Real-time Background Subtraction in C++

Haotian Wu, Yifan Yu

May 2, 2010

Boston University

Department of Electrical and Computer Engineering

Technical report No. ECE-2010-06

BOSTON UNIVERSITY

REAL-TIME BACKGROUND SUBTRACTION USING C++

Haotian Wu, Yifan Yu

Boston University
Department of Electrical and Computer Engineering
8 Saint Mary's Street
Boston, MA 02215
www.bu.edu/ece

May 2, 2008

Technical Report No. ECE-2010-06

Summary

Identifying objects of interest from a video sequence is a fundamental and essential part in many vision systems. A common method is to perform background subtraction. For automated surveillance systems, real-time background subtraction is especially important to ensure the performance of the systems. In this paper, we review various background subtraction algorithms in a binary hypothesis test way and compare their performances. We implement several of the algorithms in real-time and get robust, auto-adaptive results of the background subtraction.

Keywords: background subtraction, real-time

Contents

List of Figures

List of Tables

1 Introduction

Identifying objects of interest from a video sequence is a fundamental and essential part in many vision systems, such as traffic analysis systems and human detection and tracking systems. Background subtraction is widely used in these systems. For automated surveillance systems, processing speed may be an essential factor due to security concern. Since background subtraction is often the first step in these applications, real-time and robust background subtraction algorithms are important and need to be explored.

In background subtraction, pixels from objects of interest are considered "foreground" and the rest of the pixels in the frame is considered "background". Usually objects of interest refer to moving objects in the scene while background refers to stable objects. Every pixel in the current frame will either be detected as foreground pixel or background pixel. We can consider this as a binary hypothesis test problem with only two possible hypotheses: background and foreground. The background may not be fixed but must be adapt to several situations to get satisfying results. A robust background subtraction algorithm should adapt to various levels of illumination at different times of the day and can handle gradual and sudden illumination changes. It should also adapt to some motions changes in the scene, such as fluttering leaves, sea waves, waterfalls and camera oscillations. Changes in background geometry such as newly parked cars should also be considered in a robust background subtraction algorithms.

In this paper, we view different background subtraction algorithms in a common hypothesis test way and compare their performances. We implement single Gaussian background model, kernel density estimation (KDE) background model, Markov Random Field model and neighborhood foreground model in real-time and get robust detection results. The paper is organized as follows: the problem statement is in Chapter 2, our implementation of the algorithms and experimental results can be found in Chapter 3 and Chapter 4 separately. Results of our algorithms are shown in Chapter 5. Finally we conclude our paper in Chapter 6.

2 Literature review

Since we assume there are only two possible modes for each pixel in a single frame: background and foreground, we can view the background subtraction problem as a binary hypothesis problem. Based on the current value of the pixel and the probability distributions of two models, we can use a maximum a posterior rule (MAP) to detect the pixel's current mode. Let $I[n]$ be a grayscale image sampled on 2-D lattice $\Lambda : I[n], n \in \Lambda \subset R^2$, and P_B be the prior probability of background, P_F be the prior probability of foreground. We denote a sequence of such images $I^{(k)}[n]$, with k being the frame number. We use $P_B(I^{(k)}[n])$ to denote the background PDF and $P_F(I^{(k)}[n])$ as the foreground PDF. So we get if

$$\frac{P_B(I^{(k)}[n])}{P_F(I^{(k)}[n])} > \eta \frac{P_F}{P_B} \tag{1}$$

then the current pixel is identified as background, if not, the current pixel is identified as foreground. To attain better detect result, we need better estimation of the likelihood of background pixels and foreground pixels, which we define as the models of background and foreground.

A simple way is to do background subtraction is to assume the foreground as uniform when we consider the test. If we cannot get the explicit model of foreground, assuming it as uniform can help us avoid the decision bias. When foreground is considered uniform, $P_F(I^{(k)}[n])$ turns into a constant, so we only need to find out the likelihood probability distribution of background when we perform the hypothesis test. There are various models of background. One of the simplest and most straightforward one is to choose background as the average or median of the former frames [1][2]. However, they usually perform badly due to noise and complicate background situation changes. A single Gaussian model [3] is better, however it also performs poorly when it comes to non-stationary background, such as ceiling fans, swaying tree branches, sea waves and etc. Mixture of Gaussians model [4] usually gives out a more accurate

probability model than the former two. Its shortcomings are the number of Gaussians usually needs pre-defined and some mixtures actually models both foreground and background, resulting confusion in concept and inaccuracy in calculation. A better model to estimate the PDF is to use Kernel Density Estimator [5]. The background PDF is given by the histogram of the past pixel values, each smoothed with a kernel (usually a Gaussian kernel). But it usually requires high computational complexity than the others. All these methods above basically use the information from former frames to establish the model.

Recently, building proper model for foreground is causing more and more attention and proved to be effective in background subtraction. A method [6] proposed by Mike McHugh et al. is to using the small spatial neighborhood information to establish the model of foreground. It is demonstrated in [7] that periodicity in time also holds spatially. So we can establish the foreground model the same way as the background model except we use the information in small spatial neighborhood instead of past frames.

There exist other models which focus more on spatial information in the current frame than the temporal information in past frames to improve the model accuracy. But it is noticeable that the initial information used in these methods is obtained from results of former category. El Gammal et al. proposed foreground modeling for human body [8], and Sheikh and Shah proposed a general foreground model using past frames [9]. The first model is object-specific, and the second one needs slow object motion as its assumption, otherwise background samples will contaminate foreground model. A better principle method is using information in spatial correlation [10]. Using Markov random field modeling of changing labels can be seen as taking advantage of spatial information too [11]. Many have proved that combining temporal information with spatial information can offer us better results in subtraction.

3 Problem Statement

Our goal is to implement real-time background subtraction algorithms in different models using C++. We define the background in the common way, which refers to the stable objects in the scene and objects with repetitive small motion objects in the scene such as swaying tree branches and sea waves. We try to find the most suitable model models between time efficiency and detection results for real-time background subtraction. We choose C++ as our development language because it is easy to transplant and has high computational efficiency. We also want to make our program read in video stream directly from a web camera since networked video cameras are extensively used today.

We started from the basic methods such as using only Single Gaussian background model to do background subtraction. Then we went further to kernel density estimation background model. To increase the process speed of our algorithms, we add look-up table (LUT) and early-break method into our algorithms. Then we studied how the kernel numbers in buffer memory would affect the speed of the algorithms. We added Markov Random Field model of changing labels to improve the detection performance and studied the effect of numbers of iteration on the speed of the program. In the last, we use neighborhood information in the current frame to establish the foreground model, and studied its performance.

4 Implementation

(1) Single Gaussian Model

This model uses a single Gaussian distribution for the conditional background probability distribution and a uniform distribution for the foreground distribution probability. For every pixel in current frame, we can determine whether it is background pixel by comparing its probability of the current value with a fixed threshold. We only need to save two parameters for our background distribution since single Gaussian distribution only relates to its mean and variance. The Gaussian distribution is as follows:

$$P(I[\vec{x}]) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-\frac{(I[\vec{x}]-\mu)^2}{2\sigma^2}) \tag{2}$$

Where μ and σ are the mean and variance of the Gaussian function.

In order to adapt to the gradual illumination changes in background, we update the background probability for each pixel. In this project, we update single Gaussian model by running average.

$$\mu_{k+1} = \alpha B_k + (1-\alpha)\mu_k \tag{3}$$

$$\sigma^2_{k+1} = \alpha(B_k - \mu_k)^2 + (1-\alpha)\sigma^2_k \tag{4}$$

μ_k and σ_k are the mean and variance of the current pixel in the kth image, B_k is the background image used for the kth frame subtraction. α is the update parameter. The range of α is between 0 and 1. We update every pixel in the background image for each new frame.

By using this running average, we ensure that background can update with gradual illumination changes. Also, if there exist some false positives or miss in the initialized background, they can be corrected after some time with properly chosen update parameter α.

The flow chart of the Single Gaussian Model algorithm is:

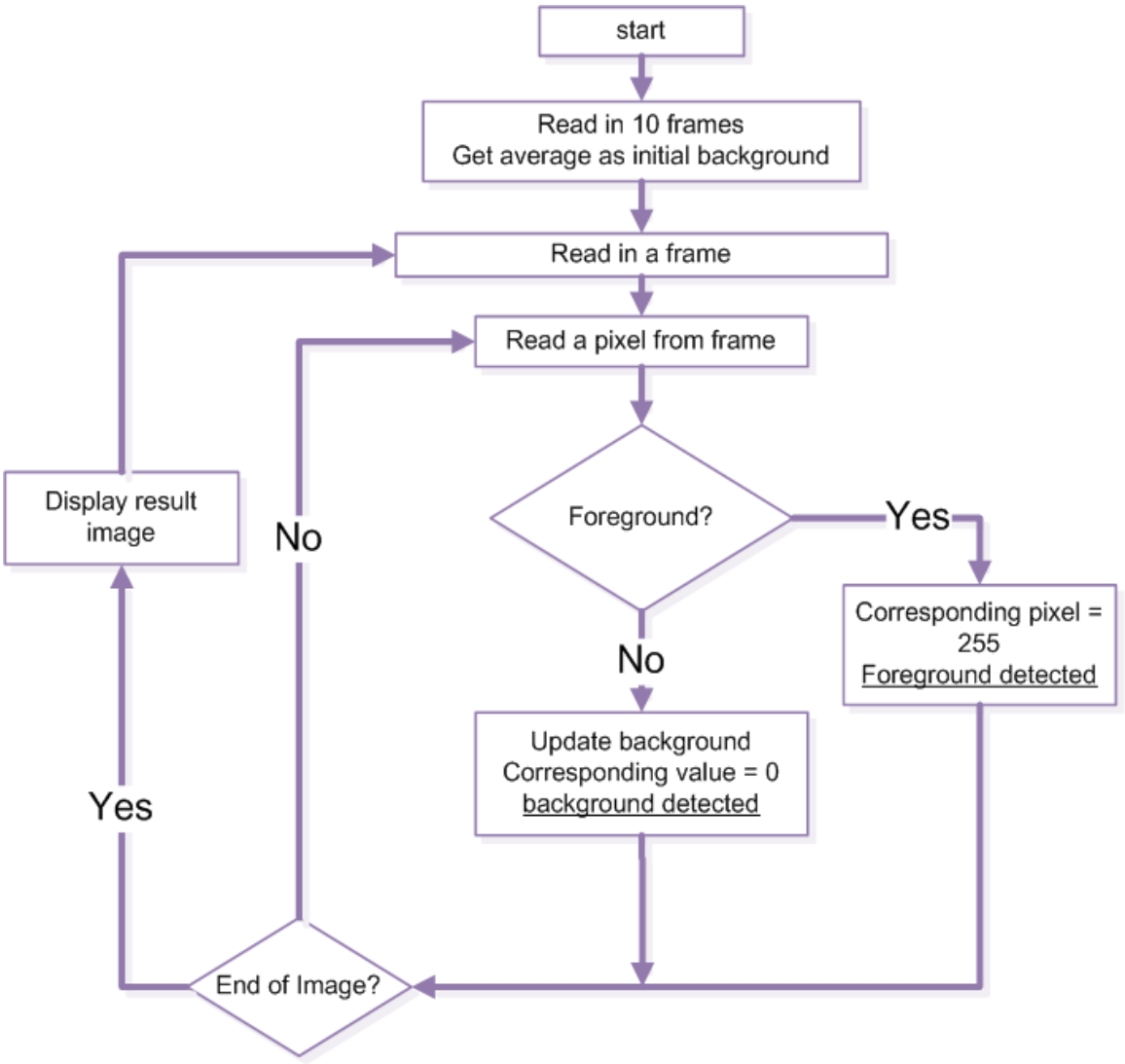

Fig. 1: flow chart of Single Gaussian Method

(2) Kernel Density Estimation

The main idea of this method is that the background PDF is given by the histogram of the n most recent pixel values, each smoothed with a kernel(specifically, a Gaussian kernel) (Fig. 2).

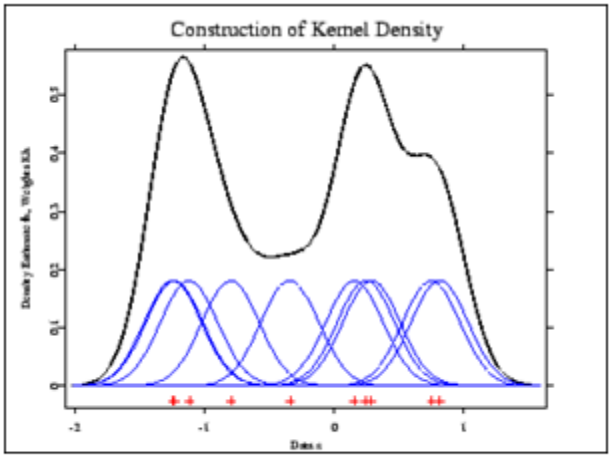

Fig. 2: Kernel Density Estimation Probability

We also assume foreground to be uniform. For every pixel in the new image, use a binary hypothesis test to check whether it is background pixel or foreground pixel. The flow chart of our program is in Fig. 3, and we will explain each step in the following text.

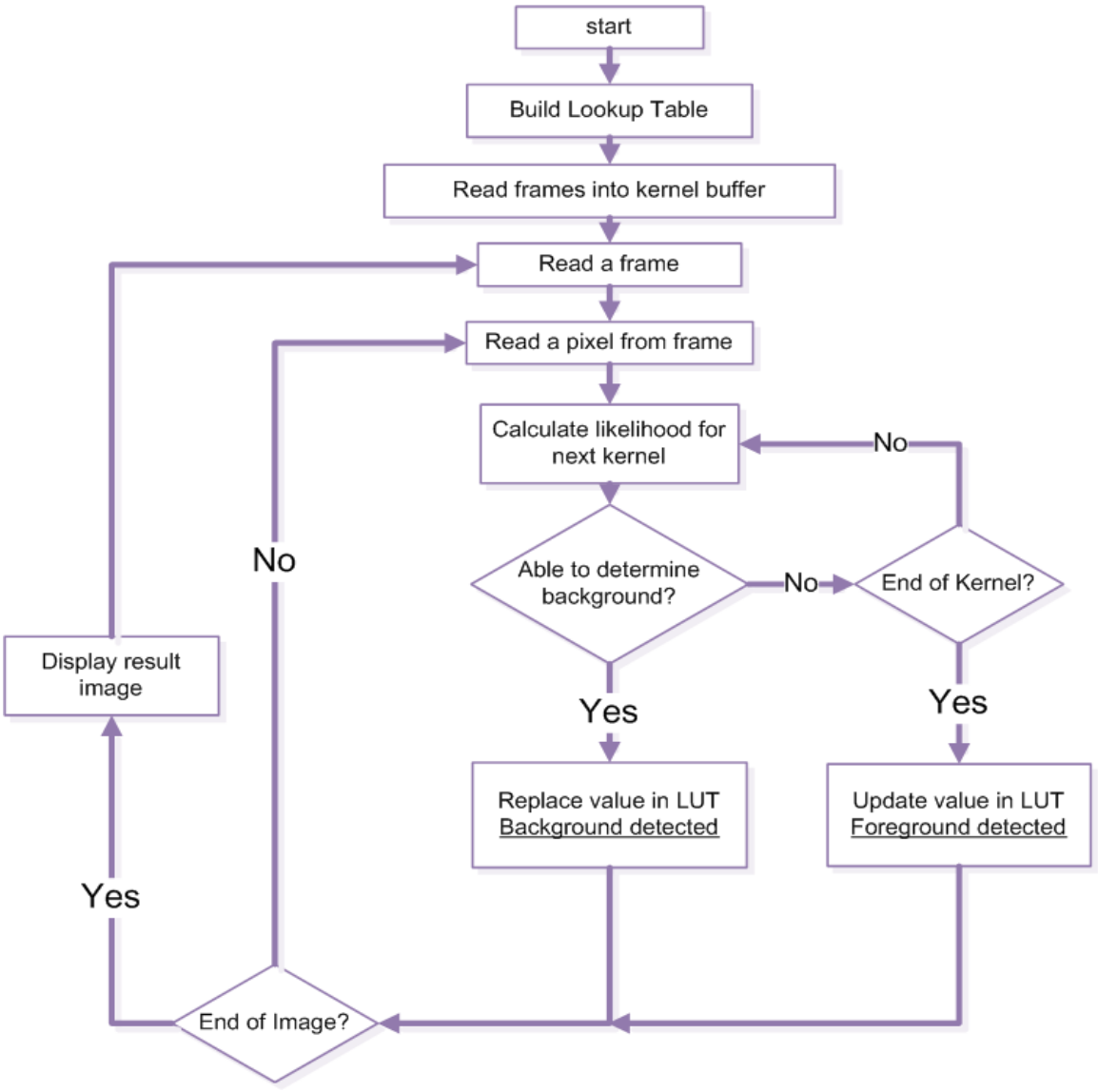

Fig. 3: flow chart KDE algorithm

In the initialization step where we build an initial background for the following detection, we pick only one frame and put it in the buffer for every ten frames we read in. We keep doing this until we fill the buffer. The buffer is a circular array which stores n frames of images, where n equals to the number of kernels. We call the frames stored in the buffer kernel images, because it provide the average parameter in each Gaussian kernel function when we do further computation. We initialize the buffer this way because if we choose continuous frames in the sequences to initialize the buffer, due to high fps, some slow-motion objects are probably stays at the same place in these frames.

These objects will be recognized as background and contaminate the background probability. Initialization this way will cost more time to initialize, but the time is negligible and it increases the accuracy of the model greatly.

More frames we store in the buffer, more accurate the probability distribution we will get. However, increase the buffer length will not only take more space but also more time to perform background subtraction. To improve the algorithm's efficiency, we introduce look-up table (LUT) and early-break method into the algorithms. .

$$P(I[\vec{x}]) = \sum_{i=0}^{n} w_{i,t}\eta_i$$
$$\eta_i = \frac{1}{\sqrt{2\pi\sigma_i^2}} \exp\left(-\frac{(I[\vec{x}]-\mu_i)^2}{2\sigma_i^2}\right) \tag{5}$$

In our Gaussian kernel function, every μ_i comes from a pixel value from the former frame, assume we have only 256 gray scale levels, then μ_i can only have 256 possible values. $I[\vec{x}]$ also have 256 possible values. Notice Gaussian function is symmetric, so there is only 256 possible values for the output result of Gaussian function. We can compute and store these values in memory before we do any processing of the frames. No matter how large the image size is and how long the sequence is, each time we read in a new pixel value $I[\vec{x}]$ and need to find the corresponding kernel probability value η_i, we can look it up directly in the memory according to $\left|I[\vec{x}]-\mu_i\right|$.

When we consider the foreground as uniform, if $P(I[\vec{x}])$ is larger than the threshold θ, then we can say the current pixel belongs to background. Notice $P(I[\vec{x}])$ is the sum of kernel probability value η_i

$$P(I[\vec{x}]) = \sum_{i=0}^{n} w_i\eta_i \tag{6}$$

and all η_i are positive numbers, so if the sum of part of η_i is larger than the threshold, we can classify this pixel to background. If we get a large enough sum of η_i in the first few η_i, then we do not need to go and find the other values of η_i. We call this algorithm as fast break because it will break out of the loop when we can ensure this pixel belongs to background. Because generally most of the pixels in one image belong to background, this early-break will make the algorithm more efficient.

Also, we use selective update method to update the kernel images in the buffer. There are two points need to be mentioned here:

(i) The update order:

Assume the buffer size is n, the buffer is circular array which will store the nearest past n frames. For frame k, suppose

$$l = k(\text{mod})n \tag{7}$$

Then we update the l^{th} frame in the buffer. Most of the time, the l^{th} frame is the oldest frame in the buffer, which is frame k – n. Our background model can track the gradual and even some sudden illumination changes (if fps is relatively high) using this update order.

(ii) Selective update between background and foreground:

When we update the kernel images, we have two separate ways to update the kernel images in the buffer depending on if it is detected as background or foreground. Let $F_k[\vec{x}]$ denotes the current value of the pixel locates at $\vec{x}$ in Frame k, and $\mu_l[\vec{x}]$ denotes the pixel value in buffer *l* at the same position. If $F_k[\vec{x}]$ is consider background pixel, then we have

$$\mu_l[\vec{x}] = F_k[\vec{x}] \tag{8}$$

If the pixel is detected as foreground, we have:

$$\mu_l[\vec{x}] = (1-\alpha)\mu_l[\vec{x}] + \alpha F_k[\vec{x}] \tag{9}$$

α is the update parameter which is between (0,1) and usually lies around 0.05 depending on different image sequences. For background, we use the current value

replace the one in the kernel image, but for foreground, we only update it a little. By doing this, for background, we keep the nearest n values of background in the buffer, and the background can track the illuminant changes; for foreground, we only slightly change the values in the buffer. However, if one pixel keeps to be detected as foreground pixel for a long time, it will gradually become background. This is reasonable because we often see there is geometry change in the background, such as a car drive into the scene and park there. By doing small update on pixels detected as foreground instead of not updating them, we can also fix the misses and false positives in the initial kernels.

(3) Markov Random Field(MRF)

If only KDE is implemented in the algorithm, as we have tried, it can be easily done in real time. But at the same time, the result is not satisfactory because of the false positives in background and some misses in the foreground objects. To remove them, we introduced Markov Random Field model of changing labels.

If U is a Markov Random Field, it can be characterized by Gibbs jointly probability distribution.

$$P(U_1 = u_1, ..., U_N = u_N) = \frac{1}{Z} e^{-E(u_1,....,u_N)/\beta} \tag{10}$$

$$E(u_1, ..., u_N) = \sum_{c \in C} V_c(u_1, ..., u_N) \tag{11}$$

Where:

Z - normalizing constant called partition function;

β - natural temperature(constant);

c - clique (geometric concept);

C - set of all cliques;

V_c - potential function (algebraic concept)

In our specific problem, the potential function is:

$$V_{\{\boldsymbol{x},\boldsymbol{y}\}}(e_k) = \begin{cases} 0 & \text{if } e_k[\boldsymbol{x}] = e_k[\boldsymbol{y}] \\ 1 & \text{if } e_k[\boldsymbol{x}] \neq e_k[\boldsymbol{y}] \end{cases} \tag{12}$$

Combine the function with binary hypothesis formula:

$$\frac{P(\Psi[\boldsymbol{x}] = \psi_k[\boldsymbol{x}]|\mathcal{M})}{P(\Psi[\boldsymbol{x}] = \psi_k[\boldsymbol{x}]|\mathcal{S})} \underset{\mathcal{S}}{\overset{\mathcal{M}}{\gtrless}} \theta \frac{P(E = e_k^{\mathcal{S}})}{P(E = e_k^{\mathcal{M}})} \tag{13}$$

After simplified, we can get:

$$\psi_k^2[\boldsymbol{x}] \underset{\mathcal{S}}{\overset{\mathcal{M}}{\gtrless}} 2\sigma_{\mathcal{S}}^2 \left(\ln(\theta \frac{\sigma_{\mathcal{M}}}{\sigma_{\mathcal{S}}}) + \frac{N_{\mathcal{S}} - N_{\mathcal{M}}}{T} \right) \tag{14}$$

This hypothesis test use spatial information to identify the status of each pixel and run it for several iterations before get satisfying result.

In MRF, for specific pixel, the status of the pixels in its neighborhood will affect its status, which means, if most of the pixels in its neighborhood in background, it is probably background pixel and vice visa. So it will remove the false positives and fill misses in foreground (because the pixels around false positives are almost background pixels and the pixels around the misses are almost the foreground pixels). On the other hand, it needs to run several iterations to get the result, so it is of comparatively lower computing efficiency. However, in the results below we can see that the main bottleneck of the efficiency is often not the number of iterations but the number of kernels in the buffer.

After we add MRF to the KDE method, the flow chart is shown as follows:

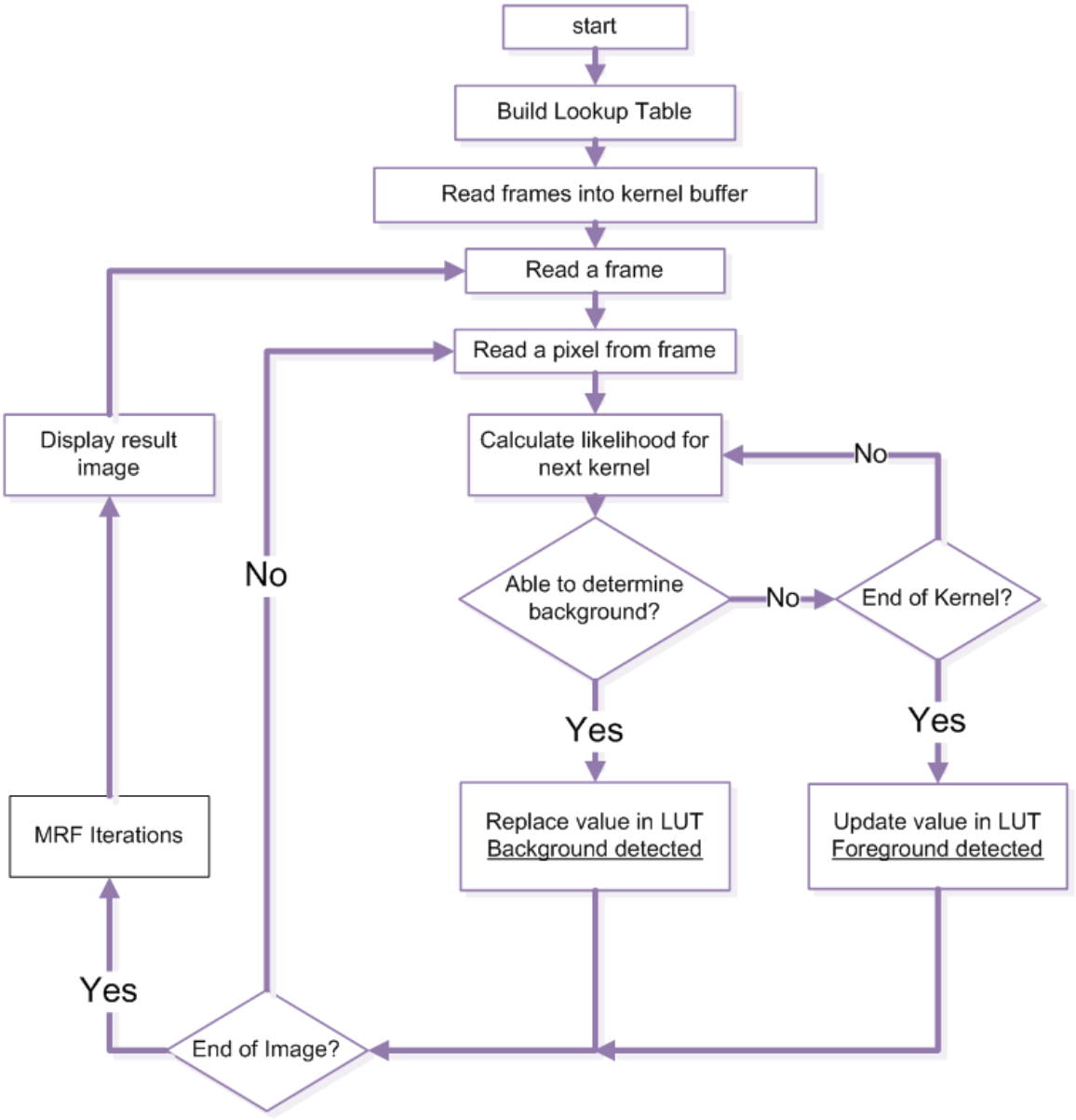

Fig. 4: KDE with MRF

Compared with the KDE method, we add a MRF iteration process after we get the motion label image in KDE.

(4) Neighborhood Foreground Models

This model is similar to kernel density estimation except to use spatial information to estimate the probability distribution, and it is explained clearly in [6]. To make this article concise, we just introduce this model in general but we implement the algorithm

and give out result below. We use neighborhood foreground model to further decrease the detection error rate. For every pixel in the frame, we use the neighborhood pixels of it to build its foreground model. Suppose $F_k[\vec{y}]$ is in the neighborhood of the current pixel $F_k[\vec{x}]$ and there is n number of neighbor of $\vec{x}$ (usually 8 or 24 or 48). Then we have its foreground model is:

$$P(I[\vec{x}]) = \sum_{i=0}^{n} w_i \eta_i \tag{15}$$

$$\eta_i = \frac{1}{\sqrt{2\pi\sigma_i^2}} \exp(-\frac{(F_k[\vec{x}] - F_k[\vec{y}])^2}{2\sigma_i^2}) \tag{16}$$

5. Experimental Results

The system parameters of the system where we test our algorithms are listed below:
CPU: Inter(R) Core(TM)2 Duo CPU T8300@2.40GHz
RAM: 3.00GB
System: Windows 7
Development Enviroment: Microsoft Visual C++ 2005, version 8.05 , OpenCV2.0

We record some long video sequences from the web camera and find the average frame rate in daytime of that camera is 20fps. This is a common value for many webcams. The processing time should be under 50ms to ensure real-time processing.

(1) Single Gaussian Method results:
The result is shown as follows:

Fig. 5: the result of Single Gaussian Model

We can find that in the right picture there is a large part of false alarms in the bottom and some misses. The reason is that when initializing the background model, there is a bus at that part, so the pixels in the bus are taken as the background, which results in errors.

After some time, it can be fixed. The time needed to fix the error depends on the update parameter α. The result after some time is shown as follows:

Fig. 6: the result after some time

The processing time of one single run plot is shown as follows (horizontal axis is the index of frames, vertical axis is the processing time measure in milliseconds, for 320*240 image, 3 channels):

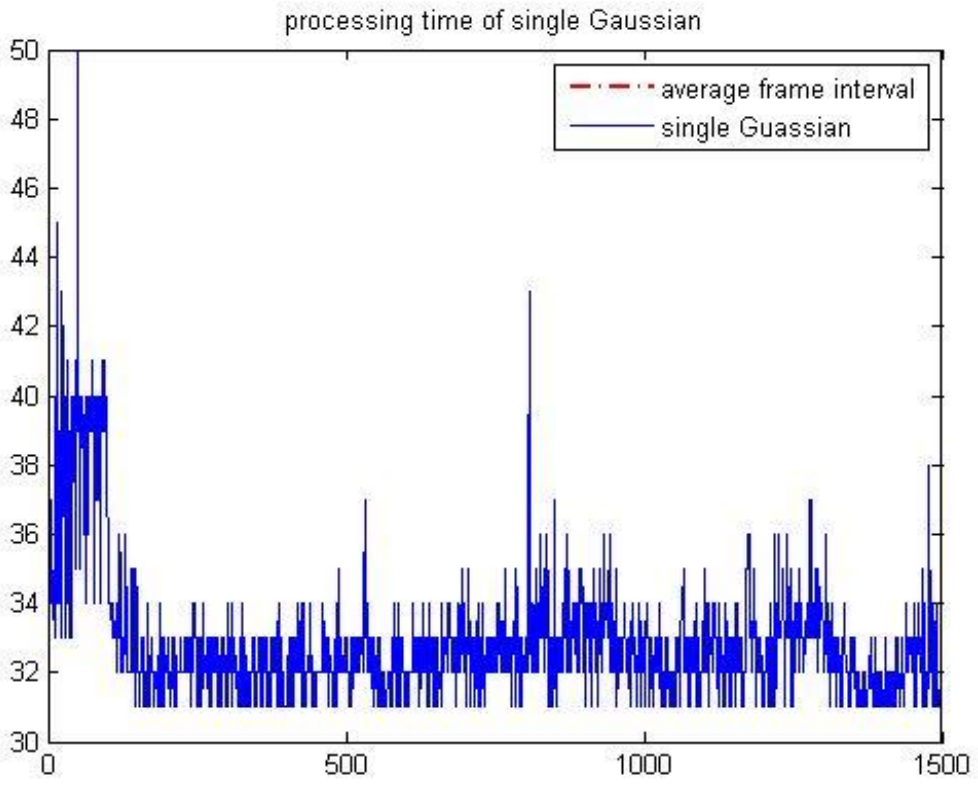

Fig. 7: processing time of Single Gaussian Method

Although single Gaussian is not accurate enough and may introduce lots of errors, it is very fast. For each channel, it just needs about 10-12ms to process. So it can be used to process very large image sequences. In experiment, it can process a 3-channel image(480* 360) in 75-78ms and it can process larger 1-channel image sequences in real time.

(2) KDE with LUT and early-break results

The result is shown as follows:

Fig. 8: the result of KDE with LUT and early-break

Compare with the result from single Gaussian Method, we can see that the false alarms are significantly decreases but there are still some misses, although the number of misses is smaller than that in Single Gaussian Method.

The processing time of one single run with and without LUT are shown as follows (for 320*240 image sequences):

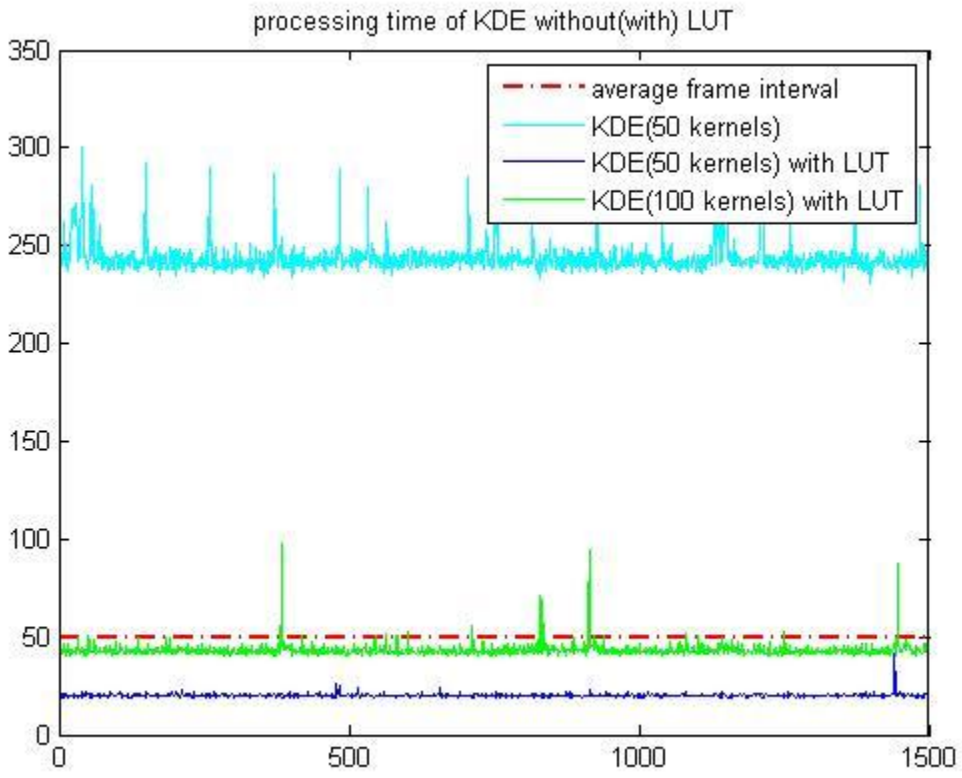

Fig. 9: processing time for KDE with LUT

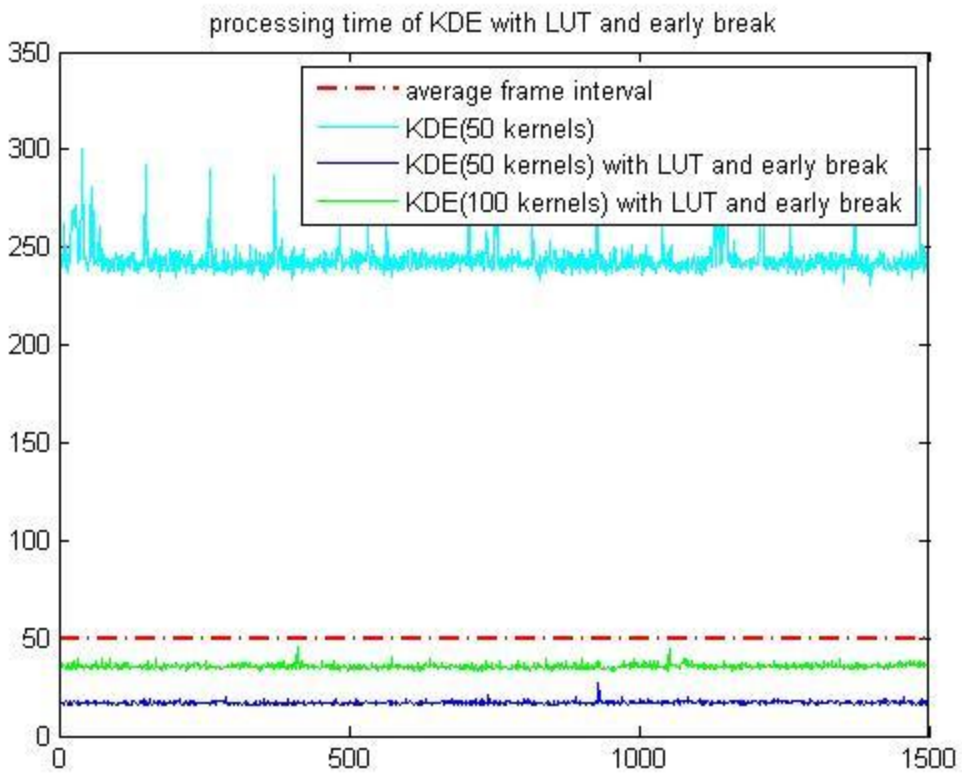

Fig. 10: processing time for KDE with LUT and early-break

It can be found that if LUT and early-break are not applied in KDE, it will cost a long time to process one frame(about 250ms), and after applying only LUT to it, it can afford 100 kernels in the buffer to process the image sequences in real time, so the LUT is of great importance in the algorithm. And after applying early-break, the processing time decreases again about 25%.

(3) KDE with MRF results

The result of KDE with 1st order MRF is shown as follows:

Fig. 11: KDE with 1st order MRF

Compared to the results of Single Gaussian Method and KDE, we can see from the result of the algorithm that most of false alarms and misses disappear and we obtain comparatively good results.

To find out the bottleneck of real-time background subtraction, we also try to run the algorithm with different number of kernels and iterations of MRF and we get the result as follows (for 320*240 image sequences):

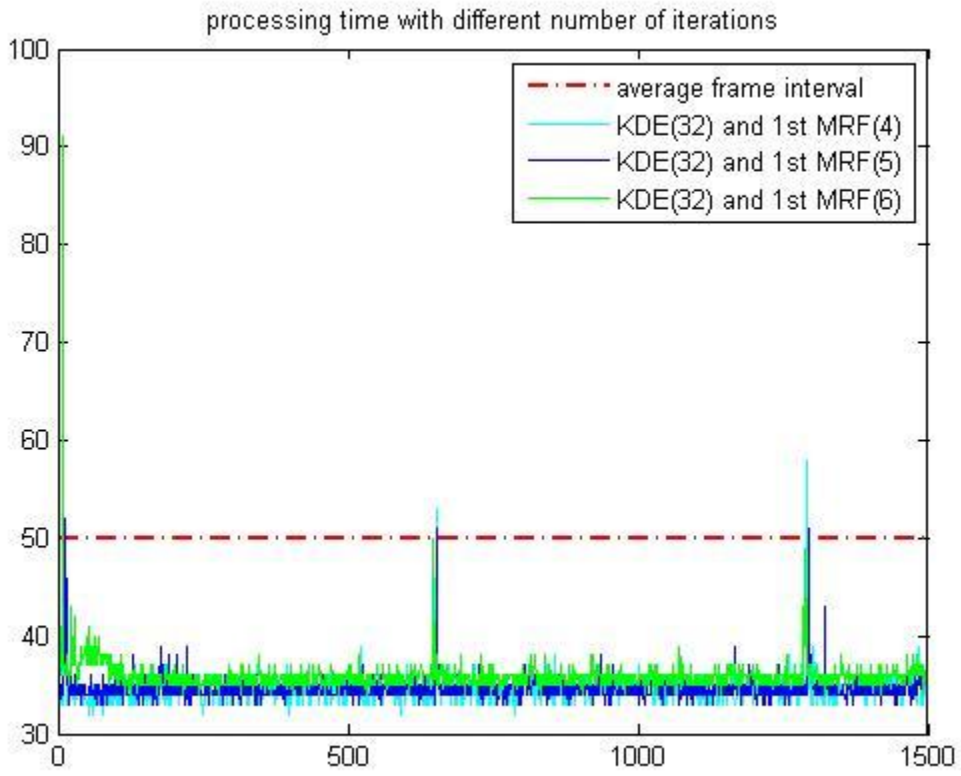

Fig. 12: processing time with different number of iterations in 1st MRF

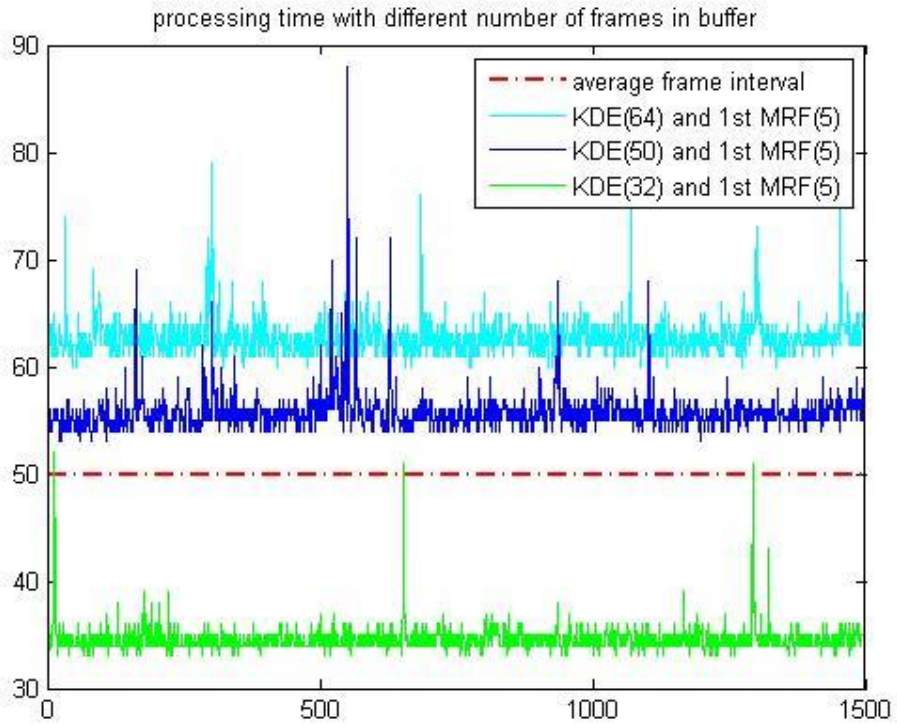

Fig. 13: processing time with different number of kernels in 1st MRF

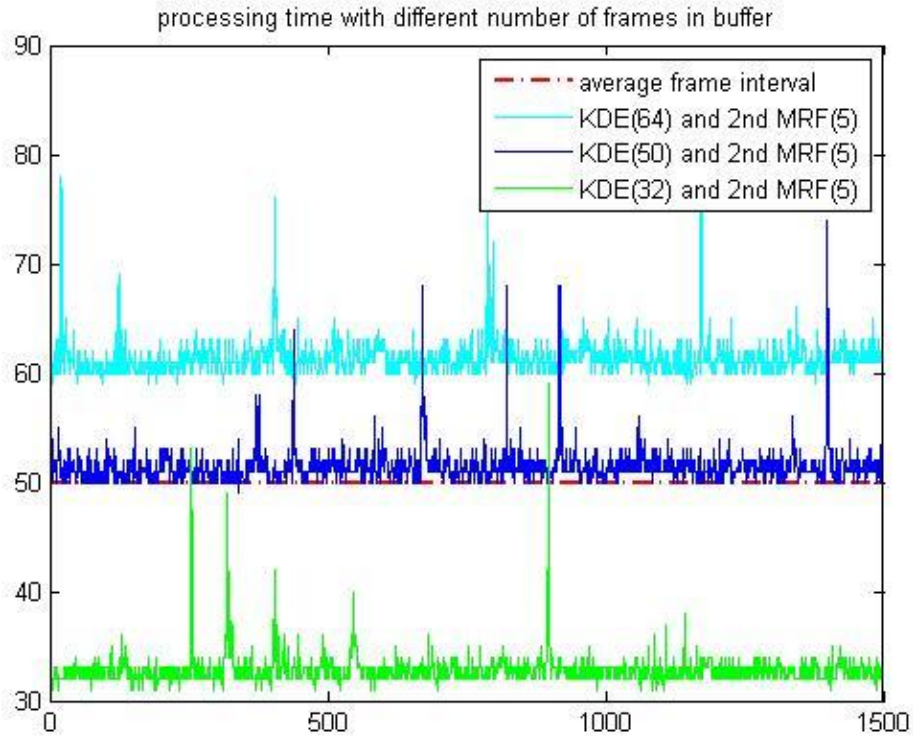

Fig. 14: processing time with different number of kernels in 2nd MRF

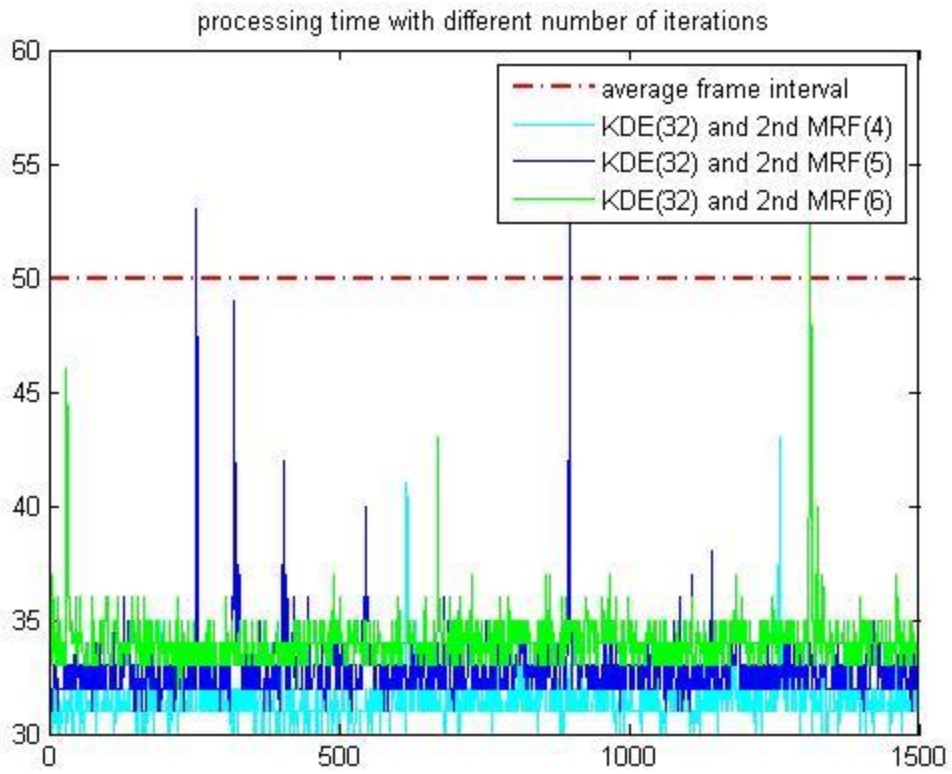

Fig. 15: processing time with different number of kernels in 2nd MRF

And from the four plots above, the algorithm needs roughly one more millisecond to processing one image as the number of kernel increases by 1 or the number of MRF increases by one. However, there is little improvement when the number of iterations is larger than 4. From all the results above, we can know that the main bottleneck factor in real-time background subtraction is the number of kernels.

(4) The result of applying foreground model

The result is shown as follows:

Fig. 16 Applying foreground model to the algorithm

Actually, after applying foreground model to the algorithm, the quality does not change much. And the processing time has increased approximately 7%:

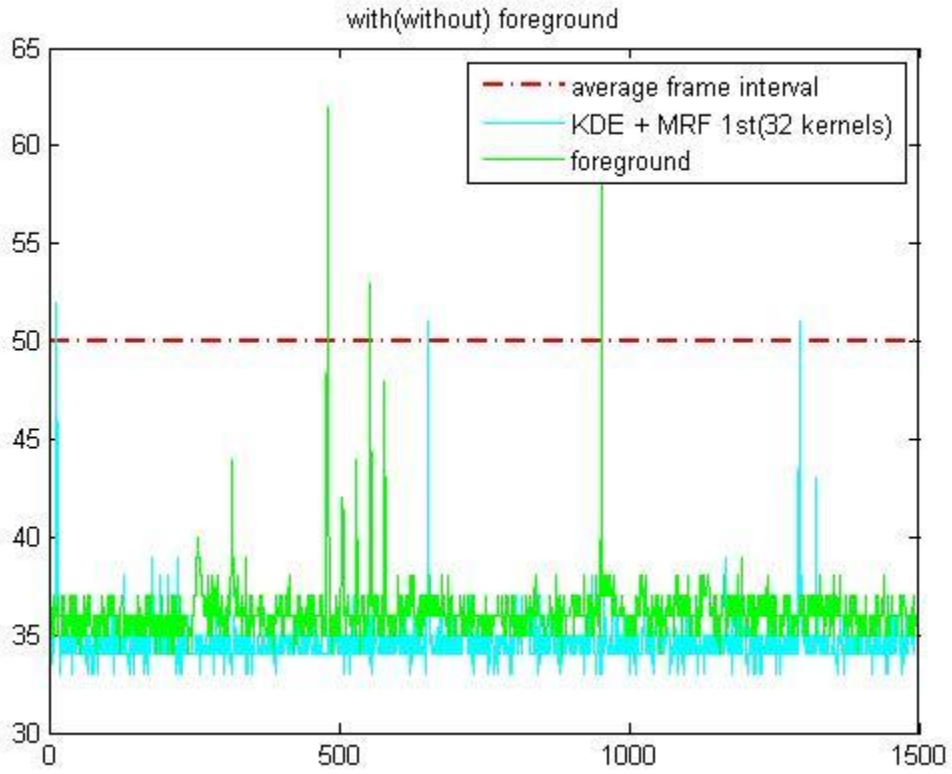

Fig. 17 the processing time after applying foreground to the algorithm

From the Fig. 17 and Fig. 18, it can be found that after applying foreground model to the algorithm, the quality has little improvement.

Finally, we give some tables for the average processing time of every method:

	320* 240 pixels	480*360 pixels
Single Gaussian Method(1 channel)	11.1ms	26.1ms
Single Gaussian Method(3 channels)	34.3ms	76.4ms

Table 1: average processing time for Single Gaussian Method (1 channel and 3 channels)

	320* 240 pixels	480*360 pixels
KDE(50 kernels)	250.4ms	600.6ms
KDE with LUT(50 kernels)	24.9ms	76.3ms
KDE with LUT(100 kernels)	49.3ms	300.2ms
KDE with LUT and early-break(50 kernels)	20.1ms	37.4ms
KDE with LUT and early-break(100 kernels)	42.2ms	94.3ms

Table 2: average processing time for KDE

	320*240 pixels	480*360 pixels
KDE(50 kernels) with MRF(4 iterations)	49.4ms	124.3ms
KDE(50 kernels) with MRF(5 iterations)	52.1ms	128.2ms
KDE(50 kernels) with MRF(6 iterations)	53.7ms	131.6ms

Table 3: average processing time for KDE with MRF (1^{st}) of different number of iterations

	320*240 pixels	480*360 pixels
KDE(32 kernels) with MRF(1^{st})	35.5ms	78.3ms
KDE(50 kernels) with MRF(1^{st})	55.5ms	124.5ms
KDE(64 kernels) with MRF(1^{st})	62.6ms	144.7ms

Table 4: average processing time for KDE with MRF (1^{st}) of different number of kernels

	320*240 pixels	480*360 pixels
KDE(50 kernels) with MRF(4 iterations)	49.3ms	123.1ms
KDE(50 kernels) with MRF(5 iterations)	50.3ms	124.4ms
KDE(50 kernels) with MRF(6 iterations)	53.2ms	129.5ms

Table 5: average processing time for KDE with MRF (2^{nd}) of different number of iterations

	320*240 pixels	480*360 pixels
KDE(32 kernels) with MRF(2^{st})	35.5ms	78.3ms
KDE(50 kernels) with MRF(2^{st})	50.6ms	116.6ms
KDE(64 kernels) with MRF(2^{st})	62.8ms	144.4ms

Table 6: average processing time for KDE with MRF (2^{nd}) of different number of kernels

We summarize the result in the next table, if its average processing time is larger than 50ms, we consider this method to be real-time (Denote as “Y” in the table), else we consider it cannot be done in real-time (Denote as “N” in the table):

		320*240 pixels	480*360 pixels
Gaussian	1 channel	Y	Y
	3 channels	Y	N
KDE(16 kernels)	LUT	N	N
	LUT and early-break	Y	Y
	MRF (4 iterations)	Y	Y
	MRF (5 iterations)	Y	Y
	MRF (6 iterations)	Y	Y
KDE(32 kernels)	LUT	N	N
	LUT and early-break	Y	N
	MRF (4 iterations)	Y	N
	MRF (5 iterations)	Y	N
	MRF (6 iterations)	Y	N
KDE(50 kernels)	LUT	N	N
	LUT and early-break	Y	N
	MRF (4 iterations)	Y	N
	MRF (5 iterations)	Y	N
	MRF (6 iterations)	Y	N
KDE(100 kernels)	LUT	N	N
	LUT and early-break	Y	N
	MRF (4 iterations)	N	N
	MRF (5 iterations)	N	N
	MRF (6 iterations)	N	N

Foreground Model	MRF(1^{st})	Y	N
	MRF(2^{nd})	Y	N

Table 7 Final results

6. Conclusion

From the project, we learned that:

(1) Single Gaussian Method is fast and easy to implement in real-time, but it usually has bad performance in complicated situations.

(2) Kernel density estimation model building is not efficient enough, often it cannot be done in real time, but if we introduce LUT and early-break method into the KDE model, it can be done in real time and we get robust and satisfying result which will adapt to illuminant and motion changes in the background.

(3) The bottleneck factors for real-time background subtraction are mainly the data amount (image resolutions) and kernel numbers, the orders or iteration numbers of MRF have insignificant effect on that.

(4) Kernel numbers is an essential factor on both the processing speed and the detection error rate. Orders or iteration numbers have much less effect on detection error rate. Although the processing time for applying neighborhood foreground models in the algorithm do not increase significantly, but it also has insignificant improvement in decreasing the error rate.

Possible future work may include: using multi-threading algorithms or multi-core workstations to process large image sequences. However, as we define our topic as read in streams from web cameras directly and perform background subtraction, larger resolution images processing seems unnecessary due to the networsk capacity constraints.

7. References

[1] B.P.L. Lo and S.A. Velastin, "Automatic congestion detection system for underground platforms," *Proc. of 2001 Int. Symp. on Intell. Multimedia*, Video and Speech Processing, pp. 158-161, 2000.

[2] R. Cucchiara, C. Grana, M. Piccardi, and A. Prati, "Detecting moving objects, ghosts and shadows in video streams", *IEEE Trans. on Patt. Anal. and Machine Intell.*, vol. 25, no. 10, Oct. 2003, pp. 1337-1342.

[3] C. Wren, A. Azarbayejani, T. Darrell, and A. Pentland, "Pfinder:Real-time Tracking of the Human Body," *IEEE Trans. on Patt. Anal. and Machine Intell.*, vol. 19, no. 7, pp. 780-785, 1997.

.

[4] C. Stauffer, W.E.L. Grimson, "Adaptive background mixture modelsfor real-time tracking", *Proc. of CVPR 1999*, pp. 246-252.

[5] A. Elgammal, D. Harwood, and L.S. Davis, "Non-parametric Model for Background Subtraction", *Proc. of ICCV '99 FRAME-RATE Workshop*, 1999.

[6] J. Mike McHugh, Janusz Konrad, Venkatesh Saligrama, and Pierre-Marc Jodoin, "Foreground-Adaptive Background Subtraction", *IEEE Signal Process. Lett*, 2009 IEEE.

[7] P.-M. Jodoin, M. Mignotte, and J. Konrad, "Statistical background subtraction using spatial cues," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 17, pp. 1758–1763, Dec. 2007.

[8] A. Elgammal, R. Duraiswami, D. Harwood, and L. Davis, "Background and foreground modeling using nonparametric kernel density for visual surveillance", *Proc. IEEE*, vol. 90, pp. 1151.1163, 2002.

[9] Y. Sheikh and M. Shah, "Bayesian modeling of dynamic scenes for object detection", *IEEE Trans. Pattern Anal. Machine Intell.*, vol. 27, no. 11, pp. 1778.1792, 2005.

[10] M. Seki, T. Wada, H. Fujiwara, K. Sumi, "Background detection based on the cooccurrence of image variations", *Proc. of CVPR 2003*, vol. 2, pp. 65-72.

[11] T. Aach and A. Kaup, "Bayesian algorithms for adaptive change detection in image sequences using Markov random fields", *Signal Process., Image Commun*., vol. 7, pp. 147.160, 1995.

8. Appendix

Below is listed C++ source code developed for this project. It can be downloaded from the address below:

(1) For single Gaussian Model

```
#include "stdafx.h"
#include <iostream>
#include <string>
#include <highgui.h>
#include <cv.h>
#include <math.h>
#include <stdlib.h>
#include <windows.h>
#include <ctime>

LARGE_INTEGER limtp;

const double e = 2.7183;
const double segama = 30;
const double pi = 3.14;
double temp;
const int numFrame = 10;
const int numInitial = 1;
const int numMRF = 4;
const int num_write = 500;
const double threshold = 0.65;

using namespace std;

double gaussian_value(int value)
{
	double index = -pow((double)value, 2)/(2 * pow(segama, 2));
	temp = pow(e, index)/numFrame;
	return temp;
}

int main(int argc, char **argv)
{
	//set up windows
	cvNamedWindow("origin", CV_WINDOW_AUTOSIZE);
	cvNamedWindow("processing", CV_WINDOW_AUTOSIZE);
	cvNamedWindow("result", CV_WINDOW_AUTOSIZE);


	CvCapture* capture = NULL;
	//capture = cvCreateFileCapture( "http://vsn3-iss.bu.edu/mjpg/video.mjpg" );
```

```
    capture = cvCaptureFromAVI("vsn3_3.avi");

    IplImage* frame = NULL; //original images of video
    IplImage* frame_u = NULL; //expected images of video
    IplImage* frame_var = NULL; //variance images
    IplImage* frame_std = NULL; //deviation images
    IplImage* frame_bin = NULL; //binary images
    IplImage* frame_diff = NULL; //difference images

    double alpha = 0.05; //updata parameter alpha
   double std_init = 20; //initialized std
   double var_init = std_init * std_init; //initialized var
   double lamda = 2.5 * 1.2; //updata parameter


    CvScalar pixel = {0}; //original values of pixels
    CvScalar pixel_u = {0}; //expected values of pixels
    CvScalar pixel_var = {0}; //variance images
    CvScalar pixel_std = {0}; //standard deviation images
    CvScalar pixel_for = {255, 0, 0, 0};
    CvScalar pixel_back = {0};

    //initiallize frame_u, frame_var, frame_std
    frame = cvQueryFrame(capture);
    frame_u = cvCreateImage(cvSize(frame->width, frame->height), IPL_DEPTH_8U, 3);
    frame_var = cvCreateImage(cvSize(frame->width, frame->height), IPL_DEPTH_8U, 3);
    frame_std = cvCreateImage(cvSize(frame->width, frame->height), IPL_DEPTH_8U, 3);
    frame_diff = cvCreateImage(cvSize(frame->width, frame->height), IPL_DEPTH_8U, 3);
    frame_bin = cvCreateImage(cvSize(frame->width, frame->height), IPL_DEPTH_8U, 1);

    CvSize size = {frame->width, frame->height};

    //CvVideoWriter* writer1 = 0;
    //CvVideoWriter* writer2 = 0;
    //int isColor1 = 1;
    //int isColor2 = 0;
    //double fps = 20;
    //writer1 = cvCreateVideoWriter("video_origin_single_gaussian.avi", CV_FOURCC('P', 'I', 'M',
'1'), fps, size, isColor1);
    //writer2 = cvCreateVideoWriter("video_back_single_gaussian.avi", CV_FOURCC('P', 'I', 'M',
'1'), fps, size, isColor2);

    for(int y = 0; y < frame->height; ++y)
    {
         for(int x = 0; x < frame->width; ++x)
         {
          pixel = cvGet2D(frame, y, x);

          pixel_u.val[0] = pixel.val[0];
          pixel_u.val[1] = pixel.val[1];
          pixel_u.val[2] = pixel.val[2];

          pixel_std.val[0] = std_init;
          pixel_std.val[1] = std_init;
          pixel_std.val[2] = std_init;
```

```
            pixel_var.val[0] = var_init;
            pixel_var.val[1] = var_init;
            pixel_var.val[2] = var_init;

            cvSet2D(frame_u, y, x, pixel_u);
            cvSet2D(frame_var, y, x, pixel_var);
            cvSet2D(frame_std, y, x, pixel_std);
          }
      }

      int num = 0;
      time_t TimeStart, TimeEnd, TimeUsed;
      int time[1500];
      char* filename = "D:\\ec720\\data\\RGB_single_Gaussian_480.txt";
      FILE* fp = fopen(filename, "w");
      while(cvWaitKey(1) != 27)    //press ESC to quit, fps = 1/33
      {
           frame = cvQueryFrame(capture);

           TimeStart = clock();
           //Guassian model
        for (int y = 0; y < frame->height; ++y)
        {
            for (int x = 0; x < frame->width; ++x)
            {
                pixel = cvGet2D(frame, y, x);
                pixel_u = cvGet2D(frame_u, y, x);
                pixel_std = cvGet2D(frame_std, y, x);
                pixel_var = cvGet2D(frame_var, y, x);

                //if|I-u| < lamda*std, background, update
                if (fabs(pixel.val[0] - pixel_u.val[0]) < lamda * pixel_std.val[0] &&
                    fabs(pixel.val[1] - pixel_u.val[1]) < lamda * pixel_std.val[1] &&
                    fabs(pixel.val[2] - pixel_u.val[2]) < lamda * pixel_std.val[2])
                {
                    //update u = (1-alpha)*u + alpha*I
                    pixel_u.val[0] = (1 - alpha) * pixel_u.val[0] + alpha * pixel.val[0];
                    pixel_u.val[1] = (1 - alpha) * pixel_u.val[1] + alpha * pixel.val[1];
                    pixel_u.val[2] = (1 - alpha) * pixel_u.val[2] + alpha * pixel.val[2];

                    //update var = (1-alpha)*var + alpha*(I-u)^2
                    pixel_var.val[0] = (1 - alpha) * pixel_var.val[0] +
                                       (pixel.val[0] - pixel_u.val[0]) * (pixel.val[0] -
pixel_u.val[0]);
                    pixel_var.val[1] = (1 - alpha) * pixel_var.val[1] +
                                       (pixel.val[1] - pixel_u.val[1]) * (pixel.val[1] -
pixel_u.val[1]);
                    pixel_var.val[2] = (1 - alpha) * pixel_var.val[2] +
                                       (pixel.val[2] - pixel_u.val[2]) * (pixel.val[2] -
pixel_u.val[2]);

                    //update deviation
                    pixel_std.val[0] = sqrt(pixel_var.val[0]);
                    pixel_std.val[1] = sqrt(pixel_var.val[1]);
```

```
                pixel_std.val[2] = sqrt(pixel_var.val[2]);

                //write into matrix
                cvSet2D(frame_u, y, x, pixel_u);
                cvSet2D(frame_var, y, x, pixel_var);
                cvSet2D(frame_std, y, x, pixel_std);
            }
         }
    }

      cvAbsDiff(frame_u, frame, frame_diff);
      for(int y = 0; y < frame->height; ++y)
      {
           for(int x = 0; x < frame->width; ++x)
           {
                if((frame_diff->imageData + y * frame_diff->widthStep)[3 * x] > 20 &&
                     (frame_diff->imageData + y * frame_diff->widthStep)[3 * x + 1] > 20 &&
                     (frame_diff->imageData + y * frame_diff->widthStep)[3 * x + 2] > 20 )
                     /*(frame_bin->imageData + y * frame_bin->widthStep)[x] = 255;*/
                     cvSet2D(frame_bin, y, x, pixel_for);
                else
                     /*(frame_bin->imageData + y * frame_bin->widthStep)[x] = 0;*/
                     cvSet2D(frame_bin, y, x, pixel_back);
           }
      }

      TimeEnd = clock();
      TimeUsed = TimeEnd - TimeStart;
      time[num] = TimeUsed;
      cout << time[num] << endl;

    //show the result
    cvShowImage("origin", frame);
    cvShowImage("processing", frame_u);
      cvShowImage("result", frame_bin);


      //cvWriteFrame(writer1, frame);
      //cvWriteFrame(writer2, frame_bin);
      num++;
      if(num == 1500)
           break;


}

 for(int i = 0; i < 1500; ++i)
      fprintf(fp, "%d\n", time[i]);
 fclose(fp);
//Release the memory
cvReleaseCapture(&capture);
//cvReleaseImage(&frame);
cvReleaseImage(&frame_u);
 cvReleaseImage(&frame_var);
 cvReleaseImage(&frame_std);
```

```
        cvReleaseImage(&frame_bin);
        cvReleaseImage(&frame_diff);

    cvDestroyWindow("origin");
    cvDestroyWindow("processing");
        cvDestroyWindow("result");

    return 0;
}
```

For KDE with LUT, early-break and MRF

```
#include "stdafx.h"
#include <iostream>
#include <string>
#include <highgui.h>
#include <cv.h>
#include <math.h>
#include <stdlib.h>
#include <windows.h>
#include <ctime>

LARGE_INTEGER limtp;

const double e = 2.7183;
const double segama = 30;
const double pi = 3.14;
double temp;
const int numFrame = 64;
const int numInitial = 1;
const int numMRF = 5;
const int num_write = 500;
const double threshold = 0.58;
const double alpha = 0.05;

using namespace std;

//    return frame_result;
//}

double gaussian_value(int value)
{
      double index = -pow((double)value, 2)/(2 * pow(segama, 2));
      temp = pow(e, index)/numFrame;
      return temp;
}


int main(int argc, char **argv)
{
      //set up windows
      cvNamedWindow("origin", CV_WINDOW_AUTOSIZE);
      cvNamedWindow("result", CV_WINDOW_AUTOSIZE);
      double hash[256];

      for(int i = 0; i < 256; ++i)
            hash[i] = gaussian_value(i);

      CvCapture* capture = NULL;
      capture = cvCaptureFromAVI("vsn3_3.avi");
      //capture = cvCreateFileCapture( "http://vsn3-iss.bu.edu/mjpg/video.mjpg" );

      IplImage* frame = NULL; //original images of video
      IplImage* frame_gray = NULL; //original images of video of gray level
      IplImage* frame_temp = NULL; //next frame to be computed
```

```
IplImage* frame_result = NULL; // the result after background subtraction
IplImage* chain[numFrame]; //save the images to compute the kernel
IplImage* diff[numFrame]; //save the absolute difference
IplImage* phi[numFrame]; //save the MRF temp results
IplImage* dividend;
//IplImage* abc;
IplImage* frame_left_shift = NULL; //save the image with 1 pixel shifted to the left
IplImage* frame_right_shift = NULL; //save the image with 1 pixel shifted to the right
IplImage* frame_up_shift = NULL; //save the image with 1 pixel shifted to the up
IplImage* frame_down_shift = NULL; //save the image with 1 pixel shifted to the down
IplImage* frame_MRF = NULL; //save the MRF
IplImage* frame_diff = NULL; //save the difference of the image

frame = cvQueryFrame(capture);
CvSize size = {frame->width, frame->height};
CvSize size_shift = {frame->width, frame->height};
dividend = cvCreateImage(size, IPL_DEPTH_8U, 1);
frame_diff = cvCreateImage(size, IPL_DEPTH_8U, 1);

frame_MRF = cvCreateImage(size_shift, IPL_DEPTH_8U, 1);


CvScalar value = {numFrame, 0, 0, 0};

cvSet(dividend, value);

for(int i = 0; i < numFrame; ++i)
     phi[i] = cvCreateImage(size, IPL_DEPTH_8U, 1);

for(int num = 0; num < numFrame; ++num)
{
     int count = 0;
     chain[num] = cvCreateImage(size, IPL_DEPTH_8U, 1);
     //cvSetZero(frame_temp);
     while(1)
     {
          frame = cvQueryFrame(capture);
          if(count == 0)
          {
               frame_gray = cvCreateImage(size, IPL_DEPTH_8U, 1);
               cvCvtColor(frame, frame_gray, CV_RGB2GRAY);
               chain[num] = frame_gray;
          }
          count++;
          if(count == 7)
               break;


          cvWaitKey(37);
          //count++;
          //if(count == numInitial)
          //    break;
     }
}// initialize
```

```
//frame_gray = cvCreateImage(size, IPL_DEPTH_8U, 1);
for(int i = 0; i < numFrame; ++i)
	diff[i] = cvCreateImage(size, IPL_DEPTH_8U, 1);


int count_write = 0;
time_t TimeStart, TimeEnd, TimeUsed;
int time[1500];
int num_time = 0;

frame_gray = cvCreateImage(size, IPL_DEPTH_8U, 1);
while(1)
{
	frame = cvQueryFrame(capture);
	TimeStart = clock();
	cvCvtColor(frame, frame_gray, CV_RGB2GRAY);

	for(int i = 0; i < numFrame; ++i)
		cvAbsDiff(frame_gray, chain[i], diff[i]);

	int num = 0;
	frame_result = cvCreateImage(size, IPL_DEPTH_8U, 1);
	for(int i = 0; i < frame_gray->width; ++i)
	{
		for(int j = 0; j < frame_gray->height; ++j)
		{
			double temp = 0;
			for(int k = 0; k < numFrame; ++k)
			{
				(frame_result->imageData + frame_result->widthStep * j)[i] = 1;
				int value = (diff[k]->imageData + diff[k]->widthStep * j)[i];
				temp += hash[value];
				/*temp += gaussian_value(value);*/
				if(temp > threshold)
				{
					(frame_result->imageData + frame_result->widthStep * j)[i] = 0;
					break;
				}
			}
		}
	}



	cvSetZero(frame_diff);
	for(int i = 0; i < numFrame; ++i)
	{
		cvDiv(diff[i], dividend, phi[i], 1);
		cvAdd(phi[i], frame_diff, frame_diff);
	}



	for(int k = 0; k < numMRF; ++k)
```

```
		{
			for(int i = 1; i < frame->width - 1; ++i)
				for(int j = 1; j < frame->height - 1; ++j)
					(frame_MRF->imageData + frame_MRF->widthStep * j)[i] =
(frame_result->imageData + frame_result->widthStep * (j - 1))[i] + (frame_result->imageData +
frame_result->widthStep * (j + 1))[i] + (frame_result->imageData + frame_result->widthStep * j)[i
+ 1] + (frame_result->imageData + frame_result->widthStep * j)[i - 1];


			//cvAdd(frame_left_shift, frame_right_shift, frame_MRF);
			//cvAdd(frame_MRF, frame_up_shift, frame_MRF);
			//cvAdd(frame_MRF, frame_down_shift, frame_MRF);

			//cvSetZero(frame_diff);
			//for(int i = 0; i < numFrame; ++i)
			//{
			//	cvDiv(diff[i], dividend, phi[i], 1);
			//	cvAdd(phi[i], frame_diff, frame_diff);
			//}

			cvSetZero(frame_result);
			for(int i = 0; i < frame->width; ++i)
				for(int j = 0; j < frame->height; ++j)
					if(pow((double)(frame_diff->imageData + frame_diff->widthStep * j)[i],
2) > 2 * pow(segama, 2) * (2.5 - 2 * (frame_MRF->imageData + frame_MRF->widthStep * (j + 1))[i + 1]))
						if(k != numMRF - 1)
							(frame_result->imageData + frame_result->widthStep * j)[i] =
1;
						else
							(frame_result->imageData + frame_result->widthStep * j)[i] =
255;
		}
		//cvDilate(frame_result, frame_result);
		//cvErode(frame_result, frame_result);
		cvShowImage("origin", frame);
		/*cvShowImage("test", frame_left_shift);*/
		cvShowImage("result", frame_result);



		for(int i = 0; i < frame_gray->width; ++i)
		{
			for(int j = 0; j < frame_gray->height; ++j)
			{
				if((frame_result->imageData + frame_result->widthStep * j)[i] == 0)
				{
					(chain[num%numFrame]->imageData + chain[num%numFrame]->widthStep * j)[i]
= (frame_gray->imageData + frame_gray->widthStep * j)[i];
				}
				else
				{
					(chain[num%numFrame]->imageData + chain[num%numFrame]->widthStep * j)[i]
= (int)((1 - alpha) * (double)(chain[num%numFrame]->imageData + chain[num%numFrame]->widthStep *
j)[i] + alpha * (double)(frame_gray->imageData + frame_gray->widthStep * j)[i]);
				}
```

```
            }
        }
        //chain[num%numFrame] = frame_gray;
        TimeEnd = clock();
        //time[num_time] = TimeEnd - TimeStart;

        cout << "The processing time is: " << TimeEnd - TimeStart << endl;
        //cvWriteFrame(writer1, frame);
        //cvWriteFrame(writer2, frame_result);

        cvReleaseImage(&frame_result);

        num++;
        //if(num == numFrame)
        //    num = 0;

        //num_time++;
        //if(num_time == 1500)
        //    break;

        if(cvWaitKey(1) == 27) //press ESC to quit, fps = 33;
            break;
    }

    //Release the memory
     //cvReleaseVideoWriter(&writer1);
     //cvReleaseVideoWriter(&writer2);
     //char* filename = "D:\\ec720\\data\\KDE_MRF_1st_16F_4I_without_boundary_480.txt";
     //FILE* fp = fopen(filename, "w");
     //for(int i = 0; i < 1500; ++i)
     //    fprintf(fp, "%d\n", time[i]);
     //fclose(fp);
    cvReleaseCapture(&capture);
    //cvReleaseImage(&frame);
    cvReleaseImage(&frame_gray);
    //cvReleaseImage(&frame_next);
    //cvReleaseImage(&frame_result);
     //cvReleaseImage(&chain[numFrame]);
    cvDestroyWindow("origin");
    //cvDestroyWindow("processing");
     cvDestroyWindow("result");

    return 0;
}
```