THE UNIVERSITY OF LONDON

QUEEN MARY COLLEGE

An Autonomous Machine Vision System For Tracking Human Motion

by

Rashed Karim

DISSERTATION SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE IN PARTIAL FULLFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

QUEEN MARY COLLEGE

LONDON, UNITED KINGDOM

September 2005

Acknowledgements

This dissertation would not have materialized without the immense support of my parents and my loving wife. I would like to thank them for their patience.

I would also like to thank my supervisors: Dr. Tao Xiang and Professor Sean Gong for their support and time.

Along the way some people have helped and supported me from time to time. I would like to thank them for being so kind and helpful, especially Haseeb Qadir at Microsoft, and Dr. Chris Stauffer at the Massachusetts institute of Technology. .

.

Table of Contents

Chapter 5: The Kalman Filtering Process

Chapter 6: Implementation

Chapter 7: Results

Appendix A: References

Appendix B: Program code

ABSTRACT

Motion tracking of humans is a challenging problem in machine vision. Modeling human motion is not a trivial problem. The dynamics of human motion is unpredictable, since it is governed by an unknown number of parameters. Tracking becomes a more complicated problem in outdoor scenes, where numerous factors are controlling the scene in a very unpredictable way.

We wish to design a system that can robustly model an outdoor scene, and hence perform tracking. We also wish to apply Kalman filtering techniques, and analyze how well it performs when applied to tracking.

Chapter 1

Introduction

1.1 THE PROBLEM STATEMENT

Tracking objects in a video sequence has been of considerable interest to machine vision researchers for quite some time. Autonomous motion tracking has its roots in radar systems, where the luminous dots appearing on a black radar screen background, were tracked by a vision system. Due to slow processor speeds, tracking on a more intricate background, in real time, was still far away from becoming reality. Tracking has evolved over the years, with the development of new algorithms and improved processor speeds. Algorithms that are available today make it possible to track objects in outdoor scenes.

Motion tracking has had very useful applications in the industry. From tracking objects in an assembly line to surveillance systems, motion tracking has become a very powerful concept. More recently, human motion tracking has been on the limelight. Since humans are the principal actors in life's daily activities, motion tracking of humans is paramount to the success of a good surveillance system. Further, by tracking human motion we can use motion data to perform event inference, and justify any observable human behavior. Such surveillance systems are still under development.

As it may otherwise seem, tracking human motion is not a trivial problem. There are infinitely many situations that could arise in a video scene.

Humans quite often appear to walk in groups in video sequences. Most vision systems and tracking algorithms fail to deal with the problem of *occlusions*. An occlusion is a situation where a human or an object partially or completely blocks the optical pathway of the vision system's camera, to another human. In cases where occlusion occurs, motion tracking can get difficult for a number of reasons. A naïve system might not recognize occlusions completely. In such cases, these systems usually fail when the human group (causing occlusions between one another) disintegrates and walks away in different directions.

This certainly should not be regarded as the only scenario. It merely illustrates the nature of the problem that is needed to be dealt with. More complex situations could arise, such as the case where the scene could get relatively crowded with people.

PREVIOUS WORKS

Numerous papers in machine vision have addressed the topic of motion tracking. We will try to summarize the developments in this field, and point to the reader to important literature.

Background modeling is a prime component of any motion tracker. It has been used by machine vision researchers for quite sometime. A primitive model was laid out by Wren R.C. et al. [4] in their paper on a real-time system for tracking people. They suggested a single Gaussian background model; which was, however, very limited in the type of backgrounds it could model. It was only performed well in indoor backgrounds. Their work was followed later by Stauffer et. al. [7], at MIT, who showed that a mixture of Gaussians could be used to model an adaptive background. This worked very well in outdoor situations. Since then there have been other models, notably the median filter approach by Haritaoglu et. al. [9] .

The earliest motion tracking systems tracked objects on radar systems. These were followed by more advanced systems which started using optical flow techniques. With the advancement in background modeling, and processor speeds, machine vision researchers have started using more sophisticated techniques. Most researchers have shifted to detection-based tracking, where objects are extracted and their states identified. Other techniques, such as match-based techniques are useful for scenes where object detection and complete extraction is difficult. These use certain features of the object to perform tracking. More recent techniques have used Bayesian networks and Hidden Markov Models.

A handful of work [2], [3], [5], [6] has been of particular interest to us. Recently, Zhao, T. et. al. [2] work on tracking multiple humans used a novel approach to track humans in 3D using ellipsoid human shape models. Previously, rectangular models have been used [3] with little success. The work also presents a unique approach to estimating the "search space" of a given human in subsequent frames and tracking using Kalman filters, hence reducing the search time complexity significantly. Kalman filters are also used to handle occlusions. We have found the use of Kalman filters in tracking, particularly interesting, and this has motivated our work.

AIMS AND OBJECTIVES

We wish to implement a system that can track multiple human motions in complex situations, using a stationary camera. The system will process video input sequences, and will output the trajectories of humans in the scene. Since, the system tracks human motions in real-time, it will also employ the ability to process less than 10 frames per second, on a regular Pentium processor. Such a feat would require the system to be implemented using fast and efficient algorithms together with suitable data-structures.

The system will be implemented in C++ in the windows environment using Intel's OpenCV library [4]

Following are a list of objectives:

1. Identifying and removing background from the scenes, in order to expose the foreground pixels. A Gaussian model approach suggested by Zhao, T. et. al [2], which originally appeared in Yamada, et. al [8], will be used for background removal.
2. The head-tops (top of head) of human candidates in the scene are located. These head-tops could also well be head-tops of shadows and reflections cast by the human objects. Hence, we should be able to classify the head-tops.
3. We then perform shadow removal by first determining the shadow cast by the ellipsoid (the human model) on the ground [2]. We use our own techniques to efficiently remove the shadow.
4. We classify each human subject using an ellipse model.
5. The next step is to track each human object in the subsequent frames. We analyze two approaches, namely the vector analysis method, and using Kalman filters to enhance the prediction process.
6. Testing the system on real image sequences shot with an actual surveillance camera.

Chapter 2

The background model

2.1 THE DEVELOPMENT OF THE BACKGROUND MODEL

We first attempt to define the notion of a background in an image sequence. The background, as most might perceive, is not necessarily always the objects in a scene that remain stationary. In the real world, there could many situations such as changes in illumination, the case where an object moves in a periodic fashion (e.g. leaves of trees), etc. Such cases should be accounted for, and these shouldn't be classified as foreground.

The background changes as a scene progresses. If we take an example of us watching a video scene: we subconsciously, compute the background by differentiating the stationary and the non-stationary objects in a scene. As objects enter the scene, we normally tend to focus on the non-stationary objects. In a way, we ignore the stationary objects, since we keep adding them subconsciously to our own model of the background.

In order to develop the background model, we have implemented and experimented with two different models, namely:

- Single Gaussian background model
- Adaptive Gaussian mixture model

2.2 THE SINGLE GAUSSIAN BACKGROUND MODEL

The simplest way to model the background is by using a single Gaussian distribution [4] for each pixel in the background. The model is first trained on a set of images, which contain no moving pixels. Though this model cannot adapt itself to changes in the background, it is important to investigate how this trivial model performs in our dataset environment.

At any time *t* during the training period, what we know about a particular pixel located at (*i,j*) is it's history, which we denote by the set *H* [7]:

$$H(i,j,n) = \left\{ \begin{pmatrix} R_0 \\ G_0 \\ B_0 \end{pmatrix}, \begin{pmatrix} R_1 \\ G_1 \\ B_1 \end{pmatrix}, \begin{pmatrix} R_2 \\ G_2 \\ B_2 \end{pmatrix}, \ldots\ldots\ldots, \begin{pmatrix} R_t \\ G_t \\ B_t \end{pmatrix} \right\}, \text{ where } 0 \le n \le t$$

Since, our initial model uses single Gaussians; we can safely use a transformation to transform the *RGB* space into the intensity space. This mapping given by:

$$I : \begin{pmatrix} R \\ G \\ B \end{pmatrix} \mapsto \Re^+$$

Yamada et. al. [8] suggests the following intensity function, which we have adopted:

$$I = 0.95R + 0.64B + 0.3G \qquad \textbf{(2.1)}$$

Hence, at any time *t*, we also know the history of the computed intensity values of the pixel located at (i, j):

$$H_{\text{int}\,ensity}(i,j,n) = \{ I_0, I_1, \ldots\ldots\ldots, I_t \}$$

The background model can now be computed from these intensity values. The single-Gaussian background model B_s can be represented by an $n \times n$ matrix of Gaussian distributions, where the $i, j\,th$ entry corresponds to the pixel at location (i, j):

$$B_s = \begin{pmatrix} G_{0,0} & \cdots & \cdots & \cdots & G_{n,o} \\ \vdots & \ddots & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ G_{n,0} & \cdots & \cdots & \cdots & G_{n,n} \end{pmatrix}, \text{ where } G_{i,j} = N(\mu, \sigma) \text{ for the } (i,j)^{th} \text{ pixel}$$

Each of these Gaussians $G_{i,j}$ are the normal distributions centered around the pixel's mean intensity value, over the training set. The distribution parameter estimates are computed during the training period, using the observed pixel intensity values: $H_{\text{int}ensity}(i, j, n)$.

The most apparent implementation of this model is a 2-dimensional array of Gaussians. Each element of the 2-dimensional array corresponds to a pixel in the background and contains the single Gaussian. Fig 2.1 shows a representation of the background model.

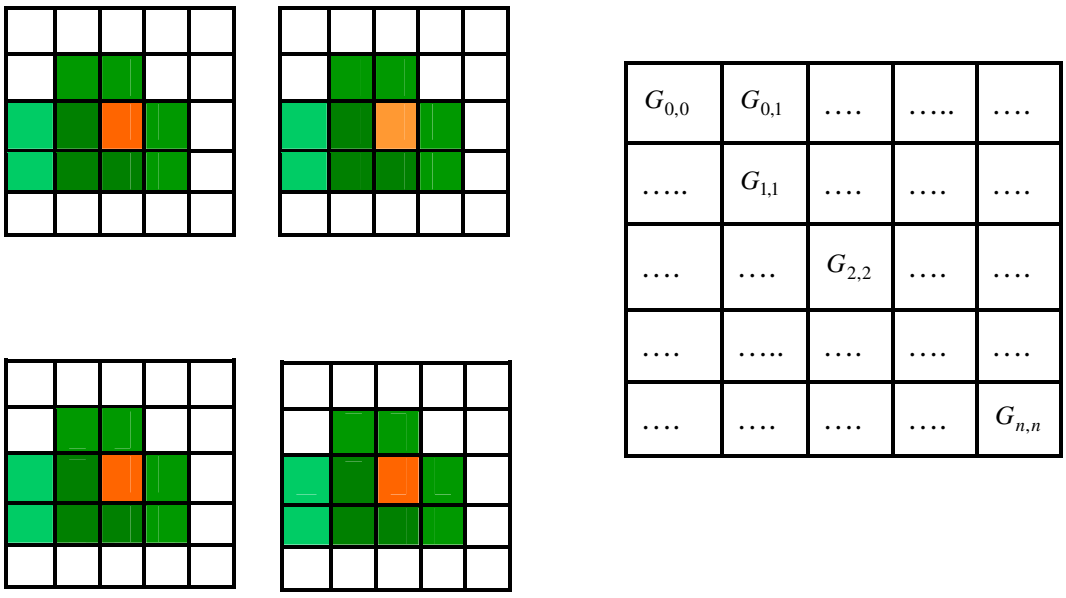


Figure 2.1: *The figure on the left shows a set of training images, and the figure on the right is what the background model would like when it is constructed completely from the training set. As an example,* $G_{2,2}$ *would be centered around the mean of the pixel intensity values of the orange pixels at location (2,2).*

Once the model is trained, it is used to classify pixels in the image sequence as either foreground or background. The foreground mask F of the k^{th} image in the sequence can be filtered out, using the following definition, where p is a pixel at location (i, j) in the k^{th} image of the sequence and $N(\mu, \sigma)$ is the corresponding Gaussian of the $(i, j)^{th}$ pixels in the background model. I is the intensity function:

$$F(k) = \left\{ p \,\middle|\, |I(p) - \mu| > n\sigma \right\} \qquad \textbf{(2.2)}$$

The single Gaussian model works well where there are no illumination changes, or frequent changes to the background. It can be manually re-initialized when changes to the background occurs. It is very swift at detecting the foreground mask, and its speed makes it a good candidate for real-time applications.

However, since we would like to deal with outdoor situations, where there are frequent changes in illumination and a constantly changing background, the single Gaussian model becomes inappropriate. It would require frequent manual re-initialization; as frequent as there are changes to the background. This would ultimately, defeat the purpose of having a motion tracker, which is to remote track human motion with minimal supervision.

The difficulty of modeling the background, in outdoor scenes, using a single Gaussian distribution model, has been illustrated in fig 2.2. In the outdoor environment, frames that are only a few minutes apart, could exhibit bi-modal intensity distributions

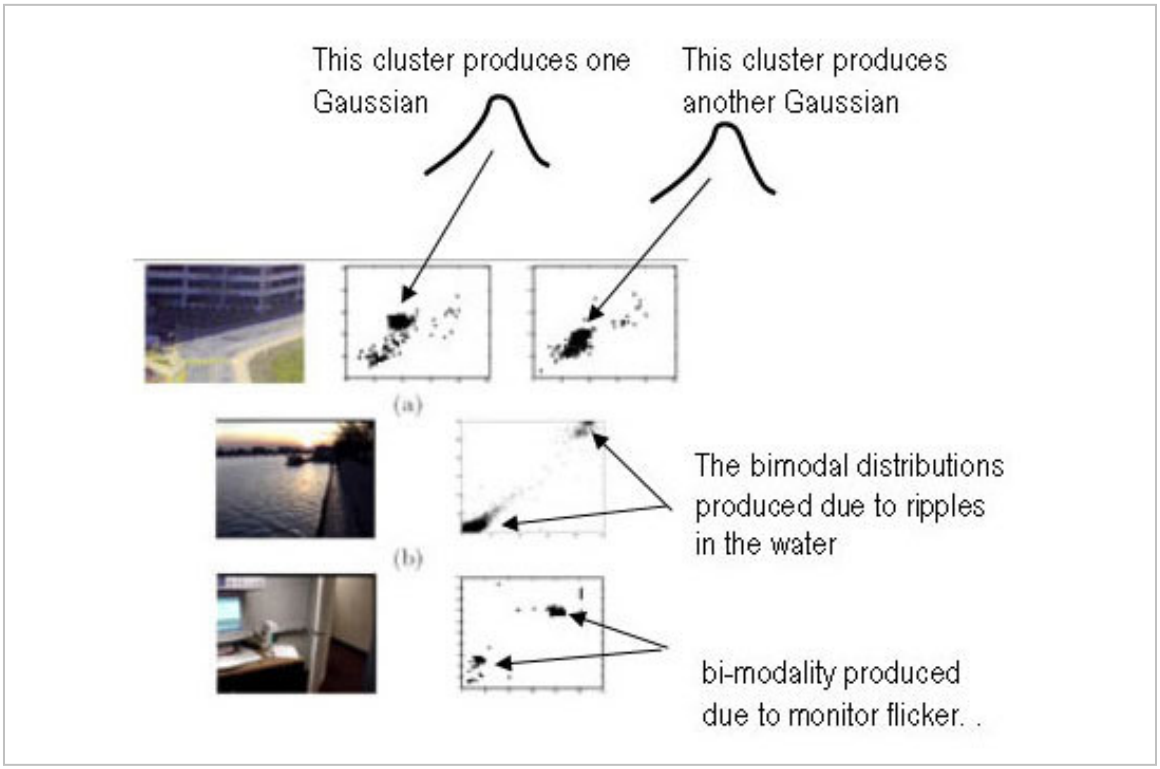


Figure 2.2: *These images have been adopted from Stauffer et. al.* [7]. *The scatter plots show the red and green values of a single pixel from the image over time. Notice how the values form clusters exhibiting bimodality.*

We implemented the single Gaussian model, and have found it to be inappropriate for the outdoor scenes that our tracker system is supposed to perform on. Most outdoor scenes frequently produce multi-modal distributions, as show in figure 2.2, and a more complex background model was required. In the next section, we will look at a more profound model that can capture and *record* such multi-modalities.

2.3 THE ADAPTIVE GAUSSIAN MIXTURE BACKGROUND MODEL

It is quite apparent that we require an adaptive background model with multiple Gaussians to model the background that occur in outdoors. Stauffer et. al. [7] describes such an adaptive background model. The model is capable of detecting and accounting for illumination changes and constantly moving objects (for e.g. leaves of trees, flickering monitor screen, etc).

The Gaussian mixture model uses unsupervised learning to learn the background; there is no initial training required, and more importantly, it adapts to scene changes. The motivation behind this method comes from scene changes and the need to break out of using just a single Gaussian to memorize the pixel's intensities. Consider the case of the moving leaves of trees in a scene. Leaves of trees frequently exhibit movement due to swaying of branches on a windy day. If we look more closely, a pixel, due to its movement, switches between the leaf color (green) to the another color (lets say blue) as a leaf moves in and out of the pixel. If we had used a single Gaussian model, we would have produced a Gaussian centered around the mean of the two different colors (green and blue). Whereas, the mixture of Gaussians model, would rather record a Gaussian centered around the leaf's color (green) and another Gaussian centered around the other color (blue) – hence recording the exact color/intensity changes, rather than an overall *average* of the changes, as in the single Gaussian model. Figure 2.3 illustrates this.

Illumination changes and scene changes can both be detected by change in pixel intensity levels. Since, we use more than a single Gaussian for each pixel, the most recent pixel intensity levels are stored using Gaussians, and assigned individual *weights.* A weight is a value that indicates how often the intensity has occurred in the pixel, in the past. A new intensity observed is given a low weight, whereas, an intensity that occurs frequently gradually attains a high weight.

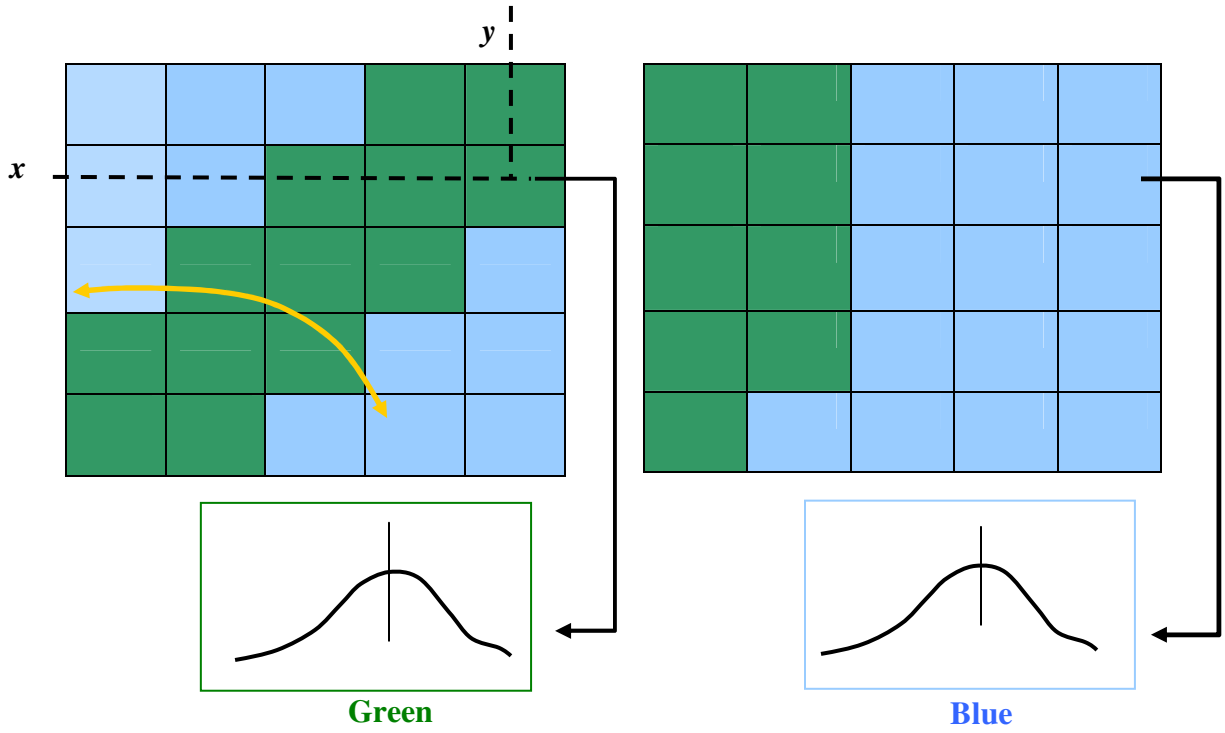


Figure 2.3: *The figure on the left shows the close-up of a leaf, marked by the yellow arrow that shows the direction of the to and fro caused due to the leaf's movement in the wind. The figure on the right shows how the leaf has now moved away from some of the pixels. The pixel, under examination, is the one located at (x,y). Notice, how its color distribution switches from a distribution centered around green to a distribution centered around blue. A good background model should record both the Gaussians.*

Given the fact that the most interesting scenes exhibit bi-modal distributions for its pixel intensities (see figure 2.2), we require a way to compute the respective Gaussians. The most common way of identifying separate Gaussian mixtures is by using an Expectation-Maximization (EM) algorithm. Nevertheless, The EM algorithm can become a bottle-neck in real-time applications, such as the motion tracker.

The EM algorithm starts out with a poor approximation of the Gaussian mixtures for the data clusters, gradually improving its approximation at each iteration. In a real-time application, waiting for the EM algorithm to achieve a good approximation for the Gaussian mixtures at each frame transition in an image sequence can become a very costly process. Moreover, an EM algorithm is more appropriate in situations where the clustered data remain *static* with no further additions. Real-time applications keep adding new data to the clusters or new clusters altogether, hence generating ever-changing Gaussian mixtures.

Stauffer et. al. describes an online K-means approximation, which we have adopted in our system. Every new pixel intensity value is checked against the existing distributions for that pixel, and incorporated into the distribution if a match is found, or otherwise, forms a new distribution indicating a new cluster. This forms the basis of our *adaptive* model.

At any time t, what we know is the history of a pixel at location (i, j)'s, intensity values:

$$H_{\text{int}\,ensity}(i, j, n) = \{I_0, I_1, \ldots\ldots, I_t\}$$

For each pixel at location (i, j), the background model at time t stores k Gaussian distributions, along with their weights $\omega_{k,t}$. This is called the Gaussian mixture $\psi(i, j, t)$, which can be represented by the set, where $G_k = N(\mu, \sigma)$ are normal distributions, as:
:

$$\psi(i, j, t) = \{\omega_{0,t} \cdot G_0, \omega_{1,t} \cdot G_1, \ldots\ldots\ldots\ldots, \omega_{k,t} \cdot G_k\},$$

The background mixture model B_M at time *t*, can be represented by an $m \times n$ matrix of Gaussian mixtures $\psi(i, j, t)$, where $0 \le i \le m$ and $0 \le j \le n$:

$$B_M(t) = \begin{bmatrix} \psi(0,0,t) & \cdots & \cdots & \cdots & \psi(m,0,t) \\ \vdots & \ddots & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ \psi(0,n,t) & \cdots & \cdots & \cdots & \psi(m,n,t) \end{bmatrix}$$

At time $t = 0$ we start with the empty background mixture model $B_M(0)$ where the Gaussian mixtures are $\psi(i, j, 0) = \{\omega_{0,0} \cdot G_0, , \ldots\ldots, \omega_{k,0} \cdot G_k\}$ and, $\omega_{0,0} = \omega_{1,0} = \ldots\ldots\ldots = \omega_{k,0} = 0$ and, $G_0 = G_1 = \ldots\ldots\ldots = G_k = N(0,0)$

:

$$B_M(0) = \begin{bmatrix} \psi(0,0,0) & \cdots & \cdots & \cdots & \psi(m,0,0) \\ \vdots & \ddots & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ \psi(0,n,0) & \cdots & \cdots & \cdots & \psi(m,n,0) \end{bmatrix}$$

At time $t = T$, for a pixel p at location (i, j) where $I(p)$ is its intensity value:

If it is the case that $\exists G_m(\mu, \sigma) \ni \left| I(p) - \mu \right| \le n\sigma$, where $G_m \in \psi(i, j, T)$ and $0 \le m \le k$, the weight, mean and variance are updated for the matched distribution G_m as follows, where α is the weight-learning rate[1], and ρ is the mean/variance-learning rate [7]:

Weight: $\omega_{m,T} = (1 - \alpha) \cdot \omega_{m,T-1} + \alpha$

Mean: $\mu_T = (1 - \rho) \cdot \mu_{T-1} + \rho \cdot I(p)$

Variance: $\sigma_T^2 = (1 - \rho) \cdot \sigma_{T-1}^2 + \rho \cdot (I(p) - \mu_T)^2$

[1] The weight-learning rate is the rate at which new pixel values should be incorporated into the existing model. A low weight-learning rate was used (~ 0.2), which indicates that new intensity values should be incorporated "slowly" into the model.

For the unmatched distributions: G_n where $n \neq m$, the weights are updated; the mean and variance remain unchanged [7].

Weight: $\omega_{n,T} = (1-\alpha) \cdot \omega_{n,T-1}$

Mean: $\mu_T = \mu_{T-1}$

Variance: $\sigma_T^2 = \sigma_{T-1}^2$

However, on the other hand, if it is the case that there are no matched distributions, i.e. :

$$\forall G_n(\mu,\sigma) \in \psi(i,j,T) \ni \left| I(p) - \mu \right| > n\sigma$$

Since, there are no matched distributions, the distribution that is least probable, i.e. with the minimum $\frac{\omega}{\sigma}$ ratio, is replaced with a new distribution with the mean as $I(P)$ and a high variance (typically 10.0).

PRELIMINARY RESULTS

We implemented the Gaussian mixture model, and have found that it performs very well in outdoor situations. Though there is still a training period, which lasts for about 100 frames, the model incorporates different pixel intensities rapidly, making it very suitable for tracking. We have used this model in our final tracking system. Results of background subtraction on actual images can be found in Chapter 7.

A complete description of our implementation is given in Chapter 6, section 6.2.

Chapter 3

Human segmentation and extraction

This chapter discusses in detail how our system handles the extraction of human subjects in a given scene. It lays out the importance of a robust human detector, and how difficult it can get to detect and extract humans. The later parts of the chapter deal with other issues such as shadow removal and identifying humans.

3.1 THE IMPORTANCE OF HUMAN EXTRACTION AND SEGMENTATION

We revisit the sole purpose of our system, which is to track humans in a given scene. The part of the system which performs tracking depends a lot on the accuracy of the human extraction model. Inaccuracies can lead into detecting un-interesting objects. The human segmentation and extraction model hence plays a very important role in motion tracking.

The remnants of background differencing are a residue of pixels which the system classifies as foreground. The residue frequently contains foreground pixels which does not necessarily represent human subjects. Shadows move alongside humans, and hence is left out by the background model in the residue. A good system should be capable of distinguishing between subjects and their shadows. Failure to do so would cause the system to track shadows inadvertently.

3.2 THE DIFFICULTIES OF HUMAN SEGMENTATION AND EXTRACTION

Extracting human subjects and segmenting them can be challenging for many reasons. The dynamics of the human body's shape is very unpredictable. It could undergo deformation, and sudden changes. Background differencing always leaves out a relatively small number of *patches* of pixels which are part of the background. These are caused due to the ever-changing illumination of outdoor scenes, which the background model takes some time to incorporate. We refer to these patches as *noise*. Noise can be removed significantly by the use of image filters and morphological operators. Figure 3.1 shows how we have removed the noise from a background differenced image.

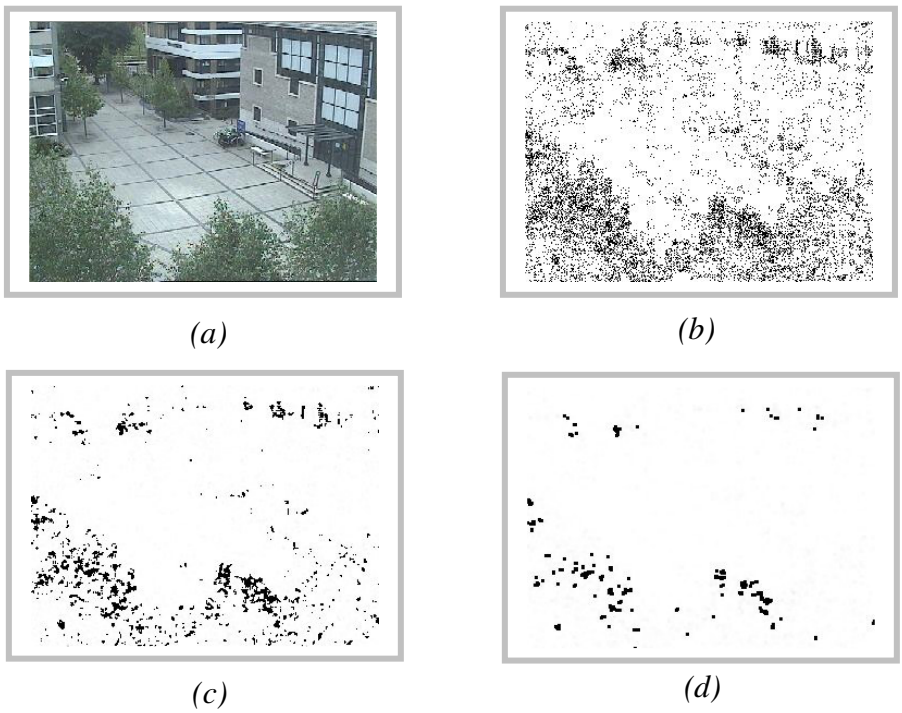


Figure 3.1: *(a) The original image of the courtyard. (b) is the image after background subtraction. Note that the black pixels indicate foreground. This snapshot was taken right after the system was trained on 17 frames. The system had not yet learnt the motion of the leaves of the trees completely, hence causing a majority of the leaves of trees to be treated as foreground. (c) is the image after image in (b) was filtered using a median filter. (d) is the image after (c) was filtered using a morphology close operator.*

Our system removes the noise in a two-step process: first passing it through a median filter and then performing a morphology close operation. Noise still remains after passing

it through the two filters (Figre 3.1 (d)), however, these are easily removed using a size filter, such as the one described in Section 3.3, equation 3.1.

A median filter, replaces a particular pixel with the median value of the color of its neighboring pixels. This removes most of the isolated one-pixel noises (black dots). What remains are bigger-sized noise pixels. Using a 3-by-3 rectangular structuring element, the morphology close operator performs well to remove the remaining noise. The use of such filters is not uncommon. It has appeared in the works of Zhao. T. et. al. [1]

The patches of black pixels, as seen in figure 3.1 (d), are called *blobs*. We will be using this term from here on, throughout our entire discussion. Blobs representing human subjects will be referred to as *human blobs*.

Removing noise from the background differenced image is not the end of the problems we face in human segmentation and extraction. There are countless other situations that might give arise to a false human hypothesis. It would be impossible to list out all such cases; however we point out a few very common ones that we have observed, and have attempted to solve.

(a) *(b)*

Figure 3.2: *figure (a) is a scene with three human subjects, where the bicycle rider is getting occluded by the human subject on the right. Figure (b) is the resulting image after background subtraction followed by filtering.*

Figure 3.2 is an example of a situation where occlusion can cause major problems. It is very difficult to hypothesize, by looking at the blobs, how many human subjects are actually there in the scene. The problem of occlusions is dealt with using prediction

models, such as the *Kalman filter*, which we introduce in section 5.3. Occluded objects follow the path predicted by the prediction model.

Apart from occlusions, we also face the problem of identifying a particular human subject, as h/she is being tracked across the scene. The human subject may cross the path of another human subject, in which case, the system can easily get confused between the two. Our system starts predicting when such occlusion arises, and captures the human object at a later frame, where it recovers from occlusion.

3.3 THE ALGORITHM FOR HUMAN EXTRACTION

A blob is a group of pixels $b = \{p_0, \ldots\ldots, p_n\}$, in any image frame, that lies within a certain *contour,* where each pixel $p_i = (x, y)$ is identified using pixel positions x and y. A contour is a closed boundary. A blob *b* may be classified as a human blob, if its total pixel area lies within a certain range. The set of all human blobs *H*, in any image frame that contains a set of blobs *B* where $b \in B$, can be represented as:

$$H = \left\{ b \;\middle|\; T_L \le \sum_{i=0}^{n} (\omega_i \bullet p_i) \le T_U \right\} \qquad \textbf{(3.1)}$$

In equation 3.1, ω_i is the *world* area occupied by the i^{th} pixel. Due to the perspective effect, some models may wish to take ω_i into account, and assign different areas to different pixels. The perspective effect causes certain pixels, which represent parts of objects nearer to the camera, to have lesser *world* area. Pixels, representing objects which are farther away, should have a higher ω_i value. Since our system employs a camera with a relatively small visibility range, and also due to its proper positioning, employing a model that takes this perspective effect into account, is unnecessary. Hence, we assign an area of 1 unit to each pixel. The T_U and T_L are upper and lower limits of the area threshold respectively. Typically, values of $T_L = 200$ and $T_U = 500$ have produced good results.

Equation 3.1 is essentially our *size filter*, and it removes most of the unwanted noise propagating from background subtraction and successive filtering.

At this point, what remains are potential human blobs. However, sudden changes in illumination may cause large areas of the background to show up as foreground. Such areas may pass the size filter, in which case, it will get treated as a human blob. However, our background model is very quick to adapting to such changes, and this only poses a minimal threat. Improvements may be made, such as using a threshold value which can indicate if human segmentation and extraction should be stopped completely until the illumination change is incorporated; and extraction started at a later time when a lesser percentage of the image is classified as foreground. Such situations are common if human extraction is started prematurely during the background training period. For tracking purposes, it is important that human extraction is started at a later time. Our system starts human extraction once 100 frames have passed.

Efficient ways of extracting human blobs have appeared in various literatures. Javed et. al. uses bounding rectangles [10] and so does many others [3],[5],[9]. Zhao. T. et al. models a human using an ellipse, and uses the top of the head for extracting human blobs. We have adopted this method of using ellipses to model humans.

As discussed earlier in section 3.2, occlusions pose a major problem when segmenting human blobs. A perfect segmentation technique which works for any blob is still an open problem. However, Zhao T. et. al. attempts to find a fair approximation of the location of humans in the blob. The argument set forth argues that the head top of humans is least likely to be occluded when they are walking in groups. This way, finding head-tops in a blob gives yield to the human(s) in the blob. After a head-top is found an ellipse is drawn, vertically downwards from the head-top, to capture the human.

3.4 FINDING HEAD TOPS

The head-top of a human blob $h \in H$, where $h = \{p_0, \ldots\ldots, p_n\}$ and $p_i = (x, y)$ can be defined as all pixels which are a local minimum point in y, within a certain neighborhood Ω of x defined by $\Omega = \{(x, y) \mid x - S \le x \le x + S\}$. The constant S is typically the size in pixels, of an average human head. We have used S values ranging from 8-10 pixels.

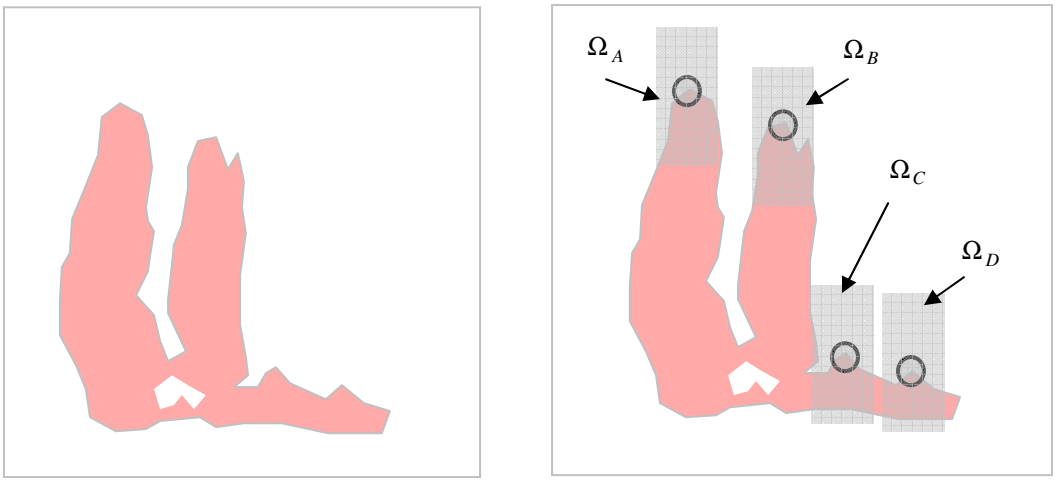


Figure 3.3: *Figure (a) shows the human blobs as they appear after background subtraction and successive filtering. Figure (b) shows the head-tops identified using the algorithm we discussed in section 3.2*

Figure 3.3 illustrates how the algorithm provides us with a fairly good approximation of the human blobs. Local minimums within the neighborhoods Ω_A and Ω_B identifies the human blobs correctly. However, the neighborhoods Ω_C and Ω_D are *false* head-tops which lie within the shadow pixels. Zhao T. et. al. [1],[2],[11] removes these false hypotheses by performing a *geometrical shadow analysis*. Shadow analysis is a two-step process. The position of the sun is determined, during that particular time of the day. The shadow area cast by each ellipsoid is then determined and head tops lying within that area is removed.

Determining the sun's co-ordinates at any time of the day and performing the geometric computations is cumbersome, and we have devised a simpler method that has performed well on the blobs that we have tested with. We calculate the *elliptical* area underneath a head-top candidate that is also part of the blob. If the area underneath exceeds a certain threshold limit T_H, the local minimum qualifies as a head-top. The threshold limit T_H is typically the average pixel area occupied by a human subject *scaled* accordingly. The scaling is required due to the perspective effect. T_H is small for human subjects lying far away from the camera. We multiply the threshold with a scaling factor s, which scales it down, if the y co-ordinate of the local-minimum is far away from the camera. The scaling factor can be written as $s = \frac{y}{h}$, where h is the height of the image. The threshold without the scaling effect, is the elliptical area which can be calculated using the formula: $\pi \times a \times b$, where a is the length of the major axis of the ellipse, and b the length of the minor axis.

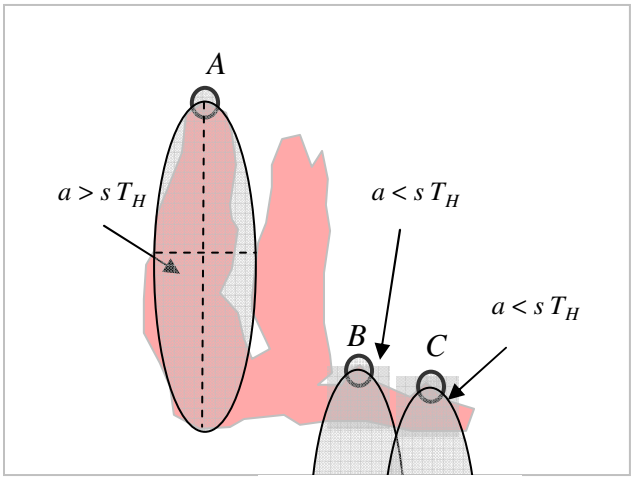


Figure 3.4: *The figure shows the area that is being calculated under each head-top candidate. We have omitted the head-top candidate in neighborhood Ω_B from figure 3.3 for brevity. The area of each region underneath the head-tops is denoted using a.*

Figure 3.4 shows why head-top B and C would not pass our area threshold test – the blob area underneath does not exceed the threshold limit $s\,T_H$.

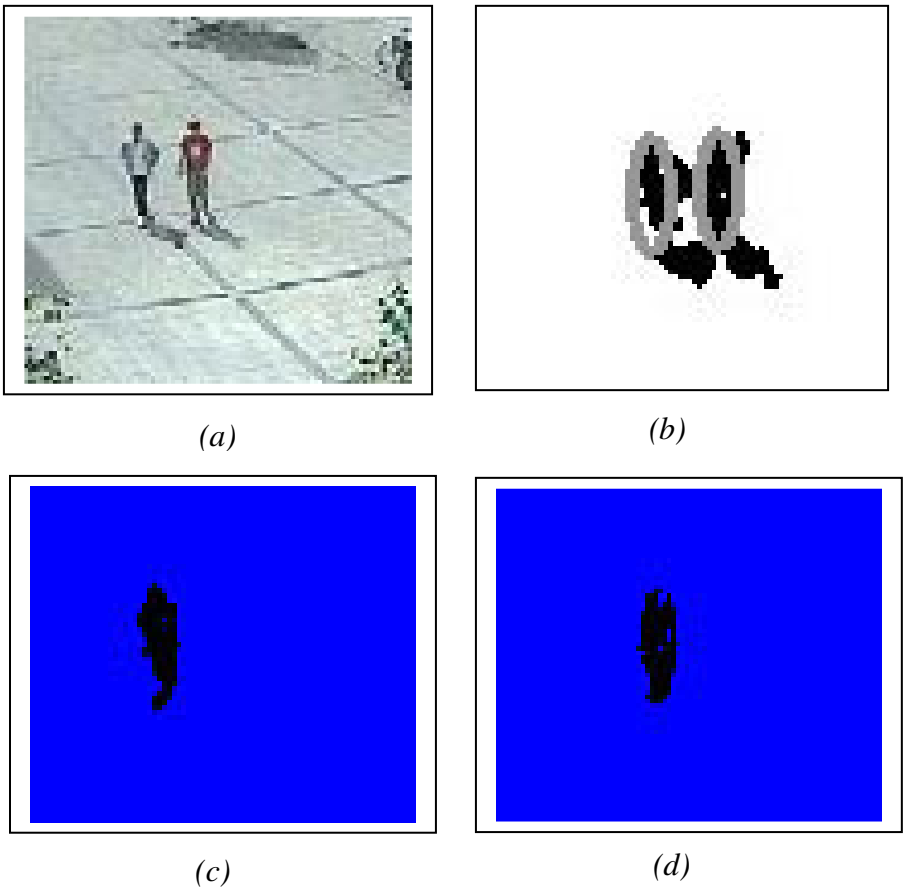


Figure 3.5: *A successful human blob extraction and segmentation performed by our system. The original image in fig (a) is first background differenced and filtered (median, close and size) to produce what is in fig (b). Head-top candidates are identified and ellipses are drawn to model possible human subjects. Shadows are completely removed in figure (c) and figure (d), separating out the two humans blobs.*

We have designed our own algorithm for segmentation and extraction, which we have described in Chapter 6, Section 6.3. The implementation of the algorithm has produced very good results. We have also analyzed its time-complexity in section 6.4, and have found it to be very suitable for real-time motion tracking. The test results on human segmentation and extraction can be found in Chapter 7.

Chapter 4

Motion Tracking

4.1 INTRODUCTION

The sole purpose of our system is to *track* the motion of humans. To "track" is to trace the trajectory produced by humans in a given scene. Tracing the path taken by a human in a scene is a trivial task. However, the problem lies in trying to make the system associate a certain path to a human. The problem gets even more intricate when complex situations arise, such as humans intercepting paths of one another, or when humans occlude one another, when viewed from the camera viewpoint. Our system addresses each of these issues and handles them robustly.

4.2 THE PURPOSE OF MOTION TRACKING

Given that a system is able to successfully identify and associate the different trajectories to the different humans that produce them - such trajectories can be used to persistently store and later analyze how humans locomote, especially when they exhibit abnormal behavior. An intelligent surveillance system is bound to be deployed with a good tracking system. Alarming situations can be easily detected and prevented, by using such a system. However, our work does not aim to classify human motion, and differentiate between normal and abnormal motion. This is a psychological issue, and models can be developed further to predict such abnormalities.

The noise that frequently emerges in blob detection, after background filtering, poses a difficult challenge to detecting motion by simply keeping track of where a certain blob appears in a certain frame. On a windy day, for example, clouds frequently block the

sun's rays, causing sudden illumination changes. This causes background clutter. A blob which is being tracked, may completely disappear for a certain number of frames, and then reappear sometime later. A robust system should be able to predict and track the blob's motion for the period for which it disappeared. There are also situations when not whole, but parts of a blob are detected. If the blob parts do not meet an *area threshold* (average pixel area occupied by a human in our scenes), they are discarded and the blob that was being searched for, is considered lost. However, if they do satisfy the threshold, since not every pixel on the blob represent the human, they only give us an approximate position of where the human has moved to. Figures 4.1 illustrates this:

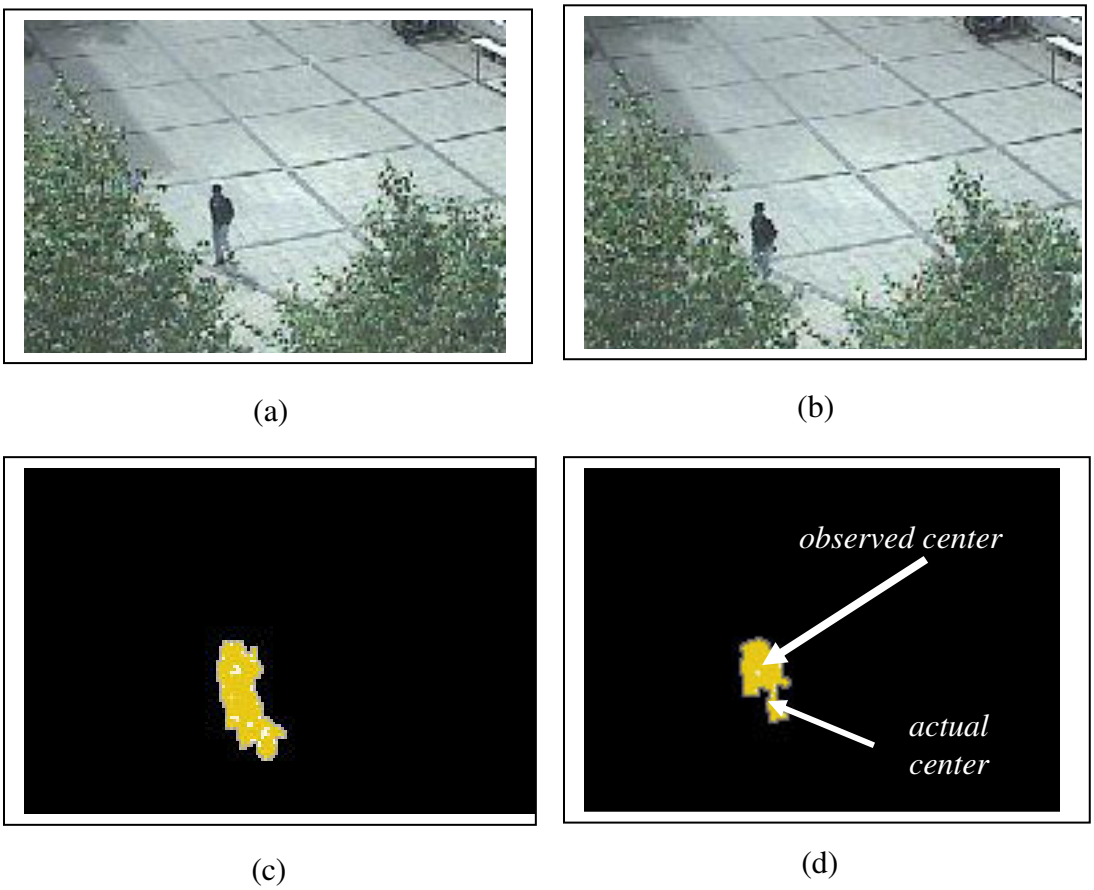


Figure 4.1: *Frames (a) and (b) are separated by 5 frames. Notice how the human figure walks into behind the tree in frame (b). Frames (c) and (d) are the blobs produced, after background subtraction, from frames (a) and (b) respectively. If the blob in frame (d) does not meet the area threshold, it may be discarded and be considered as background noise. On the other hand, if it is accepted, notice how the observed center differs from the actual center. Tracking using the center can be difficult for these reasons. Our system uses head tops, as its tracking point.*

4.3 TRACKING AS A PROBABILISTIC INFERENCE PROBLEM

Humans, in normal circumstances, tend to exhibit trajectories that are regular. This makes it easier to predict their motion, given that we know the velocity and the direction in which they are moving.

Tracking can be a viewed as a probabilistic inference problem [16]. We can model the humans as having some state at any frame *i*. We will denote the state of a human object, at the i^{th} frame, using a random variable X_i. The state of a human object X_i is the *actual* position of the human, at the instant the i^{th} frame image was captured. A human object, in its lifetime of *n* frames, passes through states:

$$\{X_0, X_1, \ldots\ldots\ldots\ldots, X_n\}$$

Our system is not always capable of getting the exact *actual* position (due to background clutter, noise, occlusions, etc.). However, it can determine the *observed* states of the human object. An observed state is the position at which the human objects appear to be, in the background differenced image. We sometimes also call it a *measurement* state. We denote it using the random variable Y_i, where Y_i is the observed state of a human object at the i^{th} frame.

Our tracker system tries to predict the state of a human object, using previous observed states. More generally, we are trying to determine a representation of:

$$P(X_i \mid Y_0 = y_0, \ldots\ldots\ldots\ldots, Y_{i-1} = y_{i-1}) \qquad \textbf{(4.1)}$$

This step is what is more popularly known as the *prediction step* [16]. Equation 4.1 tries to identify the present actual state of the human object, using previous measurement states. It assumes the fact that we already have knowledge of the measurement states of our human object for states 0 to $i-1$. From this, it derives the immediate future state.

A better prediction of the current actual state X_i can be obtained if we also use the current measured state Y_i. Hence:

$$P(X_i \mid Y_0 = y_0, \ldots\ldots\ldots\ldots, Y_{i-1} = y_{i-1}, Y_i = y_i) \qquad \textbf{(4.2)}$$

This step is known as the *correction step* [16]. Before we design algorithms for prediction and correction, we make a few assumptions that simplify our task considerably [16]:

- **The state of a human X_i is dependent only on its previous state X_{i-1}:** More formally put: (Note that the assumption causes a simplification)

$$P(X_i \mid X_1, \ldots\ldots\ldots\ldots\ldots, X_{i-1}) = P(X_i \mid X_{i-1})$$

- **The current measured/observed state Y_i is independent of all other measurement states: $\{Y_0, \ldots\ldots\ldots\ldots\ldots, Y_{i-1}\}$, given that we know current state X_i.**

Moving away from viewing tracking as a probabilistic inference problem, we now discuss the approaches we used to develop a simple tracking model, based on the assumptions and ideas represented above.

4.4 DEVELOPMENT OF THE TRACKER

Assuming that we were properly able to extract human subjects, using the human segmentation and extraction techniques presented in section 3.4, our tracker should be able to robustly measure, predict and approximate the human states. The ideas that we employ, are the ones that are represented using probabilistic inferences, as stated in the previous section: section 4.3.

To explain how we develop a simple motion tracker, we begin with a simple scene, where one single human subject makes an entrance, moving at an arbitrary velocity and changing directions, and eventually finally exiting the scene. Figure 4.2 illustrates this scene. Keeping this scene in picture, we will derive our motion model in the following sections.

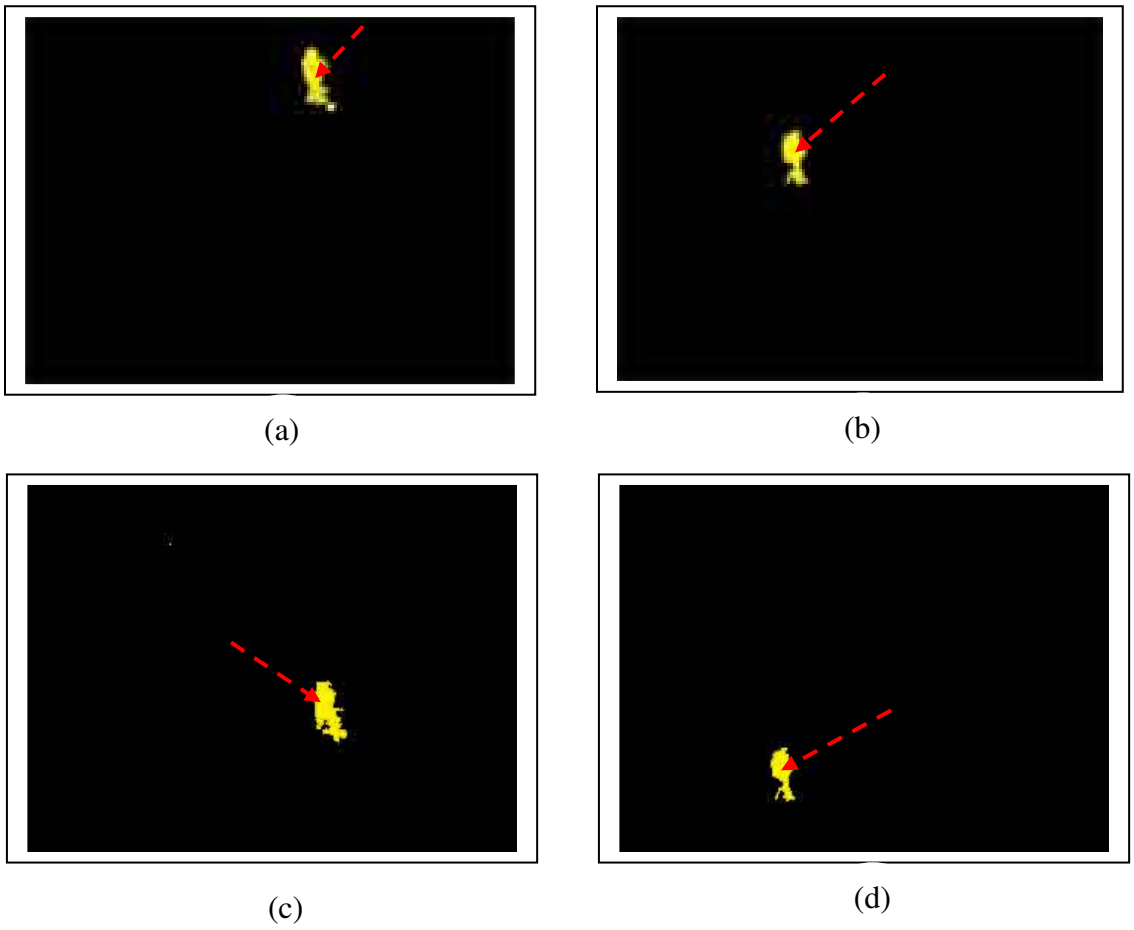


Figure 4.2: *The frames above show how the human segmented blob enters the scene in (a), and moves, changing directions in subsequent frames (b),(c) and (d). The red arrows indicate the direction of motion between the previous and the current frames. Notice how the shape of the human blob is constantly changing.*

4.5 DETECTION AND REGISTRATION OF A NEW BLOB

A blob *b*, which can be thought of as a group of pixels $b = \{p_0, \ldots\ldots, p_n\}$, in any image frame, which lies within a certain contour where each pixel $p_i = (x, y)$ is identified using pixel positions x and y. A blob *b* may be classified as a human blob, if its total pixel area lies within a certain range. The set of all human blobs *H*, in any image frame, can be represented by equation 3.1 (Section 3.3). :

Earlier in Chapter 3, section 3.3, we have discussed how we extract human blobs from a scene. The tracker uses the same principles to detect and extract a human blob. Once a human blob is detected, it is checked to see if it lies within a *hotspot.* A hotspot is associated to a registered human blob. It is simply a search neighborhood Ω [1] within which any registered human blob is expected to be observed in the consecutive frame. For a human blob centered at (a, b) it is defined as:

$$\Omega = \{(x, y) \mid (x-a)^2 + (y-b)^2 \leq r^2\}$$

The radius of the hotspot r can depend on the speed at which the human blob is moving. We have used radius values typically ranging from 5-10 pixels, which have produced good results.

At any frame instance i, our system keeps track of *n* search neighborhoods for *n* human blobs:

$$\{\Omega_0, \Omega_1, \Omega_2 \ldots\ldots\ldots\ldots, \Omega_n\}$$

When a blob falls within a search neighborhood Ω_i, it gets associated to the search neighborhood's blob, if it *appears* to be the same human blob. This is a problem which still remains to be solved. Zhao. T. [1] uses a texture template for object representation. Others [10] have used similar constructs.

A blob, which is classified as a human blob (equation 3.1), and which does not fall within any Ω_i is registered as a new blob.

4.6 PARAMETER INITIALIZATION OF NEWLY REGISTERED BLOBS

A newly registered blob does not have sufficient parameters to enable it for tracking. The system requires further vital information about its motion, to initiate tracking. After a new blob is registered, when it is first observed in a frame; the newly registered blob is searched to make a second appearance in the subsequent frame, within its neighborhood Ω. If it is found, the velocity is computed and the direction of motion is established. Figure 4.3 illustrates this:

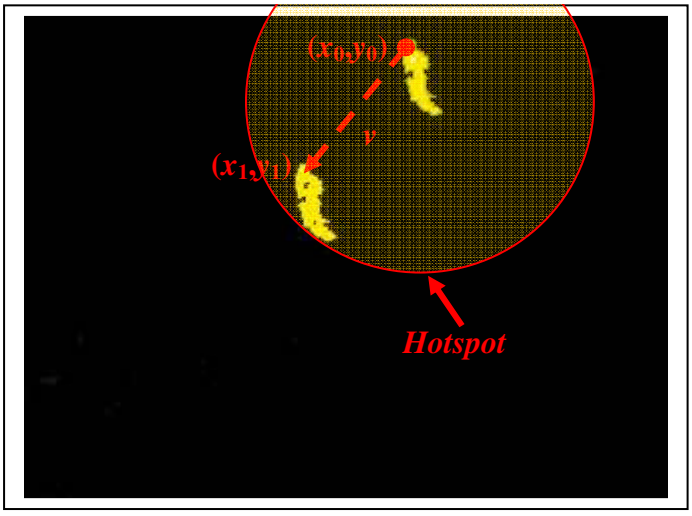


Figure 4.3: *The human blob is first observed at position* (x_0, y_0). *In the next frame, it is searched and found to lie within its hotspot, at* (x_1, y_1). *The velocity, at which the blob is moving, can now be calculated, and hence we have obtained useful information about its motion.*

A human blob may also appear completely deformed in the second frame and may not pass the area threshold test (Equation 3.1). As a result, no human blob will be found to lie within the hotspot. This makes it impossible to calculate the speed and the direction at which the blob is moving. Prediction of its next appearance cannot be computed, and hence the human blob, which was being initialized for tracking, is discarded. We expect the blob to re-appear in later frames, and for its motion to get detected properly at some point in its *lifetime*.

4.7 MOTION PREDICTION

Predicting motion of a human blob is an iterative process. It primarily involves estimating the subsequent positions of the blob, using *a priori* information such as velocity, directions of motion and the *observed* position. The noisy nature of foreground blobs, propagating from background differencing, makes it difficult to rely solely on the observed position. Our system tries to reduce the errors in the observed position by averaging out the observed and the *estimated* position. The estimated position is the position which the system computes using velocity and direction of prior motion.

We use vector analysis to predict motion. Lets assume that a blob is first detected at a frame $i = 0$, and lets denote its initial pixel position as (x_0, y_0). Let's also assume that at frame $i = 1$ the blob is again detected at (x_1, y_1). At this point, we have enough information to calculate the speed and direction at which it is moving.

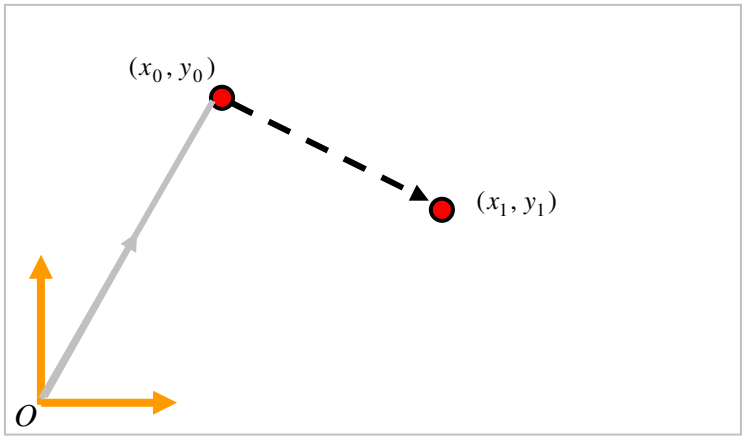


Figure 4.4: *The first two positions of a blob is crucial for determining its speed and direction of motion.*

More generally, the speed s_i of the human blob, at any frame *i* is calculated using the frame rate γ:

$$s_i = \| (x_i, y_i) - (x_{i-1}, y_{i-1}) \| \bullet \left(\frac{1}{\gamma} \right) \qquad \textbf{(4.3)}$$

The velocity v_i can now be calculated by normalizing the motion direction vector, and multiplying it with speed s_i:

$$v_i = s_i \cdot \begin{pmatrix} \dfrac{x_i - x_{i-1}}{\sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}} \\ \dfrac{y_i - y_{i-1}}{\sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}} \end{pmatrix}$$

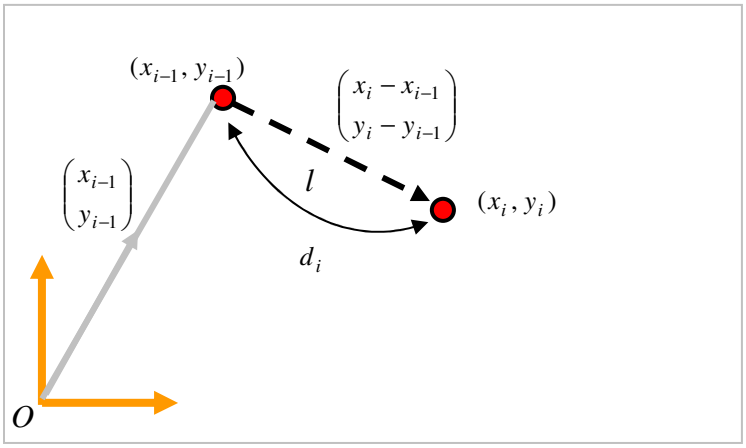


Figure 4.5: *The origin* (x_{i-1}, y_{i-1}) *and the next observed position* (x_i, y_i) *is all what is required to start predicting subsequent positions.*

The vector equation of the straight line *l* along which the blob is moving is given by:

$$l : \begin{pmatrix} x_k \\ y_k \end{pmatrix} = \begin{pmatrix} x_{i-1} \\ y_{i-1} \end{pmatrix} + \lambda_i \begin{pmatrix} x_i - x_{i-1} \\ y_i - y_{i-1} \end{pmatrix} \quad , \lambda_i \geq 0 \quad \textit{and} \quad k \geq i-1 \qquad \textbf{(4.3)}$$

Notice that the scalar parameter constant λ_i when set to 1 gives (x_i, y_i). If the distance traveled between points (x_{i-1}, y_{i-1}) and (x_i, y_i) is d_i, a λ value of 1 represents a distance d_i traveled from the blob's origin (x_{i-1}, y_{i-1}) to (x_i, y_i).

Using these information, we can easily interpolate the next position (x_{i+1}, y_{i+1}), the blob is expected to move into.

To calculate (x_{i+1}, y_{i+1}), we need to determine the value of the parameter constant λ_i. From our argument above, since we know that d_i represents $\lambda = 1$. If the blob has moved a distance of d_{i+1}, the distance traveled from the origin (x_{i-1}, y_{i-1}) now totals $d_i + d_{i+1}$, which can represented by a λ_i value of $\left(\frac{d_i + d_{i+1}}{d_i}\right)$.

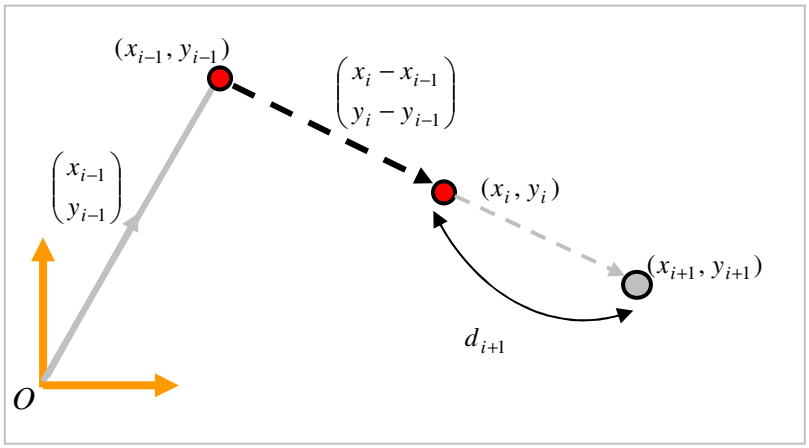


Figure 4.6: *The point to be predicted (interpolated) is* (x_{i+1}, y_{i+1}). *We use* d_{i+1} *calculated using* v_i *to determine the value of* λ_i

However, it is important to note that we are only making a prediction of the distance moved d_{i+1}, using the speed s_i, observed in the previous frame. The parameter constant λ_i can now be written as:

$$\lambda_i = 1 + \left(\frac{d_{i+1}}{d_i}\right) = 1 + \left(\frac{s_i \times t}{d_i}\right) = 1 + \left(\frac{s_i \times \frac{1}{\gamma}}{d_i}\right) \qquad \textbf{(4.4)}$$

Our surveillance camera records images at the rate of 10 frames per second, hence our tracker uses $\gamma = 10$.

Predicting *a posteriori* blob position using *a priori* information such as velocity and direction of motion, only gives us the predicted position (x_{i+1}^p, y_{i+1}^p) where we expect the blob to be in the next frame. The observed position (x_{i+1}^o, y_{i+1}^o) could be different from the predicted position. A tracking system could assign appropriate *weights* to each of the predicted and observed positions, and compute an estimated position $(\hat{x}_{i+1}, \hat{y}_{i+1})$ of where the human blob actually is. Without performing such an estimate and neglecting the observed position, we have discovered that it is impossible to do tracking. Such estimates have greatly improved our tracking results.

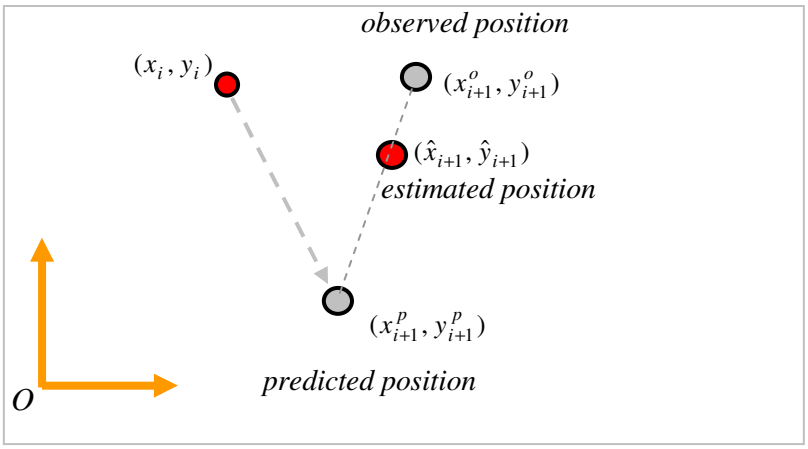


Figure 4.7: *The estimated blob position lies somewhere in between the line connecting the predicted and observed position points. The exact location of the estimated point is determined by the estimation parameter called the "tracker reliability factor".*

Depending on how much we think our system is accurate in predicting the human blob's position, we assign normalized weights to the predicted and the observed positions accordingly. Our system uses a *tracker reliability factor* (α).

$$\begin{pmatrix} \hat{x} \\ \hat{y} \end{pmatrix} = \alpha \begin{pmatrix} x_p \\ y_p \end{pmatrix} + (1-\alpha) \begin{pmatrix} x_o \\ y_o \end{pmatrix}$$

We use a value of $\alpha = 0.8$. From our experiences, α should be tested on various motion sequences, before assigning it a constant value.

4.8 THE LIFETIME OF A HUMAN BLOB

All human blobs have a certain *lifetime*. This is the period of time that elapses between the time when the blob makes its first appearance, till the time when it exits the scene. A human blob can be represented using a *Deterministic Finite State Machine (DFSA)*. It undergoes a series of state transitions during its entire lifetime. A human blob starts off with an initial state, when it is newly registered by the tracker. We call this the *"new blob*" state. If the blob can be initialized for tracking (i.e. observed in the second frame hence enabling calculation of velocity and motion direction) it transits to the "*tracked*" state. While it is being tracked, it may disappear for a few frames due to noise; in that case it has transited to the "*lost*" state, and we keep predicting its motion, this time without an observed point. The blob at this point may re-appear at a later frame, in which case it again reverts back to the "tracked" state. However, if it never re-appears again and the *lost tolerance threshold is exceeded*, it plummets to the "*discard*" state, after which it is wiped out from the system. Figure 4.8. illustrates this using a DFSA.

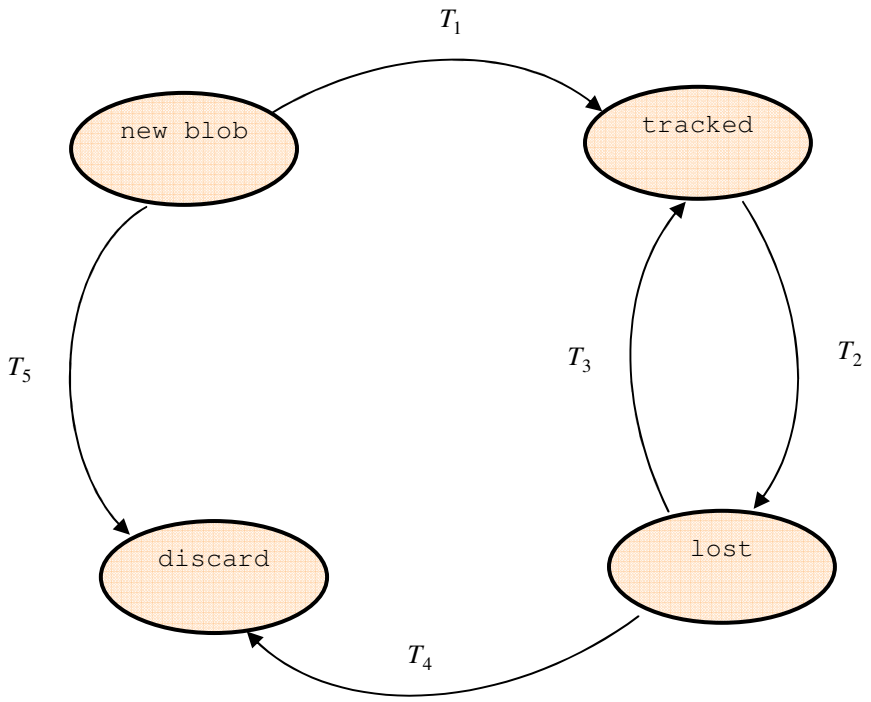


Figure 4.8: *Here are the transition states: T_1-blob is initialized completely for tracking, i.e. it is observed in the second frame and its velocity has been calculated. T_2-blob disappears after being tracked for sometime. T_3-blob reappears and has survived the lost tolerance threshold. T_4-blob has crossed the lost tolerance threshold. T_5-was not detected in the second frame, and hence initial velocity and direction could not be computed.*

From our discussion of blob states, it is important to note that blobs which are lost, during tracking, have their positions predicted completely based on its immediate prior value of velocity and direction of motion. Regular non-lost blobs are estimated and tracked normally, using a weighted average of predicted and observed points.

4.9 PRELIMIARY RESULTS

After implementing a tracker which performs prediction, wholly based on vector analysis of linear motion, has produced disappointing results. In the next chapter, we present our findings and explain why we require a better method for prediction. We introduce a *Kalman filtering process* that has known to be widely used in tracking. In later chapters (Chapter 6, 6.9) we provide implementation of both approaches, where the second approach (Section 6.9, code 6.17 – 6.22) uses both vector analysis and Kalman filtering process techniques.

Chapter 5

The Kalman filtering process

5.1 INTRODUCTION

Predicting a human blob's motion using vector analysis, works well when a human blob is *visible* throughout its entire lifetime. The visibility decreases with increasing illumination changes. The system also experiences poor visibility when the scene is dark, and the human blob's texture color is similar to its surroundings. As discussed earlier in the previous chapter, these situations cause the blob to completely disappear, since it cannot suffice the requirements of the *area threshold test*, during the human segmentation and extraction process. A blob may also disappear when it gets occluded by an object that is in the optical pathway of the camera and the blob. The system ought to tackle these problems, since they are commonplace in outdoor scenes.

5.2 THE NEED FOR A FILTER

Human blobs' position prediction and estimation calculations rely heavily on its observed positions. Highly inaccurate observed positions could suddenly cause a human blob to become lost, and eventually causing the system to exhaust its search for the lost blob, in which case it is discarded. We have analyzed and tried to reason about such inaccuracies and have found that most inaccuracies in observed positions are caused due to ill-formed human blobs.

The cause of an ill-formed blob can be attributed to the background model. Irregular deformation of a blob, after background subtraction, occurs when parts of the actual human figure are conceived to be a part of the background. Since our system uses Gaussian distributions to compare between the background and the foreground, finding a match (to a background Gaussian) for a pixel which actually represents the foreground is *not* unusual. Figrue 5.1 illustrates this.

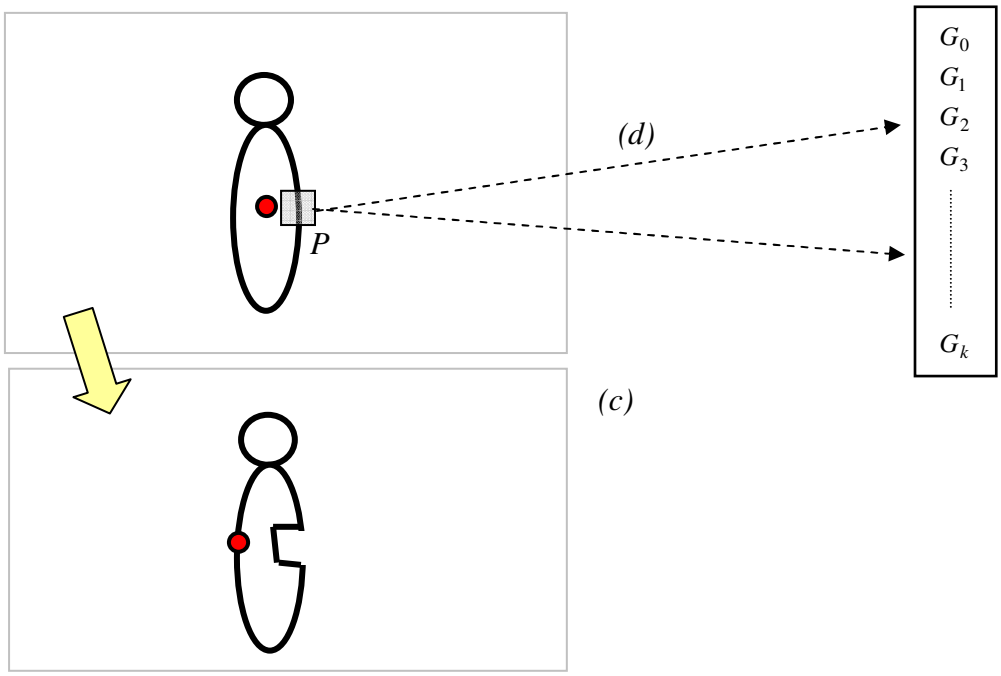


Figure 5.1: *This is a graphical monologue of the series of events showing how human blobs get "badly" deformed, eventually causing a shift in the observed position marked by the red dot. Fig-(a) are all the k-Gaussian distributions which store past frequently recurring intensities of pixel P. In other words, fig- (a) models the background intensities for pixel P. The part of the foreground fig-(b) which falls within pixel P, can have an intensity value which matches two or more Gaussians in fig-(a), as shown by arrows fig-(d). The result, is the system "thinking" that P is a part of the background, causing the deformation in fig-(c), and hence a shift in the center of the blob – which is our observed point.*

The deformations presented in figure 5.1 are minor, occurring frequently, and caused by changes in scene illumination. More severe deformations could cause the human blob to completely disappear, resulting in no observed point, and forcing the system to start predicting "*blindly*". Since we only use the immediate past (Equations 4.1, 4.2) for prediction, a noisy observed direction of motion in the immediate past could cause the system to start predicting subsequent positions in the wrong direction. Hotspots also move with the predicted positions. All these events might lead to losing a blob completely, even if it re-appears again at a later frame. We have tested our system without using a filter, and we present what we have discovered in Figures 5.2 and 5.3.

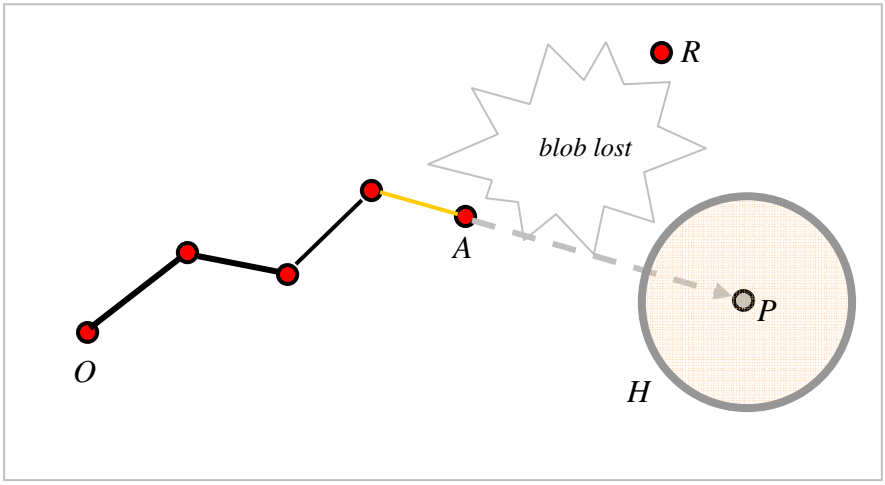


Figure 5.2: *Tracking starts off at O, with the usual "noisy" observed center, until the observed point disappears at A, leaving the system to start predicting the subsequent positions in the direction of the immediate past (orange line). At some later frame, subsequent predictions reach point P with hotspot H. If the blob ever re-appears, which it does at R, it will not be detected by Hotspot H, hence eventually causing the blob that was being tracked to be discarded. However, the blob at R is recognized by the system and is tracked as a new separate blob.*

We can alleviate the severity of the problem and the pitfalls of "blind" prediction if we are able to *smooth* the path taken by the human blob using a filter. The path from O to A in figure 5.2 can be smoothed, possibly causing prediction to occur in the desired direction, and eventually having the hotspot around the region, where the blob re-appears. Though smoothing does not guarantee perfect prediction, it definitely improves the prediction process. Figure 5.3 shows how smoothing achieves this.

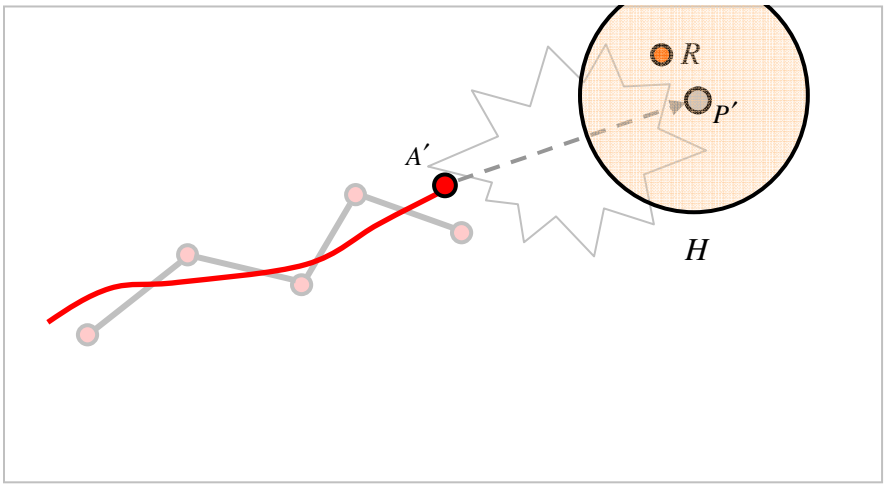


Figure 5.3: *The dark red line is the path which can be obtained from the Kalman filter. A′ is now the point at which the blob disappears, and prediction starts from there on. Prediction occurs in the desired direction enabling the hotspot to encompass the blob's position when it re-appears again at a later frame. Henceforth, facilitating the blob to be tracked again.*

Using such filtering techniques is not uncommon. It has appeared in the works Zhao. T. et. al. [1] [4]. They use the *Kalman filter* [13], invented by the mathematician R.E. Kalman in the 1960s. The Kalman filter has been known to be very powerful in its ability to perform estimates for past, present and future states by minimizing the mean squared error. The Kalman filter becomes more attractive for dynamical systems such as the motion of a human blob.

5.3 ORIGINS OF THE FILTER

The Kalman filter has its roots in the more popular *least square method* of minimizing error. The basic ideas behind the filter are derived from this method. We show how the least square develops into the Kalman filter. Due to the filter's numerous possible applications in computer systems, it makes it necessary to speed-up the computations required to compute the filter. We also show how this is being done.

5.4 LEAST SQUARES METHOD

In the least squares method [14], we attempt to find a "good" estimate for a state a by making a sequence of measurements of a. Our measurements are bound to contain some error, and hence the least square method primarily concerns minimizing this error over all measurements. We give an example [15] of an attempt to measuring the water levels of a tank.

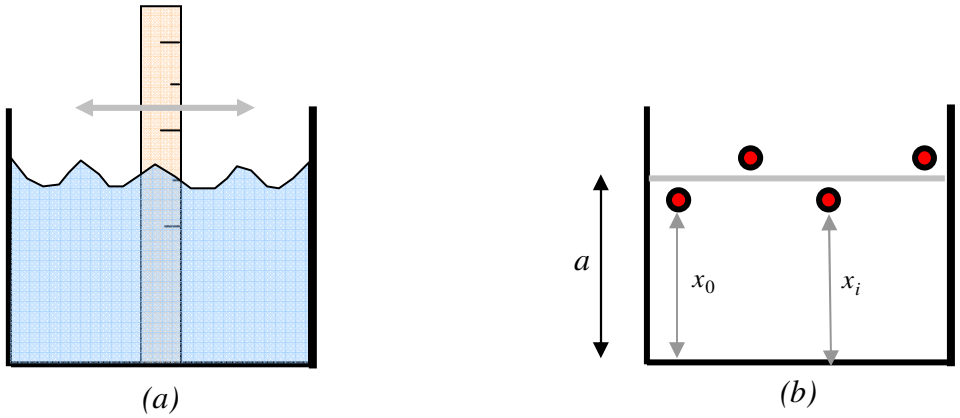


Figure 5.4: *Fig-(a) shows how we make measurements of the water levels in a tank. We use a ruler to make measurements at different times i, and possibly at different places. Fig-(b) is a pictorial representation of our measurements. We are trying to estimate state a.*

The error resulting from a single measurement i is given by:

$$E_i = (x_i - a)^2 \tag{5.1}$$

The total error over all measurements is a summation of the individual errors:

$$E = \sum_i E_i = \sum_{i=1}^{n} (x_i - a)^2 \tag{5.2}$$

We are interested in minimizing this error E, and finding the value of $a = \hat{a}$ for which E is minimum, i.e. $\frac{\partial E}{\partial a} = 0$:

$$\frac{\partial E}{\partial a} = 2\sum_{i=0}^{n}(x_i - \hat{a})$$

$$= 2\left(\sum_{i=0}^{n} x_i - n\hat{a}\right)$$

$$\sum_{i=0}^{n} x_i - n\hat{a} = 0 \Rightarrow \boxed{\hat{a} = \frac{1}{n}\sum_{i=0}^{n} x_i} \qquad \textbf{(5.3)}$$

However, this method of finding estimates becomes computationally expensive for a real-time system such as a motion tracker. For every estimate $\hat{a}_i$, we need to compute first $\sum_{k=0}^{i-1} x_k$ and then add it to x_i, to get $\sum_{k=0}^{i} x_k$. We could save the computations required by giving a recurrent relation for $\hat{a}_i$:

$$\text{Since, } \frac{1}{i-1}\sum_{k=0}^{i-1} x_k = \hat{a}_{i-1}$$

$$i\,\hat{a}_i = \sum_{i=0}^{n} x_i = \sum_{k=0}^{i-1} x_k + x_i = a_{i-1} + x_i$$

$$\hat{a}_i = \frac{i-1}{i}\hat{a}_{i-1} + \frac{1}{i}x_i$$

$$\boxed{\hat{a}_i = \hat{a}_{i-1} + \frac{1}{i}(x_i - \hat{a}_{i-1})} \qquad \textbf{(5.4)}$$

Using equation 5.4., we are able to recursively calculate $\hat{a}_i$, using the prior estimate $\hat{a}_{i-1}$. We have essentially removed the summation which we required previously, over all prior measurements.

Equation 5.4 can be modified to represent a higher-order dimensional system, with vectors representing their states and measurements. Equation 5.4 now becomes [15]:

$$\hat{a}_i = \hat{a}_{i-1} + K_i \left(X_i - H_i \, \hat{a}_{i-1} \right) \tag{5.5}$$

The matrix $H_i \in \Re^{m \times n}$ relates the state *a* to the measurement *x* [15]:

$$x_i = H_i \bullet a \quad \Rightarrow \quad \left[x_1 \quad x_2 \quad \ldots \quad x_n \right]^T = \left[H_1 \quad H_2 \quad \ldots \quad H_n \right]^T \bullet a \tag{5.6}$$

The *Kalman gain* K_i can be calculated using [15]:

$$K_i = P_i \, H_i^T \text{, where } P_i = \left(H_i \, H_i^T \right)^{-1} \tag{5.7}$$

However, now that we have solved these equations, we still have not accounted for the noise which occurs in our motion tracking system, and in just any other dynamical system. We represent the *process* and *measurement* noises using the random variables w_k and v_k respectively. They are independent of one another and assumed to be distributed normally with zero mean [12]:

$$w_k \sim N(0, Q_i) \tag{5.8}$$

$$v_k \sim N(0, R_i) \tag{5.9}$$

Rewriting the state and measurement equations (5.5 and 5.6), with added noise:

$$\hat{a}_i = A \, a_{i-1} + w_k \tag{5.10}$$

$$x_i = H_i a_i + v_i \tag{5.11}$$

Note that the square matrix $A \in \Re^{n \times n}$, relates the state at *i*-1 to the state at *i*.

The recurrent relation in $\hat{a}_i$, now becomes:

$$\hat{a}_{i-1} = A_i\,\hat{a}_{i-1} + K_i\,(x_i - H_i A_i\,\hat{a}_{i-1})$$

This equation can be solved [12],[13], giving the *Kalman gain* K_i:

$$K_i = P_i' H_i{}^T\,(H_i P_i' H_i{}^T + R_i)^{-1}$$

where: $P_i' = A_i\,P_{i-1} A_i{}^T + Q_{i-1}$

and, $P_{i-1} = (1 - K_{i-1} H_{i-1})\,P_{i-1}'$

Note that Q_i and R_i are the parameters of the normal distributions of noise (equations 5.8 and %.9).

5.5 THE APPLICATION OF THE KALMAN FILTERING PROCESS IN MOTION PREDICTION

Moving away from the mathematical abstraction of the Kalman filter, we now state how it may be used to predict the motion of the blob. First we define some parameters of blob motion. The state s_n of a blob at any time is its tracking point (x, y) (usually center of the blob, or head top), together with its velocity (v_x, v_y). We may write s_n in vector form as:

$$s_n = \begin{bmatrix} x_n & y_n & (v_x)_n & (v_y)_n \end{bmatrix}^T$$

If we were to use a constant velocity model, assuming that all humans move with a constant velocity, i.e. $v_n = v_{n-1}$, we could formulate the following:

$$x_n = x_{n-1} + \Delta t \cdot (v_x)_{n-1}$$
$$y_n = y_{n-1} + \Delta t \cdot (v_y)_{n-1}$$

We could re-write these equations in matrix form as [11]:

$$\begin{bmatrix} x_n \\ y_n \\ v_x \\ v_y \end{bmatrix} = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_{n-1} \\ y_{n-1} \\ (v_x)_{n-1} \\ (v_y)_{n-1} \end{bmatrix} \quad \textbf{(5.12)}$$

However, we cannot disregard the noise that is involved in observing a state. As previously, the process and measurement noises can be represented using normal probability distributions with zero mean: $w_n \sim N(0,\Sigma)$ and $v_n \sim N(0,\hat{\Sigma})$, respectively. Σ and $\hat{\Sigma}$ are the co-variance matrices of process and measurements, respectively. With added noise, equation 5.12 now becomes:

$$\begin{bmatrix} x_n \\ y_n \\ v_x \\ v_y \end{bmatrix} = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_{n-1} \\ y_{n-1} \\ (v_x)_{n-1} \\ (v_y)_{n-1} \end{bmatrix} + w_n$$

Now we state our measurement state $\hat{s}_n$, which follows from the actual state s_n with added measurement noise v_n:

$$\hat{s}_n = s_n + v_n$$

or more simply as,

$$\begin{bmatrix} \hat{x}_n \\ \hat{y}_n \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} x_n \\ y_n \\ (v_x)_{n-1} \\ (v_y)_{n-1} \end{bmatrix} + v_n$$

The time-interval Δt, is essentially the frame rate γ, Our camera shoots at 10 frames a second.

The most important elements of the Kalman filter process are the process and measurement noise parameters. At each new iteration, they are updated. The update equations have been omitted for brevity, and the reader may wish to read W. Greg et. al. discussion on the Kalman filter [13].

Using the Kalman filter and the state-measurement equations, as described above, we can achieve a better and more reliable level of motion prediction. To summarize up the entire process in simple terms, we first measure the initial position of the blob and its velocity (together put what is called the *initial state* s_0 of the blob). This can be done using the equations we have set forth in section 5.5. This is our initializing step. In the subsequent frames, we make a prediction, *only* using Kalman filtering. Prediction is followed by measurement and we determine the observed position of the blob. The measurement and prediction is then compared to assess how *well* we predicted. A simple pixel-wise difference between measurement and prediction positions is sufficient. This assessment of prediction updates the measurement and process noise parameters accordingly, hence preparing the Kalman filter to make *a better* next prediction.

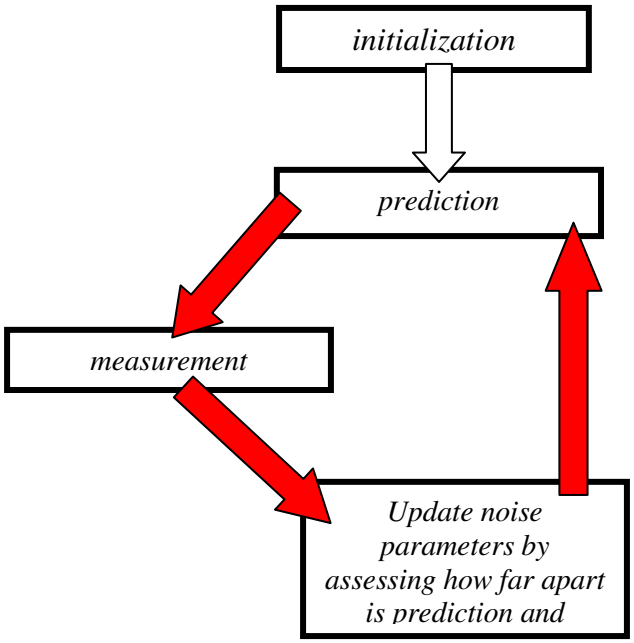


Figure 5.5: *The Kalman filtering process cycle in motion prediction*

Chapter 6

Implementation

6.1 INTRODUCTION

In this chapter we present to the reader important details of our implementation. The tracker system is divided into three different libraries namely: the background subtraction model, the head-top finder and the motion tracker. This chapter is divided into three separate sections, where each section represents a different library.

Major parts of the implementation description in this chapter include actual program code from our system. We sometimes have omitted few parts of code for conciseness. For lengthy code, we have used algorithms to describe the implementation.

The implementation was written using the C++ language. We have used Intel's OpenCV library [17] for common image processing and vision functions. Since C++ is relatively platform and compiler dependent, it may also be worth noting that our development was carried out in the Microsoft Windows® environment, using Microsoft® Visual C++ 6.0.

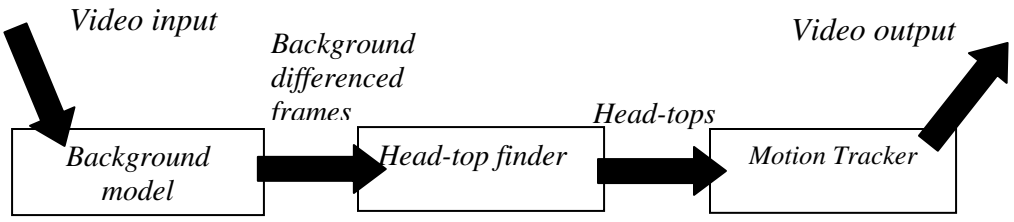


Figure 6.0: *An overall representation of our system and its data flow.*

6.2 THE BACKGROUND MODEL IMPLEMENTATION

For most parts of the implementation description of the background, we will be discussing the data structures we have constructed to fit our background model. These data structures correspond to the mathematical set and matrix forms we have laid out in Chapter 2, section 2.3. The background model algorithm adopted, is attributed to the works of Stauffer et al [7]. However, we have provided our own independent implementation of the algorithm using some efficient data structures.

We restate the Gaussian mixture $\psi(i, j, t)$ for a pixel at location (i, j) of frame t, as the set of Gaussian distributions $G_k(n, \sigma)$ with assigned weights $\omega_{k,t}$:

$$\psi(i, j, t) = \{ \omega_{0,t} \cdot G_0, \omega_{1,t} \cdot G_1, \ldots\ldots\ldots\ldots\ldots, \omega_{n,t} \cdot G_n \} \qquad \textbf{(6.1)}$$

The implementation of a Gaussian distribution $G_i(n, \sigma)$ is a C++ class with appropriate get/set functions.

```
1:    class GaussianDistribution
2:    {
3:          double variance;
4:          double mean;
5:          double weight;
6:    public:
7:          bool match(double);
8:          double getWeight();
9:          void updateDistribution(double, double);
10:         void adjustWeight(double, bool);
11:         double getWeightVarianceRatio();
12:         void setDistribution(double, double, double);
13:   };
```

Code 6.1: *The class definition for a Gaussian distribution and its function prototypes from the GaussianDistribution.h file.*

A few functions are worthy of a description:

The `updateDistribution(double pixelValue, double learningRate)` function updates a Gaussian distribution with the new pixel value, which is the intensity value of a pixel as defined in section 2.2. It updates it at a rate specified by `learningRate`. The update equations are discussed in Chapter 2, section 2.3. We require updating Gaussians during the training period to incorporate new pixel values, which are observed at a new training frame. Updating Gaussians also becomes necessary after the training period, when the background model is constantly adapting itself to new changes in pixel values.

The `adjustWeight(double learningRate, bool matched)` function adjusts the weight based on equations set forth in section 2.3, and according to *evidence* specified by `match`. At any frame, if a Gaussian distribution has been found to have a match (criteria for a match defined in Chapter 2, equation 2.2, it is said to have an evidence. Its weight is then updated according to the `learningRate`. Other Gaussians for which evidence was not found have their weights updated at a different rate, i.e. specified by `1-learningRate`.

The implementation of a Gaussian mixture $\psi(i, j, t)$, is essentially a container for storing its k Gaussians distributions. It can be represented using an array of `GaussianDistribution` objects. Code 6.2 illustrates the implementation of the mixture. The constructor, on lines 6-14, instantiates `s` Gaussian distributions. The class provides functions, such as `updateGaussians(double pixelValue)` which iterate through each Gaussian in the container, and searches for a match. If a match is found, the Gaussian is updated by invoking the Gaussian's `updateDistribution( )` function. Otherwise, it finds the *least probable distribution* (minimum ω/σ ratio) and replaces it with a new Gaussian distribution by calling the container's `replaceDistribution(int, double, double, double)` function. The function parameters are parameters for the new distribution.

```
1: class GaussianContainer
2: {
3:    GaussianDistribution** distPtr;
4:    int size;
5: public:
6:    GaussianContainer(int s)
7:    {
8:          distPtr = new GaussianDistribution*[s];
9:          for (int i=0;i<s;i++)
10:         {
11:               distPtr[i] = new GaussianDistribution();
12:         }
13:         size = s;
14:   }
15:   bool updateGaussians(double);
16:   int getSize();
17:   int getLeastProbableDistribution();
18:   void replaceDistribution(int, double, double, double);
19: };
```

Code 6.2: *The class definition for a Gaussian mixture and its function prototypes from the GaussianContainer.h file.*

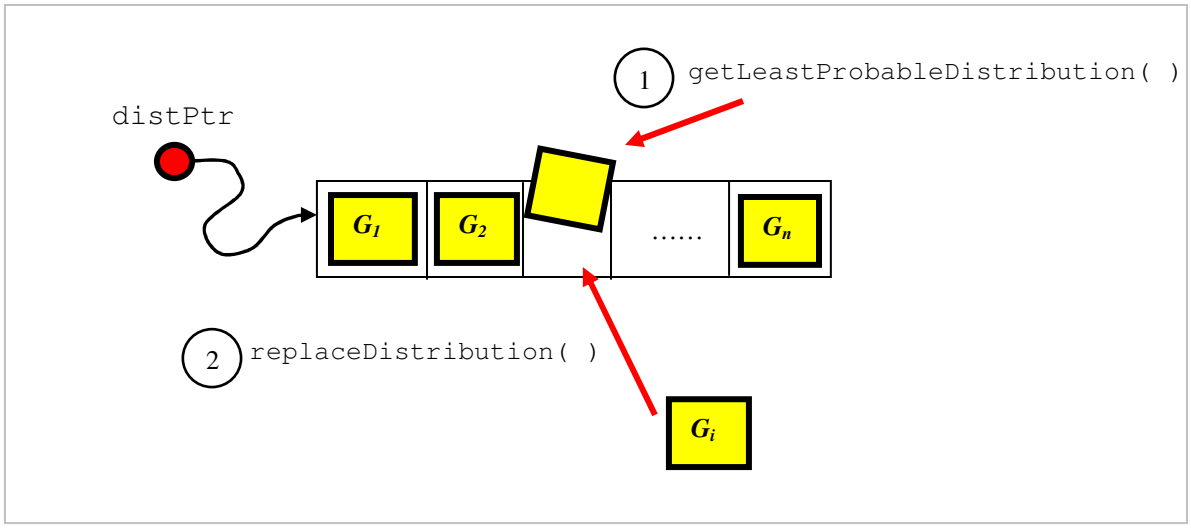


Figure 6.3: *The figure shows the two steps involved in updating Gaussians when a match is not found. Also note the structure of a Gaussian mixture.*

The `GaussianDistribution` and `GaussianContainer` structures can now be used to construct our background mixture model. Constructing the mixture model has essentially been a bottom-up approach, by first constructing the smaller entities such as the `GaussianDistribution`. We revisit the background mixture model, as described in Chapter 2 section 2.3. The background mixture model B_M, at time *t*, can be represented using an $m \times n$ matrix of Gaussian mixtures $\psi(i, j, t)$, where $0 \le i \le m$ and $0 \le j \le n$:

$$B_M(t) = \begin{bmatrix} \psi(0,0,t) & \cdots & \cdots & \cdots & \psi(m,0,t) \\ \vdots & \ddots & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ \psi(0,n,t) & \cdots & \cdots & \cdots & \psi(m,n,t) \end{bmatrix}$$

The implementation of such a structure is a two-dimensional array of Gaussian mixtures. Since a Gaussian mixture is represented by a the `GaussianContainer` class, our background mixture model is a two-dimensional array of `GaussianContainer` objects. It is represented by the `MultipleGaussianBackgroundModel` class.

```
1: class MultipleGaussianBackgroundModel
2: {
3:    GaussianContainer*** containerPtr;
4:    int height;
5:    int width;
6:    public:
7:    MultipleGaussianBackgroundModel(IplImage* img)
8:    {
9:          height = img->height;
10:         width = img->width;
11:         createEmptyModel(height,width,
12:                          NUMBER_OF_GAUSSIANS_IN_MIXTURE_MODEL);
13:   }
14:   void updateModel(IplImage*, IplImage*);
15: };
```

Code 6.3: *The class which represents the background mixture model. A few details have been omitted for brevity.*

Code 6.3, shows the constructor of the background mixture model class. The constructor accepts the image frame, as its parameter. These image frames are the 3-channel color frames from the actual CCTV video. The height and the width of the image correspond to our background mixture model B_M matrix's dimensions *n* and *m* respectively. An *empty* model is created using a private function, the details of which have been omitted. This is only an initialization step. The empty model is the one we have described in Chapter 2, section 2.2.

An important function that deserves attention is the `updateModel( )` function which updates all the Gaussian containers of the background mixture model. The function takes in the incoming 3-channel image frame from the video sequence as an input parameter. It also takes in a second 3-channel image, which is to writes to as output. The output is the current underlying background model. The update functions for the containers: `MultipleGaussianBackgroundModel`, `GaussianContainer`, and `GaussianDistribution` are a hierarchy of functions where one invokes the other, to eventually update the entire background model. This hierarchy is shown in figure 6.4.

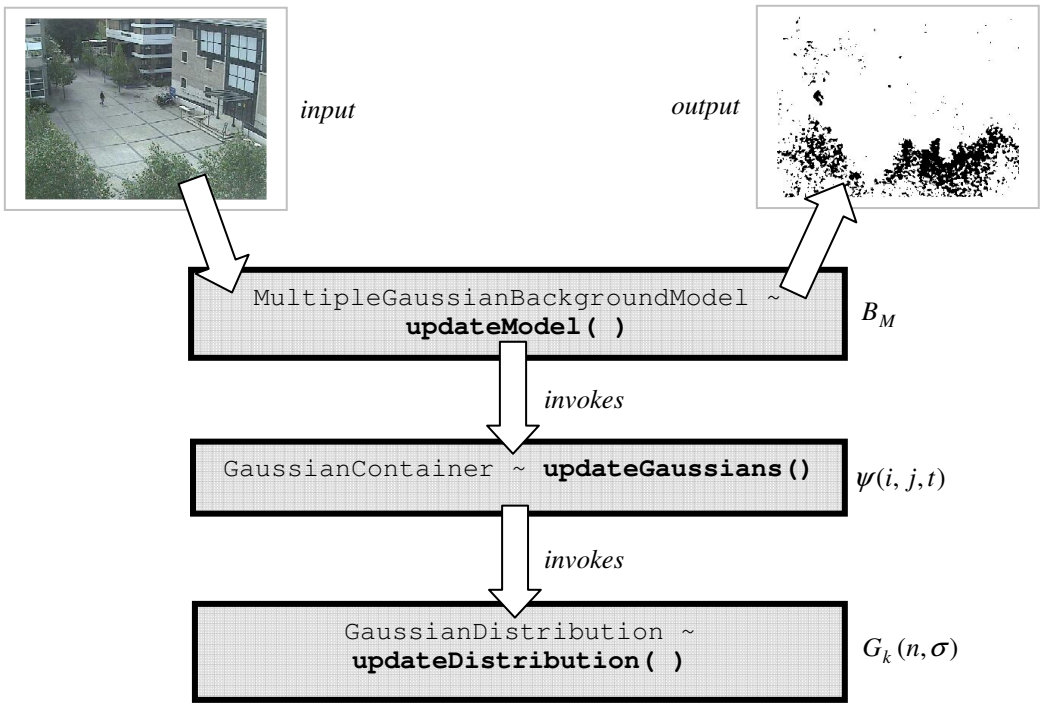


Figure 6.4: *The hierarchy of update function calls. The update produces the background differenced image as its output.*

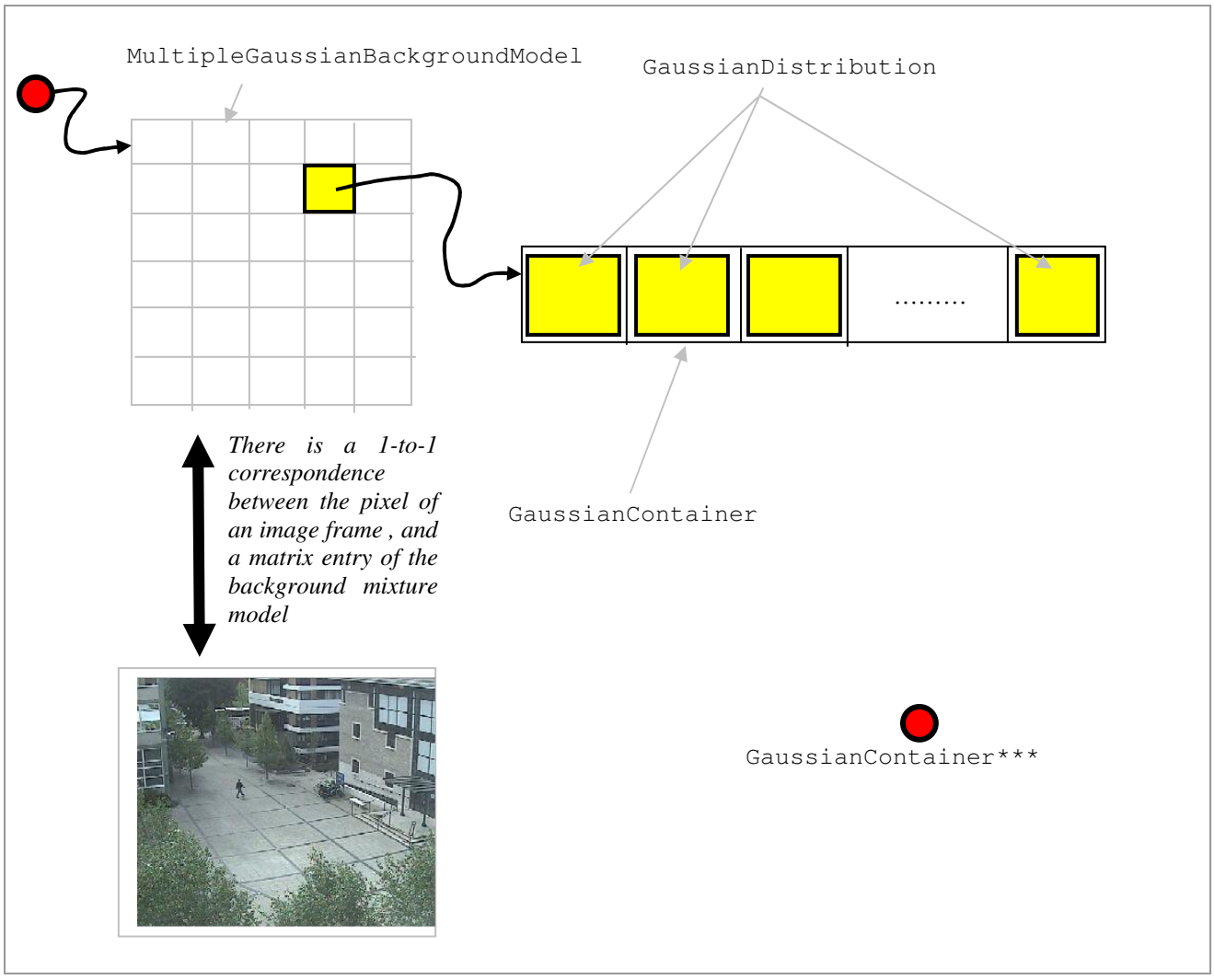


Figure 6.5: *The figure summarizes pictorially the data-structures we use to build our on-line background mixture model.*

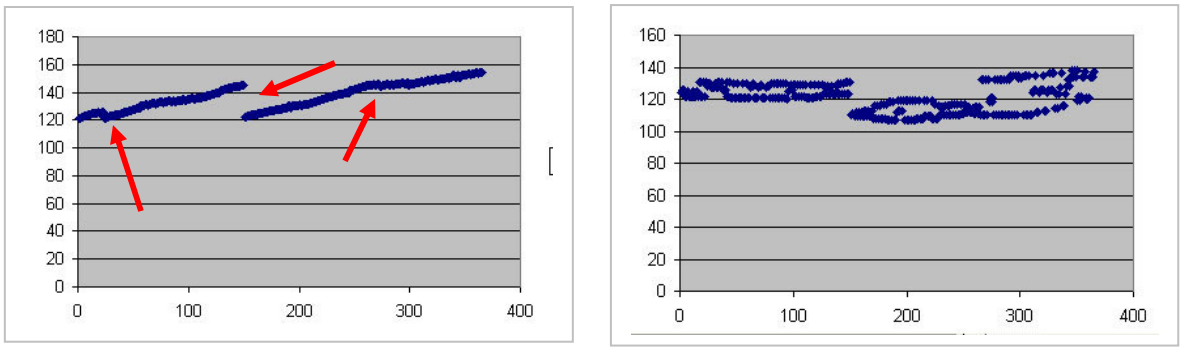
6.3 HEAD TOP FINDER IMPLEMENTATION

The algorithm for finding head-tops was discussed in Chapter 3, section 3.3. As stated earlier, head-tops are local minimums within a foreground blob. The task of finding these local minimums within a certain neighborhood is difficult owing to the manner in which OpenCV functions allow the programmer to iterate around the contour of a blob. Recollect that we iterate around the contour, since local minimums can only lie on the contour of a blob. The task of finding local minimums would have been trivial if OpenCV stored edges and allowed the programmer to iterate in an increasing or decreasing order of *y*. The most logical solution would be then to sort the edges in a particular order, before iterating. However, since the system is expected to run in real time and sorting is an expensive operation, this solution would be inappropriate.

```
1:    CvSeqReader reader;
2:    CvPoint edgeactual;
3:    cvStartReadSeq( edges, &reader);
4:    CV_READ_SEQ_ELEM( edgeactual ,reader);
5:    for(i=1; i< edges->total; i++)
6:    {
7:          CV_READ_SEQ_ELEM( edgeactual ,reader);
8:          // edgeactual contains the edge (x,y)
9:    }
```

Code 6.4: *The code required to iterate around the contour (edges) of a blob.*

Code 6.4 shows the code that we use to iterate around the edges of a blob. OpenCV stores the edges of blobs in no *precise* order. To illustrate this, figure 6.6 shows a blob which has 440 edges, and edges (with *x* and *y* coordinates separately) have been plotted in the order in which they can be iterated. The *y* values appear to be in some order, though not in a *completely* ascending order and the *x* values appear to have no sense of ordering. The absence of ordering makes it difficult to design a simple algorithm to calculate the local minimums.



(a) *(b)*

Figure 6.6: *Figure (a) The y-values may appear to be in an ascending order; however this is not the case. The red-arrows indicate the regions where there is no sense of ordering. Contours cannot be iterated along x either, since there is no ordering in x as shown in figure (b)*

Our algorithm for determining head-tops involves first determining the local minimums in a certain neighborhood, and then filtering these using the area threshold, as described in Chapter 3, section 3.3.

Input: *Edges of a blob* $E = \{e_1, e_2, \ldots\ldots\ldots\ldots, e_n\}$
Output: *Local minimum candidates* $M = \{m_1, m_2, \ldots\ldots\ldots\ldots, m_n\}$

Step 1:

Find minimum and maximum x value from all edges of the blob
minX $\leftarrow$ *minimum x from E*
maxX $\leftarrow$ *maximum x from E*
n $\leftarrow$ *(minX – maxX) / HUMAN_HEAD_SIZE*

Algorithm 6.1: *The first step of the head top finder algorithm.*

Algorithm 6.1 shows the first step of the head-top finder algorithm. We calculate the maximum number of head-tops (*n*) that could occur in the blob in a neighborhood (neighborhood is equal to the size of the human head). This can easily be calculated by first obtaining the maximum and minimum values of *x*, and then dividing their difference by the average human head size. We use 10 pixels for the human head size.

Step 2:

1: $M = \{m_1, \ldots\ldots\ldots\ldots, m_n\}$, *where* $m_k = (\infty, \infty)$ *and* $0 < k \leq n$ *(n = calculated in step 1)*
2: $Temp = \{\}$, $Temp2 = \{\}$
3: $diff \leftarrow 0$
4: ***while*** $E \neq \emptyset$
5: $E \leftarrow E - e_k$
6: $flag1 \leftarrow true$
7: $flag2 \leftarrow true$
8: $Temp \leftarrow M$
9: ***while*** $Temp \neq \emptyset \wedge flag1 = true$
10: $Temp \leftarrow Temp - t_k$
11: ***if*** $y(e_k) < y(t_k)$ *then*
12: $Temp2 \leftarrow M$
13: ***while*** $Temp2 \neq \emptyset \wedge flag2 = true$
14: $Temp2 \leftarrow Temp2 - t2_k$
15: $diff \leftarrow x(e_k) - x(t2_k)$
16: ***if*** $diff < HEAD\_SIZE \wedge y(e_k) < y(t2_k)$ *then*
17: $M \leftarrow (M - t2_k) \cup e_k$
18: $flag1 \leftarrow false$
19: $flag2 \leftarrow false$
20: ***else if*** $diff < HEAD\_SIZE \wedge y(e_k) \geq y(t2_k)$ *then*
21: $flag2 \leftarrow false;$
22: *end if*
23: *end while*
24: ***if*** $flag = true$ *then*
25: $M \leftarrow (M - t_k) \cup e_k$
26: $flag1 \leftarrow false$
27: *end if*
28: $flag2 \leftarrow true$
29: *end if*
30: *end while*
31: *end while*

M now contains the local minimums, and may contain some infinite points (∞, ∞)

Algorithm 6.2: *The second step of the head top finder algorithm. The function y(p) gives the y-coordinate of the point p (x,y). HEAD_SIZE is a program constant, and denotes the average size of a human head in pixels.*

6.4 THE TIME COMPLEXITY OF THE LOCAL-MINIMUM FINDER ALGORITHM

The algorithm works well and has produced good results in detecting the local minimum points. The time-complexity of the algorithm is dominated by the three *while* loops. Referring to algorithm 6.2, the *while* loop on line 4 has a worst-case complexity of $\boldsymbol{O}(E)$, where E is the number of edges in a blob (typically 100-150 edges for a single human blob). The *while* loops on line 9 and 13 each have a worst-case complexity of $\boldsymbol{O}(n^2)$, where n is the difference between the maximum and minimum x co-ordinates of the blob's edges, divided by the average human head size. n essentially grows with the number of people in the blob; and its average value is approximately equal to the number of people in the blob. Thus, the total time complexity of the algorithm is aggregated to $\boldsymbol{O}(En^2)$. Though not greatly efficient, modern processors make such worst-case complexity safe enough for real-time processing.

6.5 FILTERING HEAD TOPS

Not all local minimums make it to become a head top. Some are filtered out if they do not pass an area threshold test, as described in Chapter 3, section 3.4. This method effectively removes any local-minimum points within a neighborhood of pixels that represent human shadow. The `headTopFilter( )` function of the `HeadTopProcessor` class performs this filtering process. The function takes in a `vector` of local-minimums and the `IplImage` image with blobs as parameters.

The implementation is straightforward. The function first initializes a few *parallel* arrays, filling it with each local-minimum's information, such as the center of the *elliptical area* underneath, major and minor axes of the ellipse, etc. (The elliptical area underneath each local minimum has been described in detail in Chapter 3, section 3.4, figure 3.3). These arrays run in parallel, with each array's corresponding entries all representing any one particular local minimum.

```
1: for (it=headTops->begin(); it!=headTops->end(); ++it) {
2:        height_ellipse = HUMAN_HEIGHT * ((double)it->y / image_height);
3:        center = cvPoint(it->x, it->y+(double)(height_ellipse/2));
4:        headTopsArray[i] = *it;
5:        areaOccupied[i] = 0;
6:        centerOfEllipseArray[i] = center;
7:        majorAxisArray[i] = height_ellipse/2;
8:        minorAxisArray[i] = HUMAN_HEAD_SIZE/2;
9:        areaThreshold[i]=
10:       cvRound(((double)AREA_THRESHOLD_PERCENTAGE/100.0)*(PI*majorAxis
11:       Array[i]*minorAxisArray[j]));
12:       i++;
13:  }
```

Code 6.5: *The initialization step of the parallel arrays. Each array's corresponding indexes refer to the same local-minimum. A few details have been omitted from the code for brevity.*

After initialization, the `image` containing the blobs is iterated pixel-by-pixel. At each iteration step, a pixel is tested to see if it is black or white (black indicating foreground) (code 6.5 line 2). If it is a foreground pixel, it is then checked to see if it lies within any local-minimum's ellipse (the ellipse underneath each local-minimum) (code 6.5 ln 6). If it does, the local-minimum's area count is incremented by 1(code 6.5 ln 9).

```
1:  color = getPixelColor(image, cvPoint(x,y));
2:  if (isBlackPixel(color))
3:  {
4:     for (k=0;k<n;k++)
5:     {
6:          if(isInEllipse(centerOfEllipse[k],majorAxis[k],minorAxis[k],
7:          cvPoint(x,y)))
8:          {
9:               areaOccupied[k]++;
10:         }
11:    }
12: }
```

Code 6.6: *The above program steps are performed at each iteration of the pixel-by-pixel loop, which goes through each pixel (x,y). Details have been omitted, and code has been simplified for brevity.*

This way we determine each local minimum's elliptical area underneath. This area, which is stored in `areaOccupied[i]` for the i^{th} local minimum, is compared against its threshold stored in `areaThreshold[i]`. If a local minimum does not meet the threshold, it is discarded. This way unwanted local-minimums are filtered out, and what remains are actual head-top candidates.

6.6 OVERALL TIME COMPLEXITY

The time complexity of the filter implementation is dominated by the *for* loop which iterates through all the pixels of the image. Our system runs on images of size 360-by-240, which contain 86,400 pixels.

The total time complexity of the entire head-top finder implementation can now be calculated, where *a* and *b* are the height and width of the image, respectively:

$$\boldsymbol{O}\ (E n^2 + ab) = \boldsymbol{O}\,(ab)$$

Results of this implementation have been presented in Chapter 7.

6.7 IMPLEMENTATION OF THE TRACKER

The OpenCV library contains various container classes (C structs and classes) for storing graphical entities such as points, lines, curves, etc. Blobs are a central theme for any motion tracking systems. However, the container classes for blobs, namely `CvSeq` is not sufficient enough to be used in most tracking system. It provides the bare minimums for any blob, enabling only the storage of a sequence of points. The sequence of points represents the contour of a blob, which is a closed curve. A tracking system requires more functionality than that, since most computations performed by the tracker revolve around the blob.

A wrapper class for blobs, popularly attributed to D. Grossman et. al. [18], was used to extract and store blobs. The class provides indispensable functions which assist in computing useful features such centers of a blob, maximum/minimum point on a blob, perimeter and area of a blob, etc. Code 6.7 illustrates the ease with which this library can be used:

```
1:    CBlobResult blobs = CBlobResult( image, NULL, 100, true );
2:    blobs.Filter( blobs, B_INCLUDE, CBlobGetArea(), B_LESS, 600 );
3:    blobs.Filter( blobs, B_INCLUDE, CBlobGetArea(), B_GREATER, 50 );
4:    for  (int i=0; i<blobs.GetNumBlobs(); ++i)
5:    {
6:          CBlob Blob = blobs.GetBlob(i);
7:    }
```

Code 6.7: *line 1 extracts blobs from an* `IplImage` *image pointed to by the variable* `image`*. Lines 2 and 3 filters out blobs that are less than 50 and greater than 600 pixel area units. Lines 4 to 7 show how one could loop through the individual blobs.*

The Grossman wrapper seems sufficient; however, it does not wholly suffice our tracker requirements. We require more functionality. Hence we have written our own wrapper class that provides the extra functionality required. We call our wrapper class a `TrackedBlob`. Code 6.8 illustrates some of the important features of a tracked blob; we have omitted a few details for brevity.

```
1:    class TrackedBlob
2:    {
3:          HotSpot spot;
4:          CvPoint previousPosition;
5:          CvPoint currentPosition;
6:          double speed;
7:          int lostBlobCount;
8:          int red;
9:          int blue;
10:         int green;
11:         int status;
12:         vector<CvPoint> motionHistory;
13:   }
```

Code 6.8: *A wrapper class designed by us for Grossman's Blob class. The wrapper class provides some extra features that are essential for the tracker for tracking purposes.*

A `trackedBlob` stores important information about its position (lines 4 and 5 in code 6.8) and speed which are vital for predicting its motion. It also keeps track of how many frames passed since it was last observed, using the `lostBlobCount`. Usually a blob is discarded, and regarded as lost when it has not been observed for the past 4 frames. This value can be adjusted, taking external factors into consideration. On a bright sunny day, 4 frames showed good performance. However, on a windy day, when there are frequent changes in illumination, a greater frame threshold may be set to account for frequent blob losses.

A blob also stores information about its search neighborhood Ω, i.e. the hotspot. As defined earlier, this is the region within which the blob is expected to be observed in the subsequent frame (See Chapter 3, section 3.4).

Code 6.9 shows the `Hotspot` class.

```
1:    class HotSpot
2:    {
3:    public:
4:          CvPoint center;
5:          double radius;
6:          HotSpot(CvPoint p, double r)
7:          {
8:                center = p;
9:                radius = r;
10:         }
11:         HotSpot()
12:         {
13:               center = cvPoint(0,0);
14:               radius = 0;
15:         }
16:   }
```

Code 6.9: *The class Hotspot which represents the region around a blob within which the blob is expected to move in the subsequent.*

The hotspot of a blob is essentially a circle with a certain radius. Our system uses a radius value of 10 pixels. However this design can be easily used when different hotspot radii are assigned depending on the velocity observed, i.e. for example a higher radius for a higher velocity value. The constructor `HotSpot(CvPoint p, double r)` allows this feature.

The `hotspot` class provides some useful functions, as shown in code 6.10.

```
1:    bool pointInHotSpot(CvPoint point);
2:    void update(CvPoint p, double r);
```

Code 6.10: *Hotspot class functions.*

The `pointInHotSpot()` function returns true if a point lies within a hotspot. It checks to see if the point lies within the circle using the inequality $(x-a)^2+(y-b)^2 \le r$. The `update` method enables changing the hotspot's radius, which may be required if the velocity changes.

We look back to our algorithm for tracking blobs and describe the function implementations which are invoked during tracking. Recollect that when tracking a blob, we first calculate the predicted position of the blob using its velocity and direction of motion information. The next step is to determine the observed position. However, some blobs may not have an observed position (lost blobs). If there is an observed position, we perform a weighted average of the predicted and observed positions, to determine the blob's next position. If, however, there is no observed position, we simply use our predicted position as the blob's next position. A detailed discussion is given in Chapter 4, section 4.7.

The `trackedBlob` class provides the functions that are used to predict, estimate and update a blob, while it is being tracked. Code 6.11 lists the important ones.

```
1:    CvPoint getEstimatedPosition(CvPoint observedPoint);
2:    CvPoint getEstimatedPosition();
3:    void estimateTrackAndUpdate(CvPoint);
4:    void estimateTrackAndUpdate();
5:    CvPoint getPredictedPosition();
```

Code 6.11: `trackedBlob` *class functions from* `TrackedBlob.h`

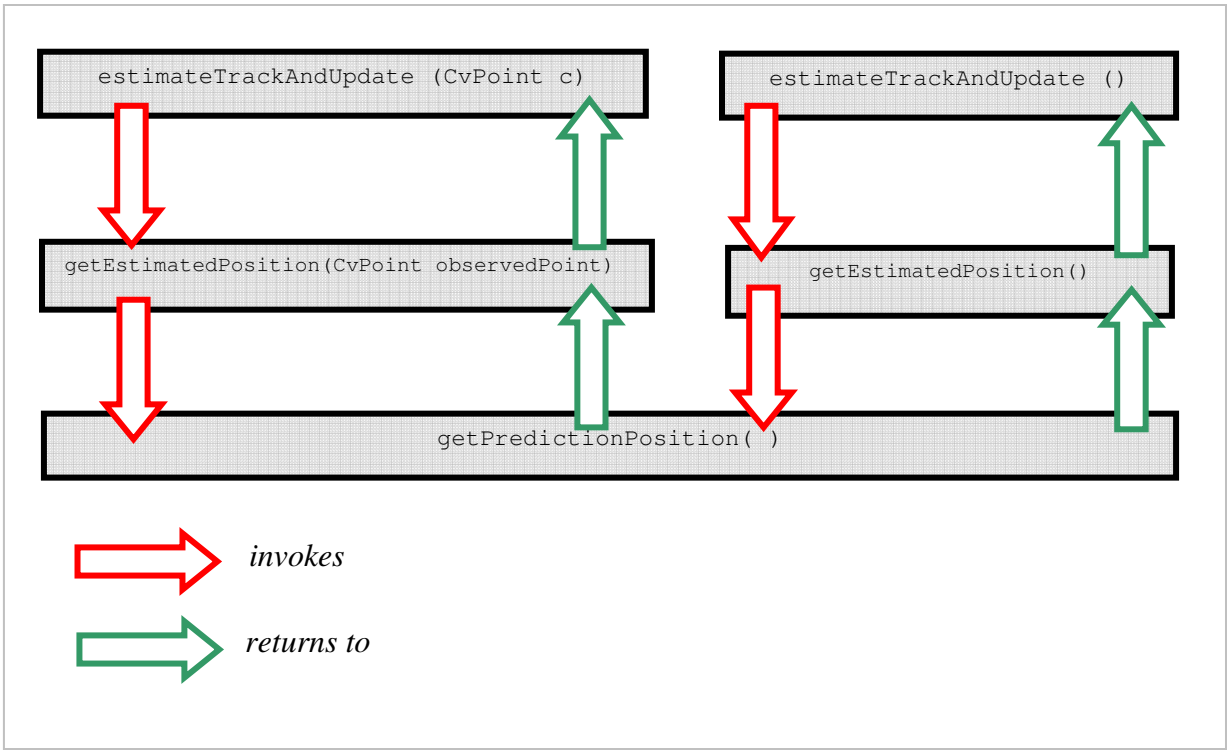


Figure 6.7: *The hierarchy of function calls during prediction, estimation and update. Note that updating is only performed after estimation and prediction.*

The most important function which requires discussion is the one at the bottom of the hierarchy: `getPredictedPosition()`. This function performs the prediction step for a blob, by calculating the distance it will travel from prior speed (line 4, Code 6.12), and then using linear motion formulas (Chapter 4, section 4.7, equations 4.3) to calculate the prediction position (lines 5-7, Code 6.12). After employing Kalman filtering, `getPredictionPosition( )` changes considerable. See code 6.21.

There are two important things to note here: if the distance traveled in the previous frame is zero, it is sensible to return the current position as the next predicted position.

```
1: CvPoint TrackedBlob::getPredictedPosition()
2: {
3:      double distanceTravelledInLastFrame =
               distance(currentPosition, previousPosition);
4:      double predictedDistanceToBeTravelled = speed * FRAME_RATE;
5:      double lambda = 1 + predictedDistanceToBeTravelled/distanceTravelledInLastFrame);
6:      double predictedXOrdinate = (1-lambda)*previousPosition.x +
                                    lambda*currentPosition.x);
7:      double predictedYOrdinate = (1-lambda)*previousPosition.y +
                                    lambda*currentPosition.y);
8:      if (distanceTravelledInLastFrame == 0)
9:      {
10:             return currentPosition;
11:     }
12:     return
        cvPoint(cvRound(predictedXOrdinate),cvRound(predictedYOrdinate));
13: }
```

Code 6.12: *The* `getPredictedPosition( )` *which performs motion prediction for a tracked blob*

After the prediction step is performed, recollect that we estimate the next position based on the observed point. If we have no observed point, we simply use the predicted position. Overloaded functions `getEstimatedPosition(CvPoint observedPoint)` and `getEstimatedPosition()` perform these estimations (see Chapter 4, section 4.7). The latter takes no parameter in order to estimate positions for blobs with no observed point.

```
1:  CvPoint TrackedBlob::getEstimatedPosition(CvPoint observedPoint)
2:  {
3:      CvPoint predictedPosition = getPredictedPosition();
4:      double estimatedPositionXOrdinate = ((1-
        (TRACKER_RELIABILITY_FACTOR*observedPoint.x);
5:      double estimatedPositionYOrdinate = ((1-
        TRACKER_RELIABILITY_FACTOR)*predictedPosition.y)
        + (TRACKER_RELIABILITY_FACTOR*observedPoint.y);
6:      return cvPoint(cvRound(estimatedPositionXOrdinate),
              cvRound(estimatedPositionYOrdinate));
7:  }

8:  CvPoint TrackedBlob::getEstimatedPosition()
9:  {
10:     return (getPredictedPosition());
11: }
```

Code 6.13: *getEstimatedPosition performs estimation*

The estimated positions are calculated using the weighted average of the predicted (acquired from the `getPredictionPosition( )` function) and the observed positions. `TRACKER_RELIABILITY_FACTOR` is a program constant, and we use a value of 50% indicating that we trust our prediction and observed positions equally. The zero-parameter `getEstimatedPosition( )` function cannot perform an estimation, since it has no observed point, and hence it uses the predicted position as the next estimated position.

We now look at the two overloaded `TrackedBlob` functions, which are at the top of the hierarchy: `estimateTrackAndUpdate (CvPoint c)` and `estimateTrackAndUpdate ()`. The latter is for blobs with no observed point. These functions invoke the predictor and the estimator functions, and perform maintenance operations such as updating blob's data members. The blob needs to be updated since the estimator returns a new estimated position, which becomes the blob's current position in the next frame.

```
1: void TrackedBlob::estimateTrackAndUpdate(CvPoint c)
2: {
3:  CvPoint centerOfBlob = kalmanFilter(c);
4:  CvPoint nextEstimatedPosition = getEstimatedPosition(centerOfBlob);
5:  // update tracked blob and prepare for next estimation
6:  speed = distance(centerOfBlob, currentPosition) / FRAME_RATE;
7:  updateTrackedBlob(nextEstimatedPosition, nextEstimatedPosition,
    currentPosition, DEFAULT_HOTSPOT_RADIUS, \
    BEING_TRACKED, speed,0);
8: }
```

Code 6.14: *The function which is at the top of the hierarchy of functions. We have included its overloaded counterpart* `estimateTrackAndUpdate ()` *for brevity.*

6.8 THE `Tracker` CLASS

The heart of the tracking system is the `tracker` class which performs the tracking process. The class definition is simple, and it uses a C++ vector to store all the blobs it is currently tracking. It only contains a single function: `track( )`, which *coordinates* the entire tracking process.

```
1:  class Tracker
2:  {
3:     vector<TrackedBlob>* trackedBlobs;
4:     public:
5:     Tracker()
6:     {
7:           trackedBlobs = new vector<TrackedBlob>;
8:     }
9:     void track(IplImage* img, vector<CvPoint>* blobCenters);
10:  };
```

Code 6.15: *The* `tracker` *class definition from* `Tracker.h`*. The destructor function has been omitted for brevity.*

The `Track( )` function performs the most important parts of the tracking process. Its task is to primarily maintain and keep track of each blob's state in its entire lifetime (See Chapter 4, section 4.8). At each frame, it updates each tracked blob's state and invokes the appropriate functions for prediction, estimation and update. The parameter to the track function is an `IplImage` to which it writes back its output: the trajectories of the human blobs. Since, we draw out the trajectories on the image sequences; we feed each image frame sequence as input to this function. The second parameter is the vector of the blob's centers, or otherwise points which it is supposed to track on the human. Head-tops can be used as an alternate. This vector is the provided to the `track( )` function by the `findHeadTops( )` function of the `HeadTopProcessor` class.

```
1: Tracker* t = new Tracker( );
2: // loop through a sequence of image
3: // loop body:
4: Vector<CvPoint>* v = ht->findHeadTops(img1);
5: t->track(img1, v);
6: cvShowImage("scene", img1);
7: // end loop
```

Code 6.16:: *Code shows how the output from the head-top finder class is redirected into the track function, for tracking purposes. Here we are tracking the head-tops of the humans.* `img1` *is each image frame in the sequence on which the trajectories are to be drawn out.*

The implementation of the track ( ) function is too verbose, and we felt that presenting the algorithm is a more effective way of explaining the `track` function. We have divided the algorithm into 3 steps, and these 3 steps are performed in a loop that iterates through the set of image sequence frames I . T is the set of blobs that are currently being tracked.

$I = \{i_o, i_1, \ldots\ldots\ldots\ldots, i_n\}$

$T = \{\ \}$

while $i \neq \emptyset$ ***do***

$I \leftarrow I - i_k$

$B \leftarrow getHeadTop(i_k)$

$T \leftarrow$ *Perform step 1 with input parameter = B*

$T \leftarrow$ *Perform step 2 with input parameter = T*

$T \leftarrow$ *Perform step 3*

Output trajectories: mark the current position of each tracked blob in T on image i_k

end while

Step 1:

Input parameter: *set of trackable points* B *and set of tracked blobs* T

Output: T'

while $B \neq \emptyset$ ***do***

$B \leftarrow B - b_i$

$Temp \leftarrow T$

while $Temp \neq \emptyset$ *do*

$Temp \leftarrow Temp - p_k$

if $b_k \in Hotspot\ (p_k)$ ***then***

if *status* $(p_k) =$ *NEW_BLOB* ***then***

measure velocity of blob, calculate direction and update

else if *status* $(p_k) =$ *BEING_TRACKED* ***then***

`estimateTrackAndUpdate ( ` b_i `)`

else if *status* $(p_k) =$ *BLOB_LOST* ***then***

`estimateTrackAndUpdate ( ` b_i `)`

end if

mark t_k *as tracked in current frame*

blobtracked $\leftarrow$ *true*

end if

end while

if *blobtracked = false* ***then***

create a new tracked blob t *from* b_i

$T \leftarrow T \cup t$

end if

end while

Algorithm 6.3: *Step 1 of the* `track()` *function algorithm*

Step 2: *Maintain and update tracked blobs that were not tracked in the current sequence frame*

Input parameter: *set of tracked blobs T*

Output: T'

Temp2 $\leftarrow$ *T*

while *Temp2* $\neq \emptyset$ ***do***

    Temp2 $\leftarrow$ *Temp2* - $t2_k$

    if $t2_k$ *was tracked in step 1* ***then***

        if *status* ($t2_k$) = *BLOB_TRACKED* ***then***

            `estimateTrackAndUpdate ( )`

        else if *status* ($t2_k$) = *BLOB_LOST* ***then***

            if *tracked blob* $t2_k$ *has reached MAX_LOST_THRESHOLD* ***then***

                status ($t2_k$) $\leftarrow$ *DISCARD_BLOB*

            else

                `estimateTrackAndUpdate ( )`

            end if

        else if *status* ($t2_k$) = *NEW_BLOB* ***then***

            status ($t2_k$) $\leftarrow$ *DISCARD_BLOB*

        end if

    end if

end while

Algorithm 6.4: *Step 2 of the* `track()` *function algorithm*

Step 3: *Remove tracked blobs that have their statuses set to DISCARD_BLOB*

Input parameter: *set of tracked blobs T*

Output: T'

$Temp3 \leftarrow T$

while $Temp3 \neq \emptyset$ ***do***

    $Temp3 \leftarrow Temp3 - t3_k$

    if *status* $(t3_k)$ = *DISCARD_BLOB* ***then***

        $T \leftarrow T - t_k$

    else

        $lostBlobCount\ (t_k) \leftarrow lostBlobCount\ (t_k) - 1$

    end if

end while

Algorithm 6.5: *Step 3 of the* `track()` *function algorithm*

6.9 IMPLEMENTING THE KALMAN FILTER FOR MOTION PREDICTION

OpenCV provides some very useful functions for implementing the Kalman filter. These functions are part of its Motion Analysis package. We associate a separate Kalman filter to each blob that is being tracked. Apart from the data members that we have listed already in code 6.8., a `TrackedBlob` contains more members that are required to store the Kalman matrices and its parameters. Code 6.17 lists them all.

```
1:    class TrackedBlob
2:    {
3:          // kalman parameters
4:          CvKalman* kalman;
5:          CvMat* measurement;

6:          // more trackedblob data members
7:    };
```

Code 6.17: *The* `TrackedBlob` *class stores the Kalman filter parameters enabling each tracked blob object to be associated to a Kalman filter, for its motion's prediction.*

It is most sensible to initialize the Kalman parameters in the constructor of `TrackedBlob`. Code 6.18 shows how we initialize these parameters.

```
1: kalman = cvCreateKalman(4,2,0);
2: cvSetIdentity(kalman->measurement_matrix, cvRealScalar(1));
3: cvSetIdentity(kalman->process_noise_cov, cvRealScalar(proc_cov));
4: cvSetIdentity(kalman->measurement_noise_cov, cvRealScalar(meas_cov)) ;
5: cvSetIdentity(kalman->error_cov_post, cvRealScalar(1));
6: memcpy( kalman->transition_matrix->data.fl, A, sizeof(A));
7: CvRNG rng = cvRNG(-1);
8: cvRandArr(&rng, kalman->state_post, CV_RAND_NORMAL, cvRealScalar(0),
            cvRealScalar(0.1));
9: kalman->state_post->data.fl[0]=centerOfBlob.x;
10: kalman->state_post->data.fl[1]=centerOfBlob.y;
11:measurement=cvCreateMat(2,1,CV_32FC1);
```

Code 6.18: *Kalman filter initialization in* `TrackedBlob` *constructor. The code has been adopted from OpenCV's Kalman filter sample example, which ships with OpenCV.*

The initialization code contains a few constants, which we have defined in the `constants.h` file. These are the values for the process and measurement covariance. We also define a transition matrix *A* using `const float A [ ]`. This matrix relates how the states interact (Chapter 5, equation 5.12). Code 6.19 shows these declarations.

```
1:    const double proc_cov=1e-5;
2:    const double meas_cov=1e-5;
3:    const float A[] = {1,1,0,1};
```

Code 6.19: *Some constants required for Kalman initialization*

In the initialization (code 6.18), we initialize a Kalman OpenCV object (line 1) using `cvCreateKalman`. The first two parameters are the dimensionality of the state and measurement vectors. See section 5.5, for a discussion of our state and measurement vectors. The next few lines 2-5, initialize the Kalman filter's internal matrices. What we are essentially doing, is multiplying the internal identity matrices with some of our defined constants, such as the ones we have declared in code 6.19. Lines 9-10 initialize the state vector to contain our initial tracking point (center of blob, or head top).

The rest of Kalman filter prediction is performed using the `kalmanFilter(CvPoint)` function of `TrackedBlob` class.

```
1: CvPoint TrackedBlob::kalmanFilter(CvPoint m) {
2:    measurement->data.fl[0] = m.x;
3:    measurement->data.fl[1] = m.y;
4:    cvKalmanPredict(kalman, 0);
5:    cvKalmanCorrect(kalman, measurement);
6:    return cvPoint(cvRound(kalman->state_post->data.fl[0]),
                  cvRound(kalman->state_post->data.fl[1]));
7: }
```

Code 6.20: *The* `kalmanFilter( )` *invokes OpenCV's prediction and correction Kalman functions.*

The `kalmanFilter( )` function performs subsequent predictions, after the initialization process. Subsequent calls to the `kalmanFilter( )` function with the measurement point, triggers automatic prediction and correction performed by OpenCV's `cvKalmanPredict` and `cvKalmanCorrrect` functions. Internal matrices are updated using the new measurement point (line 2-3 code 6.20). The process and measurement covariances are updated automatically internally by making calls to `cvKalmanCorrect` (line 5, code 6.20).

It is important to note that the Kalman filtering process is an alternative to predicting motion using vector analysis. However, for lost blobs, we use our vector analysis techniques to track their motion. We re-visit our function hierarchy in section 6.2, figure 6.4. `getPredictionPosition( )` can now be modified and greatly simplified. Code 6.21 illustrates this.

```
1: CvPoint TrackedBlob::getPredictedPosition()
2: {
3:    return kalmanFilter(currentPosition);
4: }
```

Code 6.21: *the new* `getPredictedPosition( )` *function, and greatly simplified since it is not using vector analysis methods to predict motion anymore. It uses the prediction of the Kalman filter. Refer to code 6.12 where have we introduced the* `getPredictedPosition( )` *function.*

`getEstimatedPosition( ),` which is for lost blobs, still needs to predict using vector analysis techniques for linear motion. We have found that the Kalman filter behaves poorly for lost blobs, for which there is really no measurement value. For a measurement value we predict the lost blob's motion using vector analysis, and then use this to predict using Kalman filter. The code for `getEstimatedPosition( )` now does motion prediction first, and then uses its results for the Kalman filter.

```
1:  CvPoint TrackedBlob::getEstimatedPosition()
2:  {

3:    // calculate the predicted position using position vectors
4:    // velocity of blob
5:    double distanceTravelledInLastFrame =
                distance(currentPosition, previousPosition);
6:    double predictedDistanceToBeTravelled = speed *
                FRAME_RATE;

7:    double lambda = 1 +
(predictedDistanceToBeTravelled/distanceTravelledInLastFrame);

8:    double predictedXOrdinate = (1-lambda)*previousPosition.x
                                + (lambda*currentPosition.x);
9:    double predictedYOrdinate = (1-lambda)*previousPosition.y
                                + (lambda*currentPosition.y);

10:   if (distanceTravelledInLastFrame == 0)
11:   {
12:        return currentPosition;
13:   }
14:   CvPoint c =  cvPoint(cvRound(predictedXOrdinate),
                     cvRound(predictedYOrdinate));

15:   return kalmanFilter(c);
16:  }
```

Code 6.22: `getEstimatedPosition( )` *remains with little changes. Note that in code 6.15* `getEstimatePosition()` *invokes* `getPredictedPosition( )`*. Since* `getPredictionPosition( )` *has simplified after employing the Kalman filter, we insert the code of the previous* `getPredictedPosition( )` *into here, however, passing the final returned point through the Kalman filter.*

Results from Kalman filtering have been presented on Chapter 7

Chapter 7

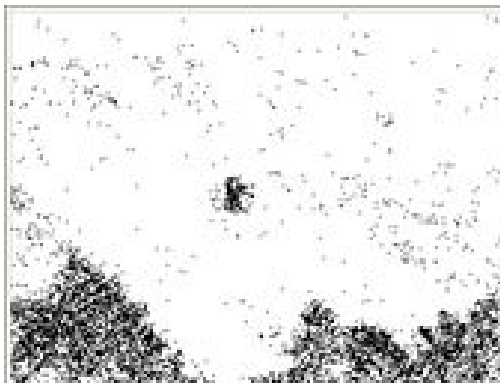
Results

In this chapter we present our results and findings, with real outputs from our system. We have divided this chapter into four sections namely: background differencing, head-top finder, appearance modeling and motion tracker. These four sections have been named according to the four most integral components of our system. We test these components by diverting their output to the screen. The components don't necessarily produce outputs to the screen when run with the tracker. See fig 6.0. for a pictorial description of the entire system, and its data flow.

7.1 BACKGROUND DIFFERENCING RESULTS

Background set 1:

After 20 frames

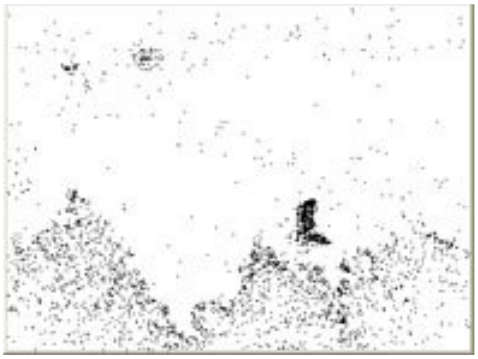


After 100 frames

Result set 7.1*: The images on the right are the background subtracted images. The leaves of trees are moving constantly due to wind, and notice how the background model has learnt and adapted to the moving leaves in the 100th frame.*

Background set 1 (continued):

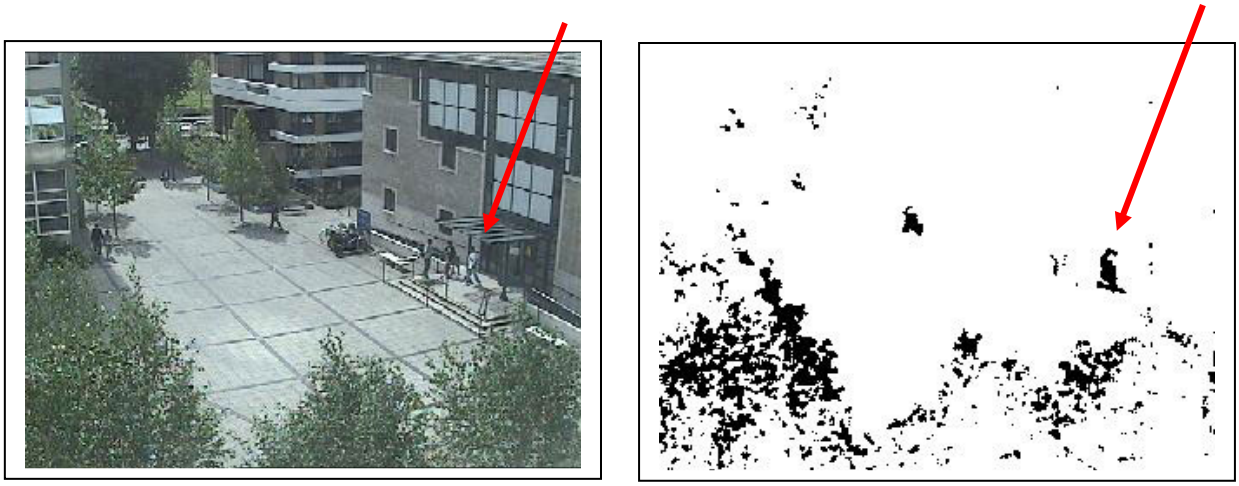
After 300 frames



After 600 frames

Result set 7.2: *There is a stark change in illumination in the courtyard between the 100th and the 300th frame. The background model has adapted very well to this change. After 600 frames have passed, the system has completely learnt about the motion of leaves of trees, and has subtracted them quite well.*

Background set 2:



Notice that a person comes out of the building (pointed to by the arrow), he is still not part of the background, as can be seen in the background-differenced image. We will be focusing on him for the rest of this example.

After 20 frames

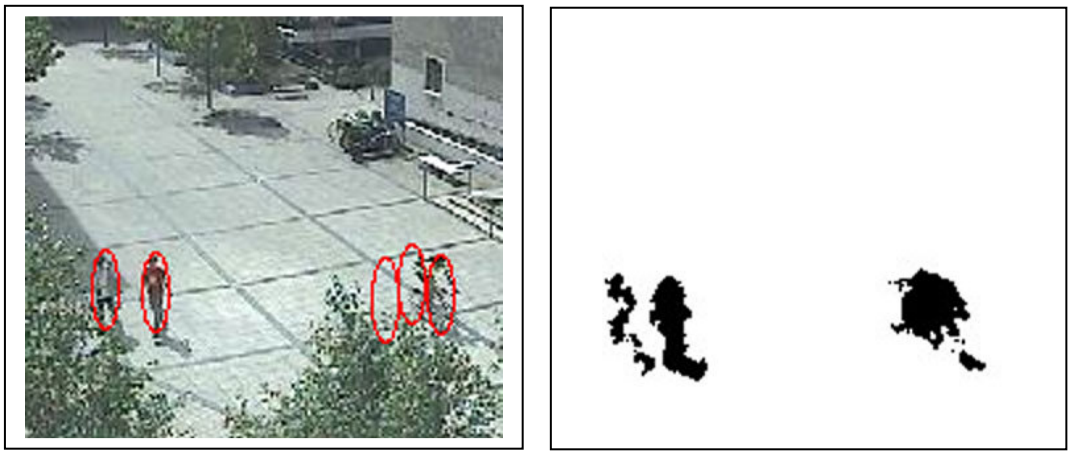
After 20 frames have passed, the person can be seen standing with the other people who apparently have become part of the background. The person, who we have been focusing on, is slowly becoming part of the background. Notice how the human subject's blob is disappearing.

After 40 frames

After another 20 frames have passed, the blob has disappeared significantly, indicating that the human subject has become a part of the background. This shows that our system's background can incorporate objects which are added to the background, and is adaptable.

7.2 HEAD-TOP FINDER RESULTS

Result set 7.4: *A well-formed human blob is detected as expected. Shadows are completely removed.*



Result set 7.5: *The man in the bicycle causes a large foreground blob. The system finds it difficult to distinguish this blob from a similar blob that would have been produced if there were a group of people instead.*

Result set 7.6: *Perfect segmentation and detection at a point far away from the camera. Notice how the shape of the ellipse has decreased due to the perspective effect. The scaling factor for the perspective works well.*

Result set 7.7: *Single human blobs are detected correctly. Shadows are again removed perfectly.*

Result set 7.8: *A relatively deformed blob is also detected properly. The shadow can clearly be seen to contain a local minimum, as marked by the arrow. The algorithm has removed it successfully.*

Result set 7.9: *The dangers of premature human segmentation and detection. The detection was performed during the background training period. To avoid such situations, detection is started at least after the system has been trained on 500 frames for the background.*

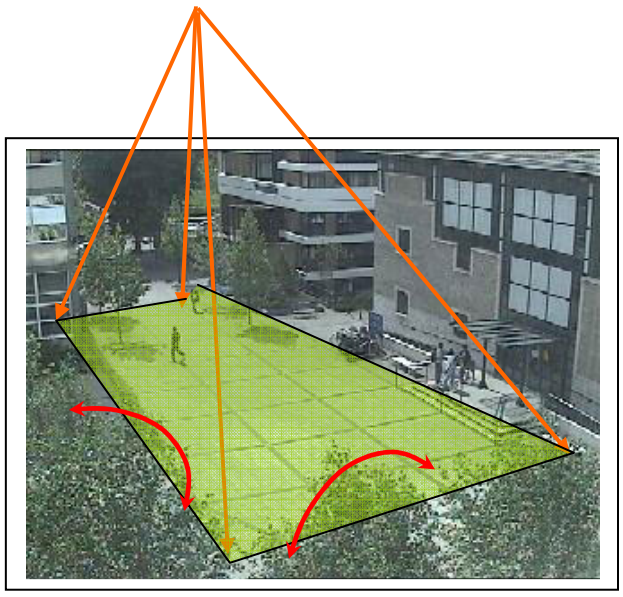
7.3 MOTION TRACKING RESULTS

Results from motion tracking have been divided into different scenes. Some frames have been cropped to focus on the regions of interest. The trajectories appear in different colors. The system assigns a random color to every human it tracks.

The scenes have been shot at Queen Mary College's courtyard, situated right in front of the Computer Science department. The camera was mounted on top of a building. Scenes were shot at different times of the day, and we have tried capturing scenes with differing illumination. Some scenes were shot on a windy day, to test how the system performs on leaves moving in the background.

Each scene has labeled frames indicating the sequence. We have also indicated, below each frame, how many frames have elapsed since the first frame. In the following diagram, we point important aspects of our system's capabilities.

c



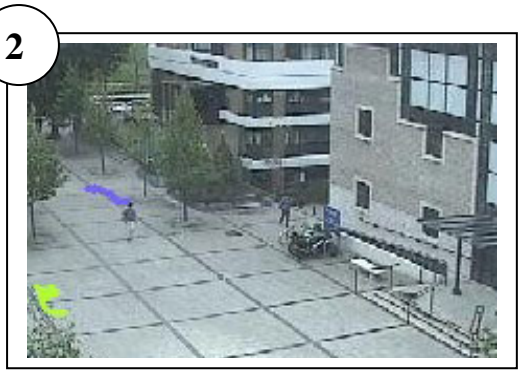
The above figure shows regions, marked by the orange arrows and light green on the ground plane, where our system had the most amounts of success rates while tracking. These are the regions where relatively bigger blobs were produced, and hence making it easier for tracking. The red-arrows indicate that the leaves of the trees are always in constant to-and-from motion

SCENE 1:

Scene shot on a relatively less windy day with poor illumination. The second human subject, who enters the scene in 4, is tracked eventually when h/she becomes visible to the tracker. Frame 5 shows the path traced for the second human subject. The colors of the trajectories happen to be very similar; however they are different when examined more closely, indicating that the system identifies them as different human subjects.

1

2



After 20 frames

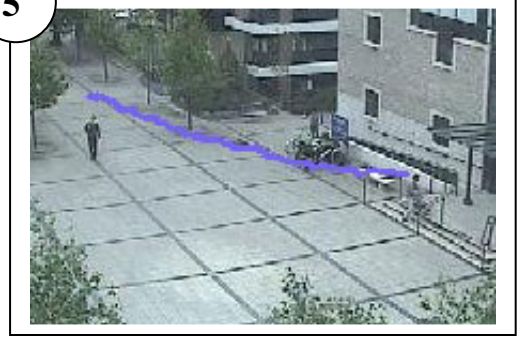
3

After 51 frames

4

After 76 frames

5



After 112 frames

6

After 154 frames

SCENE 2:

This scene was shot on a bright sunny day with ample lighting. However, tracking occurs with some success. Disruption occurs when the cyclist occludes the human group. The system loses track, but tracks the cyclist perfectly. The group is again tracked in frame 6 (light yellow trajectory). See discussion on the following page.

1

2

After 40 frames

3

After 60 frames

4

After 80 frames

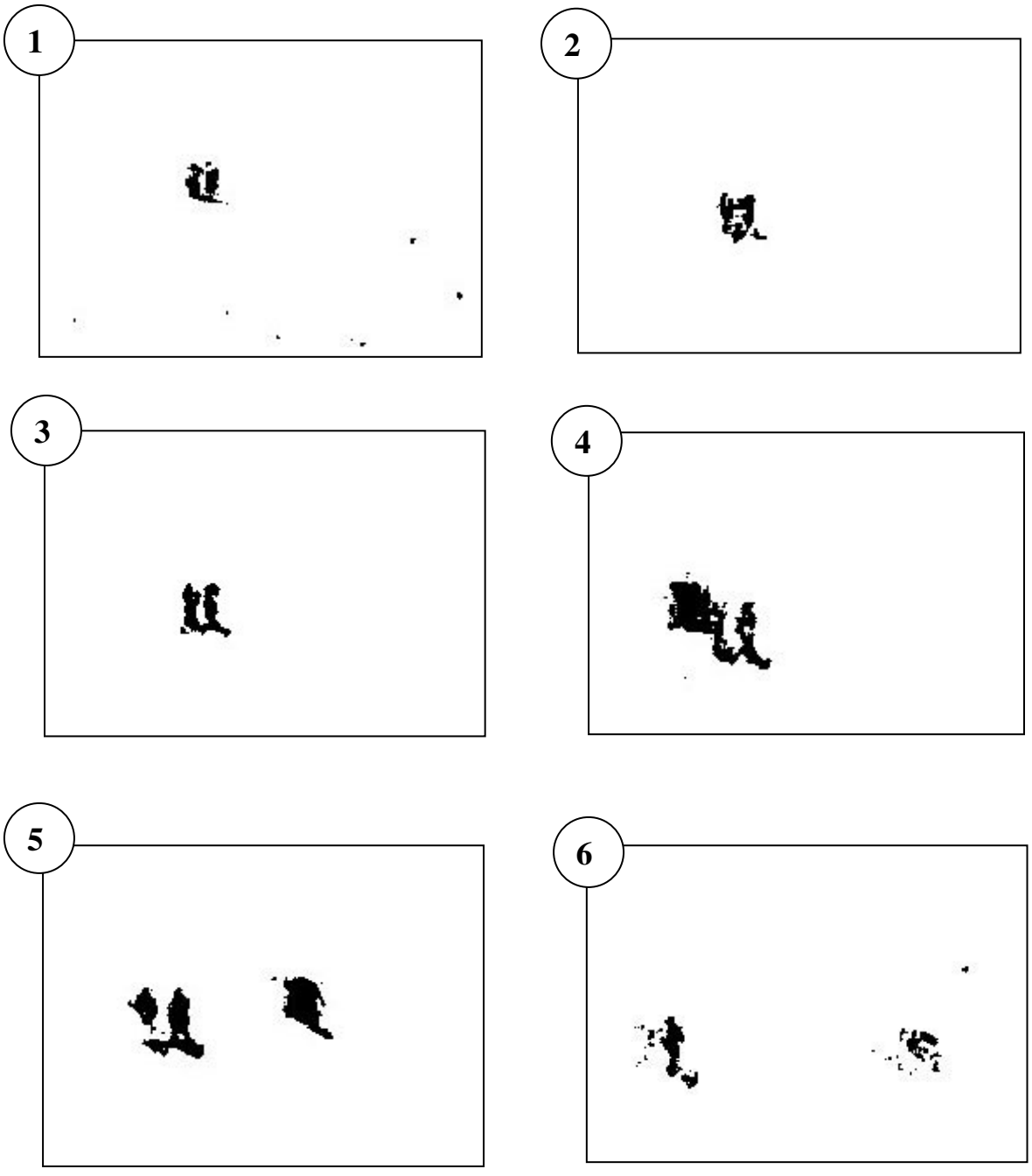
5

After 95 frames

6

After 110 frames

We analyze why scene 2 failed to track properly by examining the blobs. Notice the severity of deformations that have occurred in frames 1 and 2. However, our head-top finder algorithm would have no problems detecting two heads in frame 2. Since our default hotspot radius spans a distance of 10 pixels, it spanned the entire group of the 2 people. Frame 4 causes a serious occlusion which lasts for about 10 frames. Notice how the head-tops could have gotten shifted considerably in frame 4. Frame 6 has the one of the humans in the group completely *invisible*.



SCENE 3:

In this we analyze another situation where the scene is nicely lit; however, there is some disruption in tracking. See discussion on the following page

1

2

After 15 frames

3

After 30 frames

4

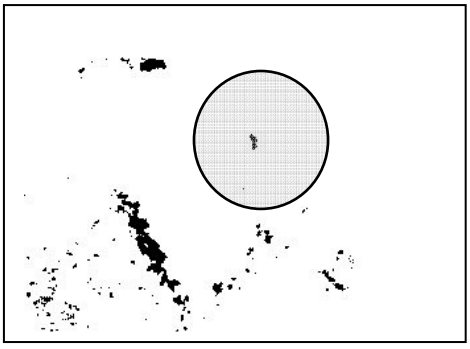
After 50 frames

5

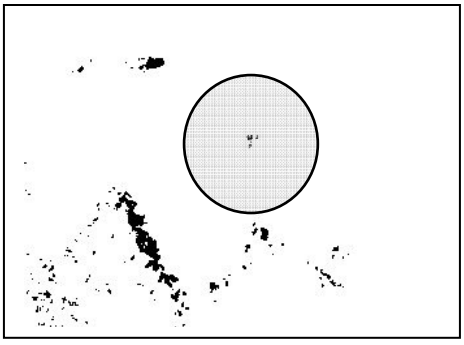
After 55 frames

Analyzing the scene puts forth two concerns that need to be examined: the human blob in frame 1 is detected late and the blob is lost and re-detected as a new blob in frame 4.

The late detection in frame 1 can easily be reasoned if we look the blobs that were produced 20-100 frames before frame 1. We have circled the places where the blob appears. Notice how the blob is very ill-formed and how its size makes it impossible for it to pass our size filter test.

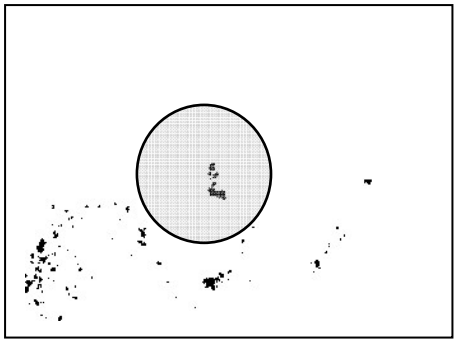


100 frames before



70 frames before

40 frames before

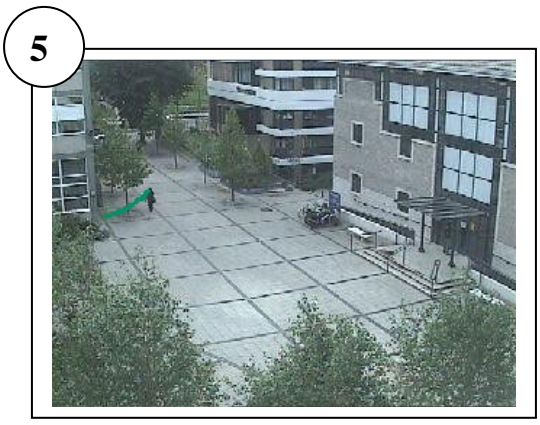


20 frames before

Frame 4, in scene 3, showed a lost blob and a re-detection as a new blob. After debugging, we have found that the blob was lost immediately right after it was detected at the *same* position. This caused the current and previous positions to be the same, hence causing the velocity to become zero. This is the reason why the system doesn't predict any further (no predict trail line which is usually characterized by a straight line for linear motion). The system waits thinking that the human has stalled. However, this is not the case. The human is later detected in frame 5, but as a different human blob. Such anomalies can occur, and are very difficult to deal with.

SCENE 4:

Scene 4 is a rather badly illuminated scene. Notice that the trees cast no shadows indicating that the sky is overcastted. The reason we have extracted this scene is to point to the reader that the system is able to track humans at all regions on the ground plane, and just not the center (which the previous scenes focused on). However, we have noticed a higher number of false trajectories for objects far away from the camera.



SCENE 5:

What we have here is a brightly illuminated scene. The system performs tracking to perfection, tracking every move of the human subject.

1

2

20 frames before

3

40 frames before

4

70 frames before

5

95 frames before

6

120 frames before

SCENE 6:

This scene shows how the tracking system performs prediction when a blob is lost. Notice how the trajectory produced in scene 5 ends in a straight line. It is impossible for humans to produce such a trajectory, and even if the move was articulated in that manner the noisy nature of the blob producing such a trajectory is highly unlikely. We can conclude that it is a straight line since the motion tracker uses its vector analysis functions to predict *linear* motion. We have also attached a background-differenced frame, produced three frames before frame no.5. Notice how the blob has completely disappeared, forcing the system to perform motion prediction. *(follow circled regions)*

SCENE 7:

We have captured various motion tracks from various scenes. Each image represents a different scene. Notice that in some scenes, there are fast moving cars in the background, and these are characterized by the straight line trajectories.

Appendix A

References

[1] T. Zhao, R. Nevatia, "Tracking multiple humans in crowded environment" *Proc IEEE transactions on pattern analysis and machine intelligence, 2004*

[2] T. Zhao, R. Nevatia, "Tracking multiple human motions in complex situations" *Proc IEEE transactions on pattern analysis and machine intelligence*, 2004.

[3] M. Han, W. Xu, H. Tao, Y. Gong, "An algorithm for multiple object trajectory tracking", *Proc. Inernatioal Conference on Computer Vision,* 2004

[4] C. R. Allen, A. Azarbayejani, T. Darrell, A.P. Pentland, "Pfinder: Real-time tracking tracking of the human body", vol. 19, no. 7, July 1997

[5] I. Cohen, G. Medioni, "Detecting and tracking moving objects for video surveillance", *Proc. of the IEEE Computer Vision and Pattern Recognition 99*, June 1999.

[6] T. Zhao, R. Nevatia, "Segmenting and tracking of multiple humans in complex situations", *Proc. of IEEE Computer Vision and Pattern Recognition*, 2001.

[7] C. Stauffer, W. Eric, L. Grimson, "Learning patterns of activity using real-time tracking", *Proc IEEE trasnsactions on pattern analysis and machine intelligence,* vol 22, no. 8, August 2000

[8] M. Yamada, K. Ebihara, J. Ohya, "A new robust real-time method for extracting human silhouettes", *Proc. of International conference on artificial intelligence, 1998*

[9] I. Haritaoglu, D. Harwood, L.S. Davis, "W4: Real-time surveillance of people and their activities", *Proc. of IEEE transactions on pattern analysis and machine intelligence*, 2000.

[10] O. Javed, M. Shah, "Tracking and object classification for automated surveillance", *Proc. of the European conference on computer vision,* 2002.

[11] T. Zhao, "Model-based Segmentation and Tracking of Multiple Humans in Complex Situations", PhD thesis, University of southern California, December 2003.

[12] G. Welch, G. Bishop, "An introduction to Kalman filter", available on the web at *http://www.cs.unc.edu/~welch/kalman/kalmanIntro.html*

[13] R. Kalman, "A new approach to Linear Filtering and Prediction problems", *Journals of Basic engineering*, Vol. 82, pp 35-45. 1960

[14] Sorenson, H.W., "Least square estimation: from Gauss to Kalman", *IEEE spectrum, vol. 7, pp 63-68,* July 1970

[15] C. Fermuller's lectures in motion tracking using Kalman filtering at the University of Maryland at college park, available on the web at *http:// www.cfar.umd.edu/~fer/cmsc426/lectures/tracking.pdf*

[16] D. Forsyth, J. Ponce, "Computer vision – A modern approach", Prentice Hall 2003, pp 374-376.

[17] Intel's opensource OpenCV library available on the web at *http://www.intel.com/research/mrl/research/opencv/*

[18] D. Grossman wrapper blob code is available in the *"files"* section of the OpenCV forum available online on *http://groups.yahoo.com*.

Appendix B

Program code

Code acknowledgements

Our system uses a blob extraction and analysis facility package developed by D. Grossman, and popularly used by machine vision researchers, all over the world. The blob analysis and extraction package has not been included in this section. It can be viewed online on the Yahoo® OpenCV forums at http://groups.yahoo.com/group/OpenCV/ in the files section, compressed in a zipped file called `cvblobslib_OpenCV_english.zip`

The package has been included on the CD, under a folder called "Blob analysis package".

The blob analysis package provides built-in functions to extract blobs and its properties.

Constants.h

```
#include <cv.h>
#include <highgui.h>
#include <vector>



#define NUMBER_OF_GAUSSIANS_IN_MIXTURE_MODEL 4
#define MATCH_FACTOR 2.5
#define LEARNING_RATE_WEIGHT 0.4
#define LEARNING_RATE_VARIANCE 0.2
#define HIGH_VARIANCE 10.0
#define LOW_WEIGHT 0.1

#define TRACK_PATH_SIZE 3.0
#define HUMAN_HEAD_SIZE 10          // always use even numbers
#define HUMAN_HEIGHT 50
#define AREA_THRESHOLD_PERCENTAGE 50

// for tracker
#define NEW_BLOB 1
#define BEING_TRACKED 2
#define BLOB_LOST 3
#define DISCARD_BLOB 4
#define FRAME_RATE 0.1
#define DEFAULT_HOTSPOT_RADIUS 10
#define TRACKER_RELIABILITY_FACTOR 0.5    // value between 0 and 1.
                                          // a value of 1 makes tracker
track
                                          // relying only on observed
                                    point (if any)
                                          // a value of 0 makes tracker
track relying
                                          // only on predicted point

#define MAX_LOST_BLOB_COUNT 10
using namespace std;


double const PI = 3.14159;


class GlobalFunctions {

public:

      // returns the greyscale value of a 1-channel image
      static double getPixelColor(IplImage* image, CvPoint point)
      {
            double grey_value;
            grey_value = ((uchar*)(image->imageData +
            image->widthStep*point.y))[point.x];
            return grey_value;
      }
```

```
	// to check if a point lies within a specified ellipse
	static bool isInEllipse(CvPoint center, double major, double
minor, CvPoint
				point)
	{
		// l = (x-h)^2 / a^2
		// r = (y-k)^2 / b^2

		double l,r;

		l = ((point.x - center.x)*(point.x - center.x)) /
(major*major);
		r = ((point.y - center.y)*(point.y - center.y)) /
(minor*minor);

		return ((l+r)<=1);
	}


	// draws straight lines between sets of points, with a specified
color
	static void drawPath(IplImage* img, vector<CvPoint> motionHistory,
CvScalar
			pathColor)
	{
		if (motionHistory.size() == 1)
		{
			return;
		}
		else
		{
			for (int i=1;i<motionHistory.size();i++)
			{
				cvLine(img, motionHistory.at(i),
motionHistory.at(i-1),
				CV_RGB(pathColor.val[0], pathColor.val[1],
			pathColor.val[2]),
					2, 8);
			}
		}
	}


};
```

Background model

GaussianDistribution.h

```
#include "../constants.h"


class GaussianDistribution
{
	double variance;
	double mean;
	double weight;

public:

	GaussianDistribution()
	{
		variance = 0;
		mean = 0;
		weight = 0;
	}

	GaussianDistribution(double m, double v, double w)
	{

		variance = v;
		mean = m;
		weight = w;
	}

	GaussianDistribution(double m)
	{
		mean = m;
		variance = HIGH_VARIANCE;
		weight = LOW_WEIGHT;
	}

	bool match(double);
	double getWeight();
	void updateDistribution(double, double);
	void adjustWeight(double, bool);
	double getWeightVarianceRatio();
	void setDistribution(double, double, double);

};
```

GaussianDistribution.cpp

```
#include "stdafx.h"
#include "GaussianDistribution.h"
#include "math.h"


bool GaussianDistribution::match(double pixelValue)
{
	double difference;
	difference = abs(pixelValue - mean);
	return (difference < MATCH_FACTOR * (sqrt(variance)));
}

double GaussianDistribution::getWeight()
{
	return weight;
}

// updates only if the pixelValue matches the distribution
// else does not update
void GaussianDistribution::updateDistribution(double pixelValue, double
learningRate)
{
	mean = ((1-learningRate)*mean) + (learningRate*pixelValue);
	variance = ((1-learningRate)*variance)+((pixelValue-
mean)*(pixelValue-mean));
}

void GaussianDistribution::adjustWeight(double learningRate, bool
matched)
{
	if (matched)
	{
		weight = (1-learningRate)*weight + learningRate;
	}
	else
	{
		weight = (1-learningRate)*weight;
	}
}

double GaussianDistribution::getWeightVarianceRatio()
{
	return (weight/variance);
}

void GaussianDistribution::setDistribution(double m, double v, double w)
{
	mean = m;
	weight = w;
	variance = v;
}
```

GaussianContainer.h

```
#include "GaussianDistribution.h"


class GaussianContainer
{
	// array of gaussian distribution (gaussian mixtures)
	GaussianDistribution** distPtr;
	int size;

public:
	// creates a container with zero gaussians
	// the number of gaussians is specified by size s
	// literature suggests size of 3-4
	GaussianContainer(int s)
	{
		// if (s<1) throw an exception

		distPtr = new GaussianDistribution*[s];
		for (int i=0;i<s;i++)
		{
			distPtr[i] = new GaussianDistribution();
		}
		size = s;
	}

	bool updateGaussians(double);
	int getSize();
	int getLeastProbableDistribution();
	void replaceDistribution(int, double, double, double);

	// destructor
	~GaussianContainer()
	{
		delete distPtr;
	}

};
```

GaussianContainer.cpp

```
#include "stdafx.h"
#include "gaussianContainer.h"

// Updates gaussians by incorporating the incoming new pixel value Xt
// From Stauffer et. al.
// returns true if a match is found and false otherwise
bool GaussianContainer::updateGaussians(double Xt)
{
	// iterate through all the gaussians in the container
	// and find the first match
	bool matchFound = false;

	// stores the index of the matched gaussian if match is found
	int index=0;

	for (int i=0;i<size && !matchFound ;i++)
	{
		if ((distPtr[i])->match(Xt))
		{
			matchFound = true;
			index=i;
		}
	}

	// distPtr[i] is the distribution which matches Xt
	if (matchFound)
	{
		// update mean and variance of the matched distribution
		(distPtr[index])->updateDistribution(Xt,
LEARNING_RATE_VARIANCE);

	}
	else
	{
		// get distribution with least alpha/sdev value
		// replace that distribution with mean = Xt, and high
variance
		int indexOfLeastProbable = getLeastProbableDistribution();
		replaceDistribution(indexOfLeastProbable, Xt, HIGH_VARIANCE,
LOW_WEIGHT);

	}

	// adjust weights
	for (int j=0;j<size;j++)
	{
		if (j==index && matchFound == true)
		{
			(distPtr[j])->adjustWeight(LEARNING_RATE_WEIGHT,
true);
		}
		else
		{
			(distPtr[j])->adjustWeight(LEARNING_RATE_WEIGHT,
false);
		}
	}

	return matchFound;

}
```

```
int GaussianContainer::getSize()
{
	return size;
}

int GaussianContainer::getLeastProbableDistribution()
{
	double temp, value;
	int tempIndex;

	value=distPtr[0]->getWeightVarianceRatio();
	tempIndex=0;

	for (int i=1;i<size;i++)
	{
		temp = distPtr[i]->getWeightVarianceRatio();
		if (temp < value)
		{
			value = temp;
			tempIndex=i;
		}
	}

	return tempIndex;


}

void GaussianContainer::replaceDistribution(int index, double mean,
double variance, double weight)
{

	distPtr[index]->setDistribution(mean, variance, weight);
}
```

MultipleGaussianBackgroundModel.h

```
#include "gaussianContainer.h"
#include "cv.h"

class MultipleGaussianBackgroundModel
{
	GaussianContainer*** containerPtr;
	int height;
	int width;

public:

	MultipleGaussianBackgroundModel(IplImage* img)
	{
		height = img->height;
		width = img->width;
		createEmptyModel(height, width,
NUMBER_OF_GAUSSIANS_IN_MIXTURE_MODEL);
	}

	double colorToIntensity(int, int, int);
	void updateModel(IplImage*, IplImage*);


	~MultipleGaussianBackgroundModel()
	{
		for (int i=0;i<height;i++)
		{
			for (int j=0;j<width;j++)
			{
				delete ((containerPtr[i])[j]);
			}
			delete (containerPtr[i]);
		}
		delete containerPtr;
	}


private:

	// helper to constructor
	// creates an empty background model
	// creates a 2D array (dimensions = height (h), width (w) )
	// each array element points to a GaussianContainer of size
specified by
	// sizeOfContainer
	void createEmptyModel(int w, int h, int sizeOfContainer)
	{

		// if sizeOfContainer < 0 then throw exception
		containerPtr = new GaussianContainer**[w];

		for (int i=0;i<w;i++)
		{
			containerPtr[i] = new GaussianContainer*[h];
			for (int j=0;j<h;j++)
			{
				// create a GaussianContainer object.
				// constructor for GaussianContainer calls
constructor
				// of GaussianDistribution
				(containerPtr[i])[j] = new
GaussianContainer(sizeOfContainer);
```

```
                        }
                    }
                }
            };
```

MultipleGaussianBackgroundModel.cpp

```
#include "stdafx.h"
#include "multipleGaussianBackgroundModel.h"

// img2 pointer to an empty image
// img2 after function finishes execution - binary image, white pixels
representing foreground
void MultipleGaussianBackgroundModel::updateModel(IplImage* img,
IplImage* img2)
{
      int step, width2;
      double intensity;
      bool isBackgroundPixel = false;

      step = img->widthStep;
      width2 = img->width * img->nChannels;
      unsigned char *data= reinterpret_cast<unsigned char *>(img-
>imageData);
      unsigned char *data2= reinterpret_cast<unsigned char *>(img2-
>imageData);

      // iterate through each pixel
      for (int i=0;i<height;i++)
      {
            for (int j=0;j<width2;j+=img->nChannels)
            {
                  int j1 = j/img->nChannels;
                  // intensity is Xt according to Stauffer et. al.
                  intensity =
colorToIntensity(data[j+2],data[j+1],data[j]);
                  isBackgroundPixel = (containerPtr[i])[j1]-
                                          >updateGaussians(intensity);

                  if (isBackgroundPixel)
                  {
                        data2[j] = 0xFF;
                        data2[j+1] = 0xFF;
                        data2[j+2] = 0xFF;
                  }
                  else
                  {
                        data2[j] = 0x00;
                        data2[j+1] = 0x00;
                        data2[j+2] = 0x00;
                  }
            }
            data+=step;
            data2+=step;
      }

}


double MultipleGaussianBackgroundModel::colorToIntensity(int red, int
blue, int green)
{
            return ((0.3*red)+(0.59*green)+(0.11*blue));
}
```

HeadTop finder

HeadTopProcessor.h

```
#include <cv.h>
#include <vector>
using namespace std;

class HeadTopProcessor
{
public:
	HeadTopProcessor()
	{}

	~HeadTopProcessor()
	{}

	vector<CvPoint>* findHeadTops(IplImage*);

private:
	vector<CvPoint>* findLocalMin(CvSeq* s);
	vector<CvPoint>* headTopFilter1(vector<CvPoint>* headTops,
IplImage*);


};
```

HeadTopProcessor.cpp

```
#include <stdafx.h>
#include "HeadTopProcessor.h"
#include "..\Blob analysis package\blob.h"
#include "..\Blob analysis package\BlobResult.h"
#include <cv.h>
#include <highgui.h>
#include <fstream>
#include "..\constants.h"
using namespace std;


vector<CvPoint>* HeadTopProcessor::findHeadTops(IplImage* img)
{
	CBlobResult blobs;
	CBlob Blob;
	vector<CvPoint>* localMins;
	vector<CvPoint>* filteredLocalMins1;
	vector<CvPoint>* allLocalMins = new vector<CvPoint>;
	double height_ellipse;

	blobs = CBlobResult( img, NULL, 100, true );
	blobs.Filter( blobs, B_INCLUDE, CBlobGetArea(), B_LESS, 600 );
	blobs.Filter( blobs, B_INCLUDE, CBlobGetArea(), B_GREATER, 100 );

	for  (int i=0; i<blobs.GetNumBlobs(); ++i)
	{
		Blob = blobs.GetBlob(i);
		IplImage* f = cvCreateImage(cvGetSize(img), 8, 1);

		localMins = findLocalMin(Blob.edges);
		filteredLocalMins1 = headTopFilter1(localMins, img);

		for (vector<CvPoint>::iterator it=filteredLocalMins1-
>begin();
			it!=filteredLocalMins1->end();++it)
		{
			height_ellipse = HUMAN_HEIGHT * ((double)it->y /
					(double)cvGetSize(img).height);
			CvPoint center=cvPoint(it->x, it-
>y+(double)(height_ellipse/2));

			allLocalMins->push_back(*it);

		}



	}
	return allLocalMins;


}
```

```
// filters any headtop which does not lie on a foreground blob
vector<CvPoint>* HeadTopProcessor::headTopFilter1(vector<CvPoint>*
headTops, IplImage* img)
{

      int i,j,k,j1, n, height, step, width;
      double image_height, height_ellipse, colour2;
      vector<CvPoint>* v;
      CvPoint center;
      int* areaOccupied;
      int* areaThreshold;
      double* majorAxisArray;
      double* minorAxisArray;
      CvPoint* centerOfEllipseArray;
      CvPoint* headTopsArray;
      unsigned char *data;

      // arrayOccupied stores cumulative pixel area
      // underneath each headtop. areaOccupied runs "parallel" to
headTopsArray
      n = headTops->size();
      areaOccupied = new int[n];
      headTopsArray = new CvPoint[n];
      centerOfEllipseArray = new CvPoint[n];
      majorAxisArray = new double[n];
      minorAxisArray = new double[n];
      areaThreshold = new int[n];
      v = new vector<CvPoint>;
      i=0;
      j=0;
      image_height = (double)cvGetSize(img).height;
      height = img->height;
      step = img->widthStep;
      width = img->width * img->nChannels;
      data= reinterpret_cast<unsigned char *>(img->imageData);

      // fill headTopsArray with CvPoints in headTops vector
      for (vector<CvPoint>::iterator it=headTops->begin(); it!=headTops-
>end(); ++it)
      {
            height_ellipse = HUMAN_HEIGHT * ((double)it->y /
image_height);
            center = cvPoint(it->x, it->y+(double)(height_ellipse/2));

            headTopsArray[i] = *it;
            areaOccupied[i] = 0;                // initialize areas
            centerOfEllipseArray[i] = center;
            majorAxisArray[i] = height_ellipse/2;
            minorAxisArray[i] = HUMAN_HEAD_SIZE/2;
            /*areaThreshold[i] = \
            cvRound(((double)AREA_THRESHOLD_PERCENTAGE/100.0)* \
            HUMAN_AREA * ((double)it->y / image_height));*/
            areaThreshold[i] = \

      cvRound(((double)AREA_THRESHOLD_PERCENTAGE/100.0)*(PI*majorAxisArr
ay[i]*minorAxis
            Array[i]));


            i++;        // keeping all arrays parallel.
      }
```

```
	for (i=0;i<height;i++)
	{
		for (j=0;j<width;j++)
		{
			j1 = j / img->nChannels;
			colour2 = GlobalFunctions::getPixelColor(img,
cvPoint(j1,i));

			if (colour2 <= 10)
			{
				for (k=0;k<n;k++)
				{
					if
(GlobalFunctions::isInEllipse(centerOfEllipseArray[k],
						minorAxisArray[k],
					majorAxisArray[k],
						cvPoint(j1,i)))
					{
						areaOccupied[k]++;
					}

				}
			}
		}
		data+=step;
	}

	for (k=0;k<n;k++)
	{
		if (areaOccupied[k] >= areaThreshold[k])
			v->push_back(headTopsArray[k]);
	}

	// free up memory
	delete [] areaOccupied;
	delete [] headTopsArray;
	delete [] centerOfEllipseArray;
	delete [] majorAxisArray;
	delete [] minorAxisArray;
	delete [] areaThreshold;

	return v;



}

vector<CvPoint>* HeadTopProcessor::findLocalMin(CvSeq* edges)
{
	int minX, maxX,n, i,j,k, diff;
	bool flag1, flag2;
	CvSeqReader reader;
	CvPoint edgeactual;
	CvPoint* localMins;
	vector<CvPoint>* v;

	// initialization
	v = new vector<CvPoint>;
	cvStartReadSeq( edges, &reader);
	CV_READ_SEQ_ELEM( edgeactual ,reader);
	minX = edgeactual.x;		// first edge in sequeunce
	maxX = edgeactual.x;		// first edge in sequeunce
```

```
      // find minimum X and maximum X
      for(i=1; i< edges->total; i++)
      {
            CV_READ_SEQ_ELEM( edgeactual ,reader);
            if (edgeactual.x < minX)
                  minX = edgeactual.x;
            if (edgeactual.x > maxX)
                  maxX = edgeactual.x;
      }

      n = cvRound((double)(maxX-minX)/(double)HUMAN_HEAD_SIZE);

      // initialize an array that will store local minimums
      localMins = new CvPoint[n];
      for (i=0;i<n;i++)
      {
            localMins[i] = cvPoint(10000,10000);
      }

      // search for local mins in a neighborhood.specified by human
head-size
      cvStartReadSeq( edges, &reader);    // reinitialize iterator
      for (i=0;i<edges->total;i++)
      {
            CV_READ_SEQ_ELEM(edgeactual, reader);
            flag1= true;
            flag2 = true;

            for (j=0;j<n && flag2; j++)
            {
                  if (edgeactual.y < localMins[j].y)
                  {
                        for (k=0;k<n && flag1;k++)
                        {
                              diff = abs(edgeactual.x - localMins[k].x);

                              if (diff < HUMAN_HEAD_SIZE && edgeactual.y
<
                                    localMins[k].y)
                              {
                                    localMins[k] = edgeactual;
                                    flag1 = false;
                                    flag2 = false;
                              }
                              else if (diff < HUMAN_HEAD_SIZE &&
edgeactual.y >=
                                    localMins[k].y)
                              {
                                    flag1 = false;
                              }
                        }

                        if (flag1)
                        {
                              localMins[j] = edgeactual;
                              flag2 = false;
                        }
                        flag1 = true;
                  }
            }
      }
```

```
	// fill localMins array into a vector for return
	// discard points (10000,10000)
	for (i=0;i<n;i++)
	{
		if (!(localMins[i].x == 10000 && localMins[i].y == 10000))
		{
			v->push_back(localMins[i]);
		}
	}

	return v;


}
```

Motion Tracker

HotSpot.h

```
// OpenCV
#include "cxcore.h"

class HotSpot
{
public:
	CvPoint center;
	double radius;


	HotSpot(CvPoint p, double r)
	{
		center = p;
		radius = r;
	}


	HotSpot()
	{
		center = cvPoint(0,0);
		radius = 0;
	}

	bool pointInHotSpot(CvPoint point);
	void update(CvPoint p, double r);


};
```

HotSpot.cpp

```
#include "stdafx.h"
#include "HotSpot.h"
// checks if a point (x,y) is in within the hotspot defined
// by a circle with center (a,b) and equation:
// (x-a)^2 + (y-b)^2 <= r^2

bool HotSpot::pointInHotSpot(CvPoint point)
{
	int d = (point.x - center.x)*(point.x-center.x) + (point.y -
center.y)*(point.y-center.y);
	return (d <= (radius*radius));
}
/*
bool HotSpot::pointInHotSpot(CvPoint point)
{
	bool cond1, cond2;

	cond1 = point.x >= center.x-radius && point.x <= center.x+radius;
	cond2 = point.y >= center.y-radius && point.y <= point.y+radius;

    return cond1 && cond2;
}*/

void HotSpot::update(CvPoint p, double r)
{
	center = p;
	radius = r;
}
```

TrackedBlob.h

```
// OpenCV
#include <cv.h>
#include <cxcore.h>
#include <highgui.h>
#include "..\Blob analysis package\blob.h"
#include "..\Blob analysis package\BlobResult.h"
#include "HotSpot.h"
#include "..\constants.h"
#include <vector>

using namespace std;
const double proc_cov=1e-5;
const double meas_cov=1e-5;
//const float A[] = {1,1,0,1};
const float A[] = {1,1,0,1};

// a registered blob
class TrackedBlob
{
public:
	HotSpot spot;					// contains center of the blob

	// new code
	CvPoint previousPosition; // (x0, y0)
	CvPoint currentPosition;  // (xi, yi)
	double speed;
	int lostBlobCount;
	bool trackedInCurrentFrame;


public:
	int red;
	int blue;
	int green;
	int status;		// TRACKED - Trackedblob was found in the
current frame
				// ON_SEARCH - Blob is still being
searched for

	vector<CvPoint> motionHistory;

	// kalman parameters
	CvKalman* kalman;
	CvMat* measurement;

	TrackedBlob(CvPoint);


	bool isThisTrackedBlob(CvPoint centerOfUnTrackedBlob);
	void updateTrackedBlob(CvPoint newCenterOfBlob, CvPoint
currentPosition, \
		CvPoint previousPosition, double radius, int status, double
speed,
		int lostBlobCount);

	void addToMotionHistory(CvPoint);
	void trackedBlobInit2(CvPoint);
	CvPoint getEstimatedPosition(CvPoint observedPoint);
	CvPoint getEstimatedPosition();
	void estimateTrackAndUpdate(CvPoint);
	void estimateTrackAndUpdate();
	CvPoint GetCenter(CBlob);
```

```
	// SET METHODS
	inline void setTrackedInCurrentFrame(bool t) {
trackedInCurrentFrame = t; }
	inline void setStatus(int s) { status = s; };

	// GET METHODS
	inline bool getTrackedInCurrentFrame() { return
trackedInCurrentFrame; }
	inline int getLostBlobCount() { return lostBlobCount; }


private:
	void updateHotSpot(CvPoint newCenterOfHotSpot, double radius);
	double distance(CvPoint, CvPoint);
	CvPoint getPredictedPosition();
	CvPoint kalmanFilter(CvPoint);

};
```

TrackedBlob.cpp

```
#include "stdafx.h"
#include "TrackedBlob.h"


TrackedBlob::TrackedBlob(CvPoint centerOfBlob) {


            spot.center = centerOfBlob;
            spot.radius = DEFAULT_HOTSPOT_RADIUS;
            red=rand()%255;
            blue=rand()%255;
            green=rand()%255;
            status = NEW_BLOB;
            trackedInCurrentFrame = false; // only tracker can confirm
this info.

            motionHistory.push_back(centerOfBlob);

            // new code
            previousPosition = centerOfBlob;
            speed = 0;        // when a blob is spotted first, its speed
is not
                              // known, until it is tracked in the
subsequent frames

            lostBlobCount = 0;

            // kalman filter
            kalman = cvCreateKalman(4,2,0);
            cvSetIdentity(kalman->measurement_matrix, cvRealScalar(1));
            cvSetIdentity(kalman->process_noise_cov,
cvRealScalar(proc_cov));
            cvSetIdentity(kalman->measurement_noise_cov,
cvRealScalar(meas_cov));
            cvSetIdentity(kalman->error_cov_post, cvRealScalar(1));
            memcpy( kalman->transition_matrix->data.fl, A, sizeof(A));
            CvRNG rng = cvRNG(-1);
            cvRandArr(&rng, kalman->state_post, CV_RAND_NORMAL,
cvRealScalar(0),
                        cvRealScalar(0.1));
            kalman->state_post->data.fl[0]=centerOfBlob.x;
            kalman->state_post->data.fl[1]=centerOfBlob.y;

            measurement = cvCreateMat(2,1,CV_32FC1);
}


bool TrackedBlob::isThisTrackedBlob(CvPoint centerOfUnTrackedBlob)
{
      return spot.pointInHotSpot(centerOfUnTrackedBlob);
}


void TrackedBlob::updateHotSpot(CvPoint newCenterOfHotSpot, double
radius)
{
      spot.update(newCenterOfHotSpot, radius);
}
```

```
// updates a tracked blob and its hotspot
void TrackedBlob::updateTrackedBlob(CvPoint newCenterOfBlob, CvPoint
cPos, CvPoint pPos, double radius, int s, double sp, int bc)
{

      status = s;
      currentPosition = cPos;
      previousPosition = pPos;
      speed = sp;
      lostBlobCount = bc;

if (newCenterOfBlob.x == 0 && newCenterOfBlob.y ==0)
{
      int i=0;
}

      // update the blob's hotspot
      updateHotSpot(newCenterOfBlob, radius);

      // add to motion history
      addToMotionHistory(cPos);

}

void TrackedBlob::addToMotionHistory(CvPoint p)
{
      motionHistory.push_back(p);
}

// the second time a blob is tracked, we have a value for (x1,y1)
// and hence we can calculate its velocity, and predict its motion
void TrackedBlob::trackedBlobInit2(CvPoint c)
{
      CvPoint centerOfBlob = c;
      speed = distance(centerOfBlob, previousPosition) / FRAME_RATE;

      if (centerOfBlob.y == previousPosition.y && centerOfBlob.x ==
previousPosition.x)
      {
            updateTrackedBlob(centerOfBlob, centerOfBlob,
previousPosition,
                              DEFAULT_HOTSPOT_RADIUS, NEW_BLOB, speed,
                        0);
      }
      else
      {
            // update tracked blob and hotspot
            updateTrackedBlob(centerOfBlob, centerOfBlob,
previousPosition,
                              DEFAULT_HOTSPOT_RADIUS, BEING_TRACKED,
                        speed, 0);
      }
}

// for tracked blobs for which an observed blob has been detected in its
hotspot
void TrackedBlob::estimateTrackAndUpdate(CvPoint c)
{
      CvPoint centerOfBlob = c;
      CvPoint nextEstimatedPosition =
getEstimatedPosition(centerOfBlob);

      // update tracked blob and prepare for next estimation
      speed = distance(centerOfBlob, currentPosition) / FRAME_RATE;
```

```
      updateTrackedBlob(nextEstimatedPosition, nextEstimatedPosition,
currentPosition,
                        DEFAULT_HOTSPOT_RADIUS, BEING_TRACKED, speed,0);


}
```

```
// for tracked blobs for which no observed blob has been detected in its
hotspot
void TrackedBlob::estimateTrackAndUpdate()
{

      CvPoint nextEstimatedPosition = getEstimatedPosition();
      speed = distance(nextEstimatedPosition,
currentPosition)/FRAME_RATE;
      updateTrackedBlob(nextEstimatedPosition, nextEstimatedPosition,
currentPosition,
                        DEFAULT_HOTSPOT_RADIUS, BLOB_LOST, speed,
++lostBlobCount);

}



// the distance between two CvPoints
double TrackedBlob::distance(CvPoint a, CvPoint b)
{

      return  sqrt((a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y));
}


// function can only be called when blob's status is set to
BEING_TRACKED or LOST_BLOB
// function should not be called when blob's status is NEW_BLOB
CvPoint TrackedBlob::getPredictedPosition()
{
      /*
      // calculate the predicted position using position vectors
      // velocity of blob
      double distanceTravelledInLastFrame =
distance(currentPosition,previousPosition);
      double predictedDistanceToBeTravelled = speed * FRAME_RATE;


      // lamba is the scalar constant found in the vector equation of
straight
      // lines
      double lambda = 1 +
predictedDistanceToBeTravelled/distanceTravelledInLastFrame);

      double predictedXOrdinate = (1-lambda)*previousPosition.x +
                                  (lambda*currentPosition.x);
      double predictedYOrdinate = (1-lambda)*previousPosition.y +
                                  (lambda*currentPosition.y);

      if (distanceTravelledInLastFrame == 0)
      {
            return currentPosition;
      }

      return cvPoint(cvRound(predictedXOrdinate),
cvRound(predictedYOrdinate));*/
      return kalmanFilter(currentPosition);
}
```

```
// estimate the position of the blob when it has been observed
CvPoint TrackedBlob::getEstimatedPosition(CvPoint observedPoint)
{
      CvPoint predictedPosition = getPredictedPosition();

      double estimatedPositionXOrdinate = ((1-
            TRACKER_RELIABILITY_FACTOR)*predictedPosition.x) +
            (TRACKER_RELIABILITY_FACTOR*observedPoint.x);

      double estimatedPositionYOrdinate = ((1-
            TRACKER_RELIABILITY_FACTOR)*predictedPosition.y) +
            (TRACKER_RELIABILITY_FACTOR*observedPoint.y);

      return cvPoint(cvRound(estimatedPositionXOrdinate),
                        cvRound(estimatedPositionYOrdinate));
}

// estimate the position when there is no observed point
// useful when blob is lost
CvPoint TrackedBlob::getEstimatedPosition()
{

      // calculate the predicted position using position vectors
      // velocity of blob
      double distanceTravelledInLastFrame =
distance(currentPosition,previousPosition);
      double predictedDistanceToBeTravelled = speed * FRAME_RATE;


      // lamba is the scalar constant found in the vector equation of
straight
      // lines
      double lambda = 1 +

            (predictedDistanceToBeTravelled/distanceTravelledInLastFrame
            );

      double predictedXOrdinate = (1-lambda)*previousPosition.x +
                                    (lambda*currentPosition.x);
      double predictedYOrdinate = (1-lambda)*previousPosition.y +
                                    (lambda*currentPosition.y);

      if (distanceTravelledInLastFrame == 0)
      {
            return currentPosition;
      }

      //return kalmanFilter(cvPoint(cvRound(predictedXOrdinate),
      cvRound(predictedYOrdinate)));
      CvPoint c =  cvPoint(cvRound(predictedXOrdinate),
cvRound(predictedYOrdinate));
      return kalmanFilter(c);
}
```

```
CvPoint TrackedBlob::GetCenter(CBlob Blob)
{
                  CBlobGetXCenter getXc;
                  CBlobGetYCenter getYc;
                  double blobCentre[2];
                  blobCentre[0] = getXc(Blob);
                  blobCentre[1] = getYc(Blob);
                  return cvPoint(blobCentre[0], blobCentre[1]);

}

CvPoint TrackedBlob::kalmanFilter(CvPoint m)
{
      measurement->data.fl[0] = m.x;
      measurement->data.fl[1] = m.y;

      cvKalmanPredict(kalman, 0);
      cvKalmanCorrect(kalman, measurement);
      return cvPoint(cvRound(kalman->state_post->data.fl[0]),
                        cvRound(kalman->state_post->data.fl[1]));

}
```

Tracker.h

```
#include "TrackedBlob.h"

#include <vector>
#include <fstream>
using namespace std;

class Tracker
{
	vector<TrackedBlob>* trackedBlobs;

public:
	Tracker()
	{
		trackedBlobs = new vector<TrackedBlob>;
	}

	~Tracker()
	{
		delete trackedBlobs;
	}

	void track(IplImage* img, vector<CvPoint>* blobCenters);


};
```

Tracker.cpp

```
#include "stdafx.h"
#include "Tracker.h"
#include "..\Blob analysis package\blob.h"
#include "..\Blob analysis package\BlobResult.h"
//#include "..\constants.h"
#include <vector>
using namespace std;

void Tracker::track(IplImage* img, vector<CvPoint>* blobCenters)
{
	CBlobResult blobs;
	CvPoint blobCenter;
	bool blobTracked;
	vector<TrackedBlob>* newBlobs;

	newBlobs = new vector<TrackedBlob>;
	blobTracked = false;

	for (vector<CvPoint>::iterator it0 = blobCenters->begin();
!blobTracked &&
		it0!=blobCenters->end(); ++it0)
	{
			blobCenter = *it0;


			// iterate through every registered blob
			for (vector<TrackedBlob>::iterator it = trackedBlobs-
>begin();
				!blobTracked && it!=trackedBlobs->end(); ++it)
			{

				if (it->isThisTrackedBlob(blobCenter))
				{

					if (it->status == NEW_BLOB)
					{
						// we can now measure the velocity,
and hence
						// predict motion
						it->trackedBlobInit2(blobCenter);
					}
					else if (it->status == BEING_TRACKED)
					{
						// an old tracked blob, keep
tracking with observed
						// point
						it-
>estimateTrackAndUpdate(blobCenter);
					}
					else if (it->status == BLOB_LOST)
					{
						// an old tracked blob, keep
tracking with observed
						// point
						it-
>estimateTrackAndUpdate(blobCenter);
					}

					// notify that this blob has been tracked
in the current
					// frame
					it->setTrackedInCurrentFrame(true);
					blobTracked = true;
```

```
                }
            }
```

```
                  if (!blobTracked)
                  {

                        TrackedBlob t1(blobCenter);
                        // new blob, didnt fall inside any hotspot,
collect to register
                        // this new blob
                        newBlobs->push_back(t1);
                  }


      } // end for

      // blobs not tracked in this round, find them and change their
statuses
      for (vector<TrackedBlob>::iterator it2=trackedBlobs->begin();
            it2!=trackedBlobs->end(); ++it2)
      {
            if (!(it2->getTrackedInCurrentFrame()))
            {
                  // blob was not tracked in this frame
                  // it either got lost, or was lost already.
                  // in latter case, check if its been lost for too long

                  // blob tracked in the previous frame, but didnt get
track this time
                  if (it2->status == BEING_TRACKED)
                  {
                        // this estimates position and changes status to
LOST_BLOB
                        it2->estimateTrackAndUpdate();
                  }

                  // blob not tracked in the previous frame(s), and not
tracked
                  // this time either
                  else if (it2->status == BLOB_LOST)
                  {
                        if (it2->getLostBlobCount() >
MAX_LOST_BLOB_COUNT)
                        {
                              it2->setStatus(DISCARD_BLOB);
                        }
                        else
                        {
                              // MAX_LOST_BLOB_COUNT not reached, keep
estimating
                              it2->estimateTrackAndUpdate();
                        }
                  }

                  // blob that was registered as a new tracked blob in
the previous
                  // frame, and now got lost. In such cases, we cannot
                  predict motion
                  // since we dont know the direction
                  // of the motion and velocity
                  else if (it2->status == NEW_BLOB)
                  {
                        it2->setStatus(DISCARD_BLOB);
                  }
            }
            else
            {
                  // maintenance operation
```

```
                  // blobs that were tracked, must have their flag
resetted for next
                  // frame
                  it2->setTrackedInCurrentFrame(false);
            }



      }


      // remove blobs that are marked for discard
      vector<TrackedBlob>::iterator it1=trackedBlobs->begin();
      while (it1!=trackedBlobs->end())
      {
            if (it1->status == DISCARD_BLOB)
            {
                  it1=trackedBlobs->erase(it1);
            }
            else
            {
                  it1++;
            }
      }


      // register collected new blobs that didnt fall into any hotspot
      // insert blobs that havent been tracked before.
      for (vector<TrackedBlob>::iterator it3=newBlobs->begin();
            it3!=newBlobs->end();++it3)
      {
            trackedBlobs->push_back(*it3);
      }

      // output motion history of all tracked blobs
      for (vector<TrackedBlob>::iterator it4=trackedBlobs->begin();
            it4!=trackedBlobs->end();++it4)
      {
            /*
            vector<CvPoint>::iterator it5=(it4->motionHistory).begin();
            while (it5!=(it4->motionHistory).end())
            {
                  cvCircle(img, *it5, 3, CV_RGB(it4->red, it4->green,
it4->blue), 1);
                  it5++;
            }*/
            GlobalFunctions::drawPath(img, it4->motionHistory,
                              cvScalar(it4->red, it4->green, it4-
                        >blue));
      }

}
```

Driver.cpp

```
// some helped functions
// char* to CString converter
CString convert(char* s)
{
	CString c;
	for (int i=0;s[i]!='\0';i++)
	{
		c+=s[i];
	}
	return c;
}

// pads numZero zeros to the left-side of integer
// useful for filenames of image files
// with names of this format: cyard7-00001.jpeg
CString padZerosToLeft(int i, int numZeros)
{
	CString s, temp2;
	char* temp1 = new char[256];
	_itoa(i,temp1,10);
	temp2 = convert(temp1);

	int zeros = numZeros - temp2.GetLength();
	for (int j=0;j<zeros;j++)
	{
		s+='0';
	}

	return s+temp1;

}

// the median filter
IplImage* medianFilter(IplImage* img)
{
	if (img!=NULL)
	{
		IplImage* cpy_img = cvCloneImage(img);
		cvSmooth(img,cpy_img,CV_MEDIAN,3,0,0);
		return cpy_img;
	}
	else
		return 0;

}
```

```
// driver
void myDriver()
{
	// path of the image frames of video input
	CString s = "C:\\visionDataSets\\org_data\\all\\cyard7-";

	CString filename, org_filename;


	IplImage* img1, *one_channel_output, *output, *output_median;
	MultipleGaussianBackgroundModel* multGPtr;
	HeadTopProcessor* ht;
	Tracker* t;
	vector<CvPoint>* v;

	// output to this window
	cvNamedWindow("scene", 1);


	t = new Tracker();
	ht = new HeadTopProcessor();

	// iterate 9000 image frames
	for (int i=0;i<9000;i++)
	{
		filename = s+padZerosToLeft(i,6)+".jpeg";
		img1 = cvLoadImage(filename);

		if (i == 0)
		{
			// initialize background model with zero mean and
std. dev.
			multGPtr = new MultipleGaussianBackgroundModel(img1);
			output = cvCreateImage(cvGetSize(img1),
IPL_DEPTH_8U, 3);

		}
		else
		{
			multGPtr->updateModel(img1, output);
			output_median = medianFilter(output);

			one_channel_output = cvCreateImage(cvGetSize(img1),
IPL_DEPTH_8U,
                                   1);
			cvSplit(output_median, one_channel_output,0,0,0);


			v = ht->findHeadTops(one_channel_output);       //
find headtops
			t->track(img1, v);                              //
track

			cvShowImage("scene", img1);


			cvReleaseImage(&img1);
			cvReleaseImage(&one_channel_output);
			cvReleaseImage(&output_median);
```

```
            }
        }
        cvDestroyWindow("scene");
    }
```