size(21,21), int maxLevel=3, TermCriteria criteria = TermCriteria (TermCriteria::COUNT + TermCriteria::EPS, 30, 0.01), int flags=0, double minEigThreshold=1e-4)

3.5 AFFINE TRANSFORMATION

Affine Transform is used to make a correction to distortions caused in an image. Fewer distortions in an image are favorable. Affine transform preserves points, parallel lines, and planes during correction.

Affine transform by a 2x3 matrix is denoted by equation Eq(11)[12][8]:

$$A = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}_{2\times 2} \quad B = \begin{bmatrix} b_{00} \\ b_{10} \end{bmatrix}_{2\times 1}$$

$$M = [\,A \quad B\,] = \begin{bmatrix} a_{00} & a_{01} & b_{00} \\ a_{10} & a_{11} & b_{10} \end{bmatrix}_{2\times 3} \qquad \ldots(11)$$

Applying the transform to a 2D vector using A and B we get equation Eq(12):

$$T = M \cdot [x, y, 1]^T \qquad \ldots(12)$$

Applying affine transform of negative rotation and a scale factor to the image in Fig 17, we get the image in Fig 18.

Fig 17. Original image, before affine transform

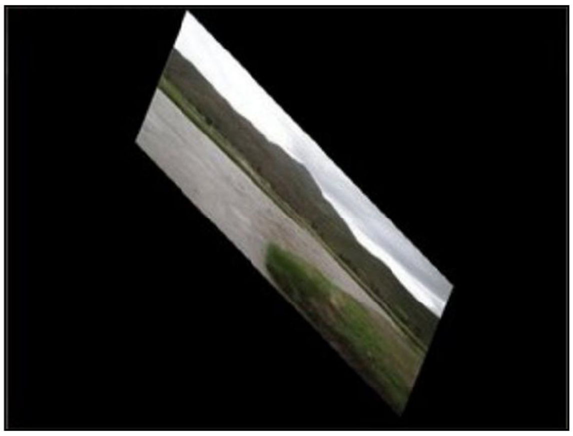


Fig 18. Result image after applying affine transform.

Affine transform is a necessary operation involved in video stabilization. It is used to apply a transform to video frames to maintain detected features at a fixed position across consecutive video frames.

3.6 SYSTEM INDEPENDENT PROGRAM DEVELOPMENT

To run our program on any Linux based system, irrespective of the OpenCV dependency, we have developed a stand-alone application with the file type of *.AppImage*. For many years until now, Linux packages came in two file type extension: *.deb* and *.rpm*; *.deb* only run on Debian based systems and *.rpm* will only run on Fedora based systems. *.AppImage* removes this compatibility issue, it can be executed on both types of Linux systems, Debian as well as Fedora[19].

AppImage can be defined as a universal software package format. By packaging the software in AppImage, the developer provides just one file 'to rule them all'. We packed our program in *.AppImage* format to compare our program performance on various machines. .AppImage can also be executed on a Windows-based machine by using VirtualBox. There are many advantages of *.AppImage*application: high portability, no need of installation, one-click run[19].

Steps to build *.AppImage* are given in this Github repository: https://github.com/probonopd/linuxdeployqt

While packing a program in *.AppImage* format, all the libraries that the program requires(OpenCV libraries) get linked to it so that while executing the application, it does not scan the host system for its dependencies, instead, it fetches the dependencies from the packed *.AppImage*itself.

3.7 SETTING STANDARDS FOR TESTING

For testing our program, we took many cases into consideration to get a generalized idea about the performance of our program on various systems. We had two systems for testing our program:

1. Intel® Core ™i7 CPU 860 @ 2.80GHz, RAM: 8GB.
2. Intel® Core ™i7 4750HQ @3.2GHz 8 cores, RAM: 16GB, GPU: Nvidia GTX 950M

We tested our program on these video resolution and FPS:

1. Live, 320 x 240 @ 30FPS
2. Live, 320 x 240 @ 100FPS
3. Live, 640 x 480 @ 30 FPS
4. Live 1280 x 720 @ 15 FPS
5. Live 1280 x 720 @ 30 FPS
6. Offline, 1920 x 1080 @ 30FPS

Thus we get numerous scenarios to test our program on. Live implies real-time camera input while offline means taking recorded video as input. Apart from the standard video input, we also tested our program on a single image by executing our program repetitively on this image, to get an idea of maximum FPS count our video stabilization program can process. We shall discuss the result in the next chapter; chapter 4, experimental results.

CHAPTER 4

EXPERIMENTAL RESULTS

4.1 GRAPHICAL RESULT OF VIDEO STABILIZATION

As discussed earlier in Chapter 3, section 3.7, in this chapter we shall discuss the resulting output video specification. Before proceeding, let's discuss the video stabilization quality by plotting a graph of average dx, dy of each pixel value of a video against video frame index.

4.1.A PROCEDURE FOLLOWED FOR PLOTTING THE GRAPH

We first accumulated the coordinates of detected features in a video, then found out the mean of all the coordinated throughout the video across all frames, let's call this mean as global mean. Then for each frame, we found the mean of all the detected features, let's call this mean as local mean then compared it with the global mean by finding out the difference between the local mean and global mean, thus we found dx and dy at each frame, let's arrange this procedure in mathematical equations:

Following equation, Eq(13) and Eq(14) describes calculation for the local mean of x and y coordinates of detected features respectively while Eq(15) and Eq(16) describes calculation for the global mean of x and y coordinates of detected features respectively

$$\overline{x_l} = \left(\sum_{i=0}^{n} P\, x_i\right) / n \qquad \ldots(13)$$

$$\overline{y_l} = \left(\sum_{i=0}^{n} P\, y_i\right) / n \qquad \ldots(14)$$

$$\overline{x_g} = \left(\sum_{i=0}^{N} \overline{x_l}\, i\right) / N \qquad \ldots(15)$$

$$\overline{y_g} = \left(\sum_{i=0}^{N} \overline{y_l}\, i\right) / N \qquad \ldots(16)$$

Where,

$\overline{x_l}$ is the local mean of x coordinates, $\overline{y_l}$ is the local mean of y coordinates of detected features; $\overline{x_g}$ is the global mean of x coordinates, $\overline{y_g}$ is the global mean of y coordinates of detected features;

$P\,x$ is the x coordinate of the detected feature, $P\,y$ is the x coordinate of the detected feature; n is the total number of detected features in one frame; N is the total number of frames in a video.

Now we have the local mean and global mean, to plot graph we need d_x and d_y . Following equations, Eq(17) and Eq(18) show us how to find d_x and d_y .

$$d_{xi} = \overline{x_g} - \overline{x_{li}}$$

$$d_{yi} = \overline{y_g} - \overline{y_{li}}$$

Where,

d_{xi} is the deviation of the local mean of tracked features from their global mean in the x-direction

d_y is the deviation of the local mean of tracked features from their global mean in the y-direction

i is the frame number

These coordinates were stored in a file and the graphs were plotted by using python Matplotlib package.

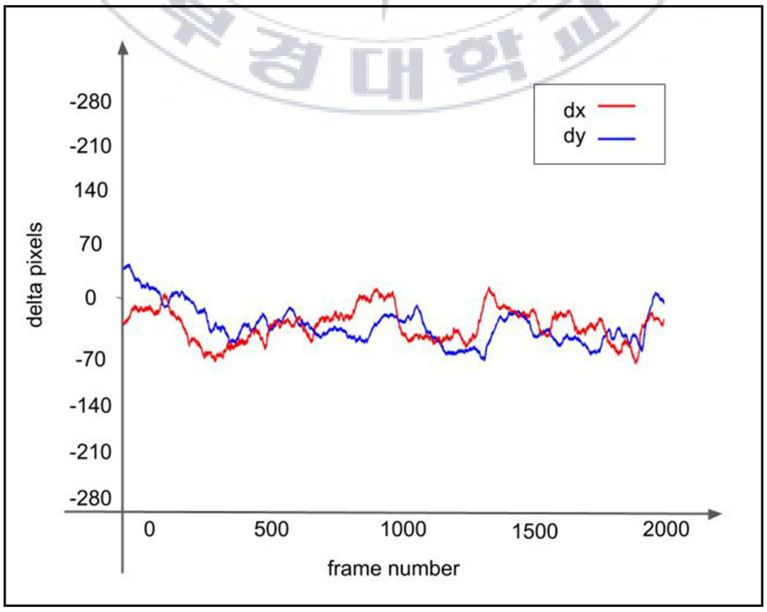


Fig 19. dx and dy of original video against frame number.

From the graph in Fig 15, we can conclude that the average displacement of pixels in the video frame is very random and very noisy. The aim of our program is to diminish the average displacement of each pixel in both x and y-axis, to make the graph look smooth.

Thus after the application of the video stabilization program, the resulting graph of average displacement of each pixel in x and y-direction is shown in Fig 16.

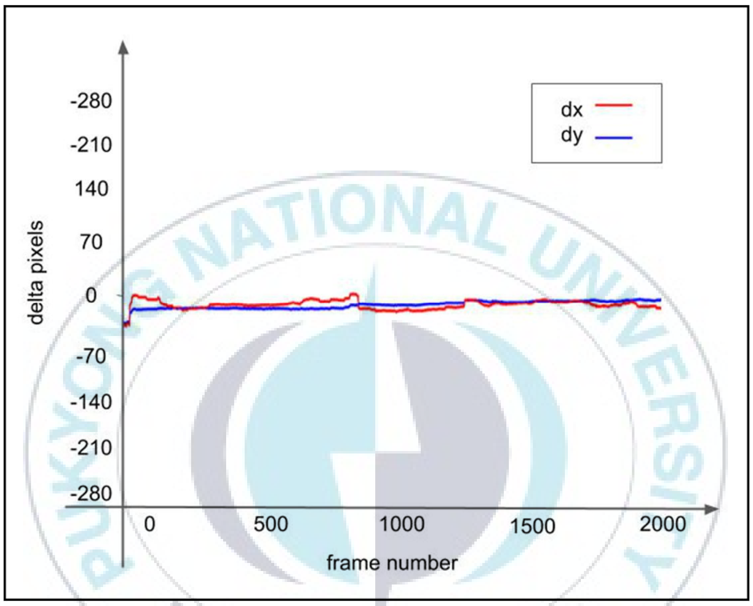


Fig 20. dx and dy of stabilized video against frame number.

From the graph shown in Fig 16, it is concluded that the average displacement of each pixel in x and y-direction is reduced drastically with greatly reduced noise. Thus it is proved, the video stabilization program reduces the jittery motion of each pixel and gives a smooth output video. For a good video stabilization program, the deviation of the tracked features should be very less from 0 value.

4.2 TABULAR REPRESENTATIONS OF RESULTS OF DIFFERENT SCENARIOS

4.2.A PROCEDURE TO MEASURE FPS

First, we initialized a timer at the start of our video stabilization program. In the program loop, we used two variables to store the time at the start of the loop and one at the end of the loop. By finding the difference between these two times, we get the time required for the execution of one loop.

Frame per second of a video stream can be defined as the frequency at which it displays frames, i.e. how many frames can the video stream display in one second. Thus to calculate the fps at which our program can process video stabilization we need to find how many frames can our program process in one second.

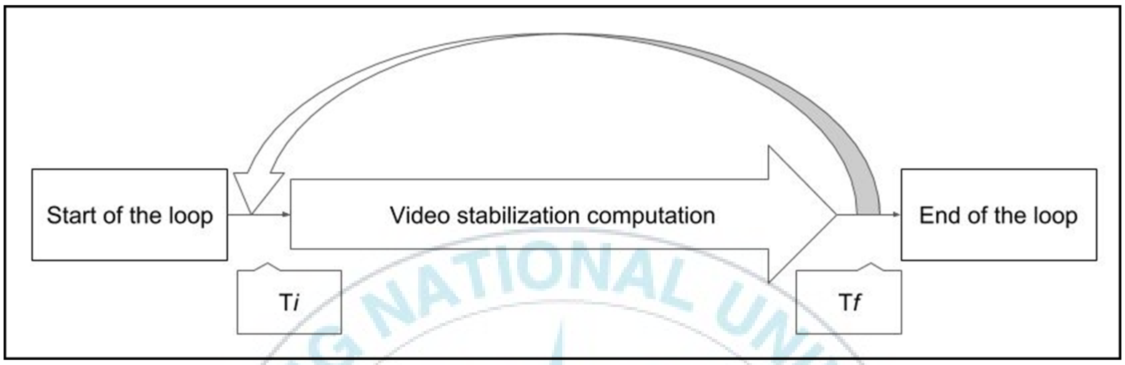


Fig 21. Diagram showing time measure at start and end of the loop

Let us denote FPS calculation by following equations, Eq(17) and Eq(18):

$$\text{FPS} = F_n / \quad t \qquad \text{....(17)}$$

Therefore,

$$\text{FPS} = 1 / \quad t / F_n \qquad \text{....(18)}$$

$$t = T_f - \quad T_i \qquad \text{....(19)}$$

Where,

FPS is frame per second;

F_n is the number of frames processed in time = t seconds

t is the time difference between system time at the start of the loop(T_i) and that at the end of the loop(T_f)

Now, let us consider an example for calculating the frame per second a program can process:
If $t = 25497\mu s$

As we are considering t for one loop, therefore F_n for one loop = 1.
By equation (18), we get FPS = 1 / 25497μs / 1 = 1000000 / 25497 = 39.220 FPS

We have tested our calculation with pre-recorded videos and our calculated FPS matches with the known FPS count of the pre-recorded video, We shall now discuss the results of our program in different testing scenarios as discussed in Chapter 3, section 3.7.

By testing our program in different scenarios of the input video mention in chapter 3 section 3.7, we get the following result:

S. No.	System no.	Input video resolution	Input video FPS	Output video resolution	Output video FPS
1	1	320 x 240	30	320 x 240	27
2	2	320 x 240	30	320 x 240	30
3	1	320 x 240	100	320 x 240	72
4	2	320 x 240	100	320 x 240	98
5	1	640 x 480	30	640 x 480	24
6	2	640 x 480	30	640 x 480	30
7	1	1920 x 1080	30	1920 x 1080	13
8	2	1920 x 1080	30	1920 x 1080	28

Table 1. Our Video stabilization program results in different scenarios

From the table, we can conclude that frame per second of the output video is greatly affected by the system configuration. As mentioned in chapter 3, section 3.7, the system configuration of system number 2 is far better than that of system number 1[24]. So it is natural to expect high frame per second output from system number 2.

One more parameter to be considered for testing is how many features are being tracked by our video stabilization program, In Table 1 the maximum trackable features is set to 300 with minimum distance between them as 4 pixels, the table below, Table 2, shows the result of this experiment in which we have varied the maximum features that our program can track.

S No.	Maximum trackable features	Minimum distance between two features	Input video specification	Output video resolution	Output video FPS
1	50	4	1280 x 720 @ 15 FPS	1280 x 720	15

2	100	4	1280 x 720 @ 15 FPS	1280 x 720	15
3	300	4	1280 x 720 @ 15 FPS	1280 x 720	15
4	600	4	1280 x 720 @ 15 FPS	1280 x 720	15
5	10000	4	1280 x 720 @ 15 FPS	1280 x 720	14
6	50	4	1920 x 1080 @ 30 FPS	1920 x 1080	30
7	100	4	1920 x 1080 @ 30 FPS	1920 x 1080	30
8	300	4	1920 x 1080 @ 30 FPS	1920 x 1080	28
9	600	4	1920 x 1080 @ 30 FPS	1920 x 1080	28
10	10000	4	1920 x 1080 @ 30 FPS	1920 x 1080	28
11	100	1	1280 x 720 @ 15 FPS	1280 x 720	15
12	10000	1	1280 x 720 @ 15 FPS	1280 x 720	14
13	100	1	1280 x 720 @ 30 FPS	1280 x 720	30 FPS
14	10000	1	1280 x 720 @ 30 FPS	1280 x 720	28 FPS
15	100	1	1920 x 1080 @ 30 FPS	1920 x 1080	29 FPS
16	10000	1	1920 x 1080 @ 30 FPS	1920 x 1080	25 FPS

Table 2. Experimental results obtained by varying maximum number of trackable features and distance between them

From Table 2, we can conclude that there is a slight variation in output FPS when we vary the maximum number of trackable features, but there is a significant change when the distance between adjacent features is reduced to 1 pixel, as seen in Table 2, Scenario 16, there is a significant drop in FPS, 5 FPS.

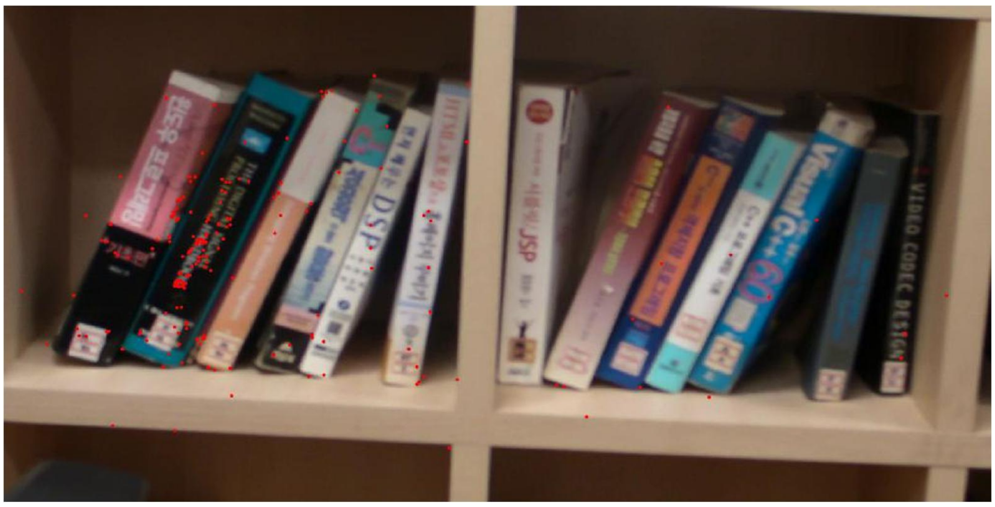


Fig 22. Scenario 4 result frame, red dots indicate the detected features

The actual quality of the stabilization was poor when the number of features is reduced to around below 100. But we saw no significant improvement in video stabilization quality when we increased the maximum number of trackable features beyond 300, so we can neglect the degraded performance of our program for a very high number of maximum trackable features such as Table 2, Scenario 16.

Fig 17. shows one frame from our test video showing very few features being tracked that is 50, the result of this scenario was poor with very less stabilization intensity. Among the three figures, 17, 18, 19, the scenario in Fig 18 showed the best result with good video stabilization quality as well as good FPS.

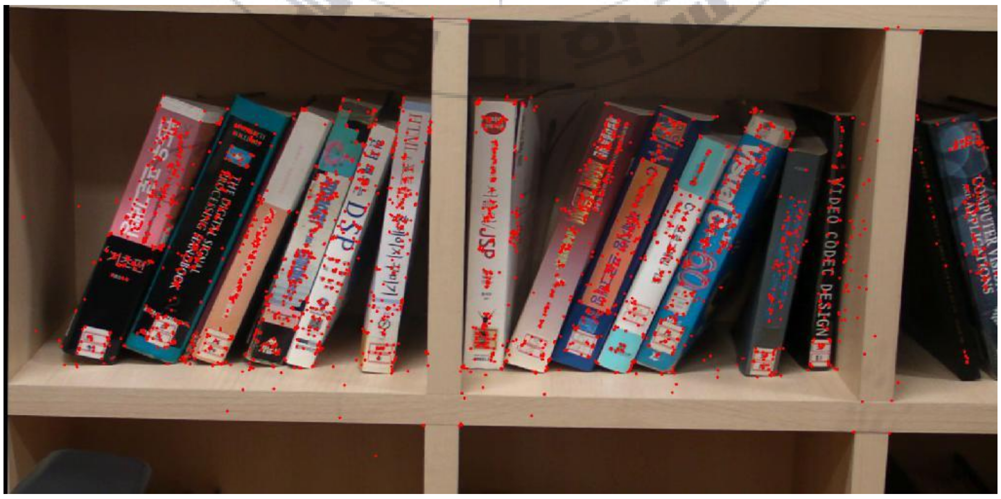


Fig 23. Scenario 10 result frame, red dots indicate the detected features

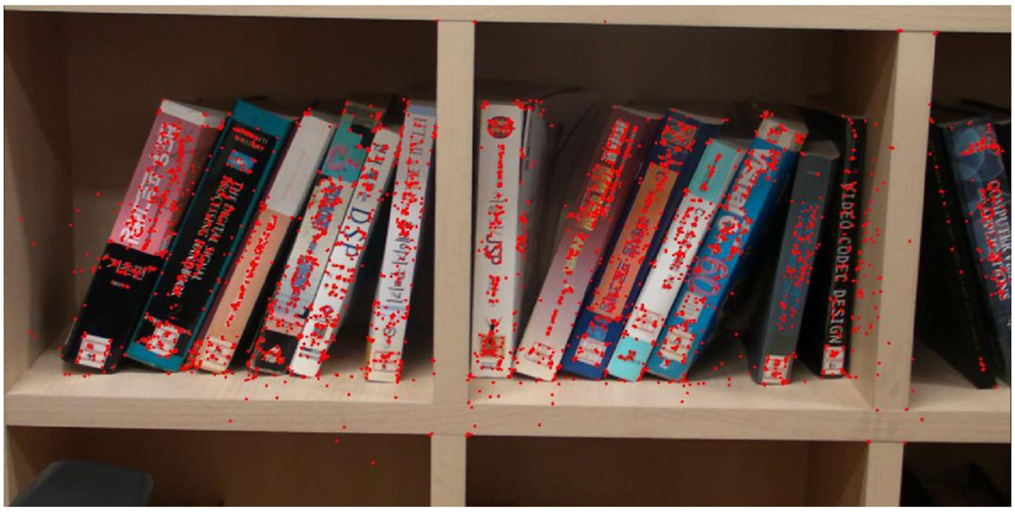


Fig 24. Scenario 16 result frame, red dots indicate the detected features

The last step of our video stabilization algorithm is warp affine, due to the large motion of the features being tracked, caused large black borders around the image, but this can be easily be avoided using image cropping, we purposefully haven't implemented image cropping in our present program to test the performance of video stabilization only.
One more factor affecting the performance of the program is the time taken for the camera to communicate with our program, this time will vary from camera to camera and system to system. In our experiment, we have calculated the time taken for the camera to communicate with the program by using the Chrono library in C++, by using two timers, one at the very start of the program loop and one after accepting frame input from the camera or a video stream.

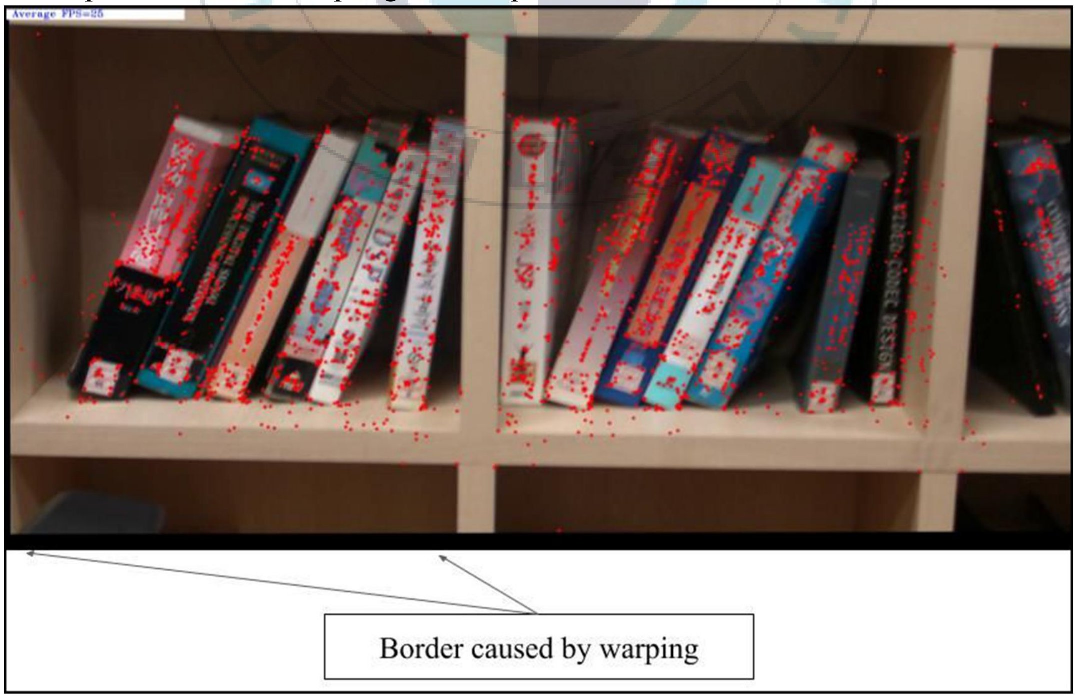


Fig 25. Undesirable borders created due to warp affine transformation

On an average, the time taken for the camera to transfer an image to our program is 5827 microseconds. So we made a new scenario to test our program by dropping this system dependent factor by passing the same image to the loop at each loop execution, this transfer of a pre recorded image to the loop took 0 microseconds. By doing so we can obtain the maximum FPS that our program can reach a particular resolution. The table below, Table 3 shows the maximum FPS reached by our program at different resolutions, in this scenario, the maximum trackable feature is set to an optimum value of 300 at 4 pixels apart. All the testings carried out in the table below were on system number 2.

Resolution of the input image	Resolution of the output image	FPS of the output image
320 x 240	320 x 240	312
640 x 480	640 x 480	198
1280 x 720	1280 x 720	116
1920 x 1080	1920 x 1080	57

Table 3. Experimental results obtained by passing a single image to our program at each loop execution

The table below, Table 4 shows the system usage of System number 2 at various scenarios mentioned in table number 3.

Resolution of the input image	CPU usage	RAM usage
320 x 240	15 %	22.9 MB
640 x 480	23 %	30.6 MB
1280 x 720	28 %	48.6 MB
1920 x 1080	28 %	82.2 MB

Table 4. System usage of scenarios mentioned in table number 3.

From the table above, Table 4, we can conclude that the RAM usage is more when input video resolution is more while CPU usage is not much affected by varying input video resolution. Thus to process very high definition videos the host machine should have a fast(more RAM clock speed) and large RAM.

Thus, we have performed a rigorous testing of the video stabilization program over various scenarios and noted down our results in form of output video resolution and output video FPS, further, we have also noted down our system usage while running our program. By introducing parallelism in our program system resources can be used more efficiently, we shall discuss the scope of improvement in our video stabilization software in chapter 6.

4.2.B COMPARISON WITH OTHER OPEN SOURCE VIDEO STABILIZATION PROGRAMS

For making a comparison between our program and other programs we used video stabilization program by Nghia Ho[23]. Nghia Ho primary program does not work on real time camera feed instead it needs pre-recorded video as input while our program works on a real-time video feed from a camera as well as on pre-recorded video. Let us look at a comparison table based on different video resolutions and FPS. For making the comparison we tested our program on the same system: system number 2 as mentioned in chapter 3 section 3.7.

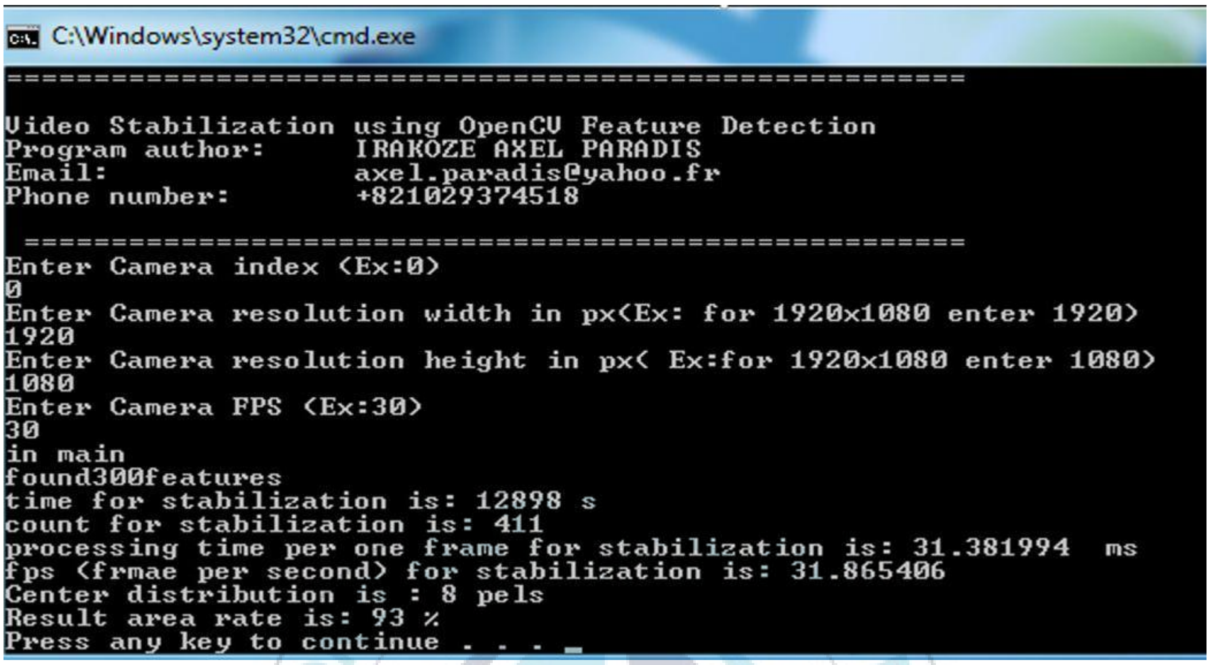
S no.	Input Video specs	Nghia Ho's program Output Video FPS	Our program output video FPS
1	320 x 240 @ 30FPS	30	30
2	640x480 @ 30FPS	29	30
3	1280 x 720 @ 30 FPS	24	29
4	1920 x 1080 @30 FPS	14	28

Table 5. Comparison results with other open source video stabilization program

From the table it is clear that our program gives a better result than other open source programs for video stabilization, this is due to the fact our program stabilizes the video based on only optical flow parameters, while other programs also use Kalman Filter for video stabilization.

Thus, our program is convenient for performing real-time video stabilization on low-performance machines. The performance of our program can be drastically improved on slow machines by using multithreading and parallelism methods by Intel TBB or Nvidia CUDA, we shall discuss these improvements in Chapter 6.

4.3 SCREENSHOT OF OUR PROGRAM'S OUTPUT TERMINAL



```
=======================================================

Video Stabilization using OpenCV Feature Detection
Program author:      IRAKOZE AXEL PARADIS
Email:               axel.paradis@yahoo.fr
Phone number:        +821029374518

 =======================================================
Enter Camera index (Ex:0)
0
Enter Camera resolution width in px(Ex: for 1920x1080 enter 1920)
1920
Enter Camera resolution height in px( Ex:for 1920x1080 enter 1080)
1080
Enter Camera FPS (Ex:30)
30
in main
found300features
time for stabilization is: 12898 s
count for stabilization is: 411
processing time per one frame for stabilization is: 31.381994  ms
fps (frmae per second) for stabilization is: 31.865406
Center distribution is : 8 pels
Result area rate is: 93 %
Press any key to continue . . . _
```

Fig 26. Screenshot of our program's output terminal

CHAPTER 5

PROGRAM DOCUMENTATION

We developed our program in C++ language due to its better speed than python, the reason is that python program execution is interpreter based while C++ has compiler based execution. For using image processing function, we have used the most widely used library family: OpenCV 3.4. We have tested our program on both Windows as well as Linux based machines.

5.1 STEPS INVOLVED IN OUR PROGRAM

1. Accept input video.
2. Find Corners in each frame of the video. Corners in a frame provide us with distinct features in a video frame which are "trackable".
3. Store these corners in an array/vector.
4. Calculate the apparent motion between identical corners/blocks of two consecutive frames using Optical Flow.
5. Calculate affine transform of the status matrix formed by Optical Flow. This gives us the mathematical representation of the apparent motion of the identical corner/block between consecutive frames.
6. Apply warping to the matrix such that the affine transform across multiple frames remains close to constant.
7. Store or show this warped matrix.

5.2 DIFFERENCES BETWEEN OUR PROGRAM AND ALREADY EXISTING OPEN SOURCE VIDEO STABILIZATION:

For this comparison, we are considering Nghia Ho's program[23]. Nghia Ho's program uses Kalman Filter to dynamically eliminate the blank borders around the stabilized video frames, our program excludes this feature because its high requirement of host device's computational power. By excluding this feature, our program saves around 5% of CPU usage(on test system number 2), which makes our program to process higher number of frames in given time as we are targeting our program to run on low performance devices also.

CHAPTER 6

CONCLUSION AND FUTURE RESEARCH

6.1 FUTURE RESEARCH

Presently we have not used multithreading and parallel programming in our program. While parallelizing the video stabilization program will greatly improve the performance on high resolution and high FPS videos. Intel TBB can be used with our program. To take advantage of Intel TBB, we need to build OpenCV with TBB flag equal to 1. Using Intel TBB the performance of our program can be boosted by around 30%[22], that is on Full HD resolution (1920 x 1080) our program can process around 75 FPS than normal 57 FPS.

One more method to improve the program performance is to transfer our program execution on a GPU with CUDA support. Introducing GPU will greatly boost the video stabilization performance by splitting large matrix mathematical calculation in small chunks that can be solved by hundreds of CUDA cores available in GPU. Using a modern GPU boost the image processing performance by around 10 times[13]. But there is one drawback of using Nvidia CUDA, that is our application will be very hardware specific, and will only run on Nvidia powered machine. To solve this issue we can obtain the system configuration programmatically before running the actual CUDA implemented the program and dynamically decide whether to use CUDA and TBB or not based on the support provided by the host machine.

6.2 CONCLUSION

This thesis covers steps involved in video stabilization using Optical Flow with the mathematical representation of each step: Feature detection, Optical Flow using the Lucas-Kanade method and warp Affine transform. In the earlier part of the thesis, we also discuss other video stabilization methods involving the mechanical system and their disadvantages making digital video stabilization a better alternative. At the end of the thesis, we have discussed our system usage while running video stabilization using Optical Flow and the results we obtained in various test cases and how the performance of our program can be improved by using latest technologies such as CUDA and Intel TBB and multithreading.

REFERENCES

[1] Paresh Rawat, Jyoti Singhai, “Review of Motion Estimation and Video Stabilization techniques For handheld mobile video”.

[2] Flashpoint ZeroGrav 2-Axis Digital Gyro Stabilizer.

[3] Serhat Bayrak, “Video stabilization: digital and mechanical approaches”, November 2008.

[4] Ehab Salahat, Member, IEEE, and Murad Qasaimeh, Member, IEEE, “Recent Advances in Features Extraction and Description Algorithms: A Comprehensive Survey”.

[5] Tinne Tuytelaars and Krystian Mikolajczyk, “Local Invariant Feature Detectors: A Survey”.

[6] Chris Harris & Mike Stephens, “A Combined Corner And Edge Detector”, 1988.

[7] J. Shi and C. Tomasi, “Good Features to Track”,1994.

[8] OpenCV 3.0.0-dev documentation.

[9] David J. Fleet, Yair Weiss, “Optical Flow Estimation”.

[10] Dr. Mubarak Shah, UCF Computer Vision Video Lectures 2012, Subject: Optical Flow.

[11] Jean-Yves Bouguet; Intel Corporation, “Pyramidal Implementation of the Lucas Kanade Feature Tracker Description of the algorithm”.

[12] George Bebis, Michael Georgiopoulos, Niels da Vitoria Lobo and Mubarak Shah, “Learning Affine Transformations”.

[13] James Bowley, Blog: “OpenCV 3.4 GPU CUDA Performance Comparison (Nvidia vs Intel)”.

[14] Robin David, Blog: “Chapter 4: Histogram and Backprojection”.

[15] Gary R. Bradski, Microcomputer Research Lab, Santa Clara, CA, Intel Corporation, "Computer Vision Face Tracking For Use in a Perceptual User Interface".

[16] OpenCV video analysis Meanshift and Camshift documentation.

[17] OpenCV video analysis Background subtraction documentation.

[18] Zoran Zivkovic a, Ferdinand van der Heijden, "Efficient adaptive density estimation per image pixel for the task of background subtraction".

[19] It's FOSS, Blog: "How To Use AppImage in Linux [Complete Guide]".

[20] Dhanashree Bhujbal, Prof. B.V. Pawar, "Review of Video Stabilization Techniques Using Block Based Motion Vectors".

[21] OpenCV Team, OpenCV CUDA platform documentation, 2018.

[22] Sagi Zeevi, Blog: "Faster smiles on OpenCV with TBB", published march 23, 2017 · updated October 30, 2017.

[23] Nghia Ho, Blog "Simple video stabilization using OpenCV".

[24] CPU benchmarking at: http://cpuboss.com/cpus/Intel-Core-i7-860-vs-Intel-Core-i7-4750HQ

[25] Ghostwriter examples, Blog "Comparison of the OpenCV's feature detection algorithms"

ACKNOWLEDGMENT

Our effort in work will be incomplete without our beloved supervisor support and knowledge which always helps in getting the easier solution; we would like to express our sincere gratitude to Professor JONG NAM KIM for his guidance, support, and feedback throughout our thesis work. We would always share gratefulness to subjective analysis participants which were a major boost for our thesis work. Everything is incomplete in our lives without our great family who supported all over our career with love and financial support and indeed friends for their ever existing love, affection, and support. We are indebted to our professors, colleagues around in Pukyong National University for their support.

- Axel Paradis IRAKOZE