Project Report

Automated Busy Discovery

Subhrendu Sarkar

Internet Real Time Lab, Columbia University

Fall - 2007

Table of Contents

1. Abstract

We describe an efficient and robust software system which can detect and track humans in a room and update the status of the person at a Presence Server. This aims in understanding the state of a user being busy/idle. The information being available on Presence Server, it can accessed by user agents and clients to fulfill appropriate actions. Of special consideration in the design of this system are real-time and robustness issues. We thus utilize a detection/tracking scheme in which we detect the full body contour of the user(s) and track the detected contour of the user(s). Robustness is implicit in this design , as the system automatically detects the limbs as part of the same contour and hence the same user. Also it detects any user entering or leaving the scene and tracks the user as long as he/she is within the scene. The status of the user is updated at the Presence Server with a SIP (Session Initiation Protocol) Message being sent to the Presence Server from the client software when the user is detected to be busy. The design is conducive to real-time processing as detection and tracking is not performance intensive and thus reasonable frame-rate can be achieved with a short latency. Experiments on real-time setting demonstrate the efficacy of this approach.

1 Introduction

1.1. Objective

We aim to design and build an application that detects if a user is likely to be receptive to receiving phone calls or otherwise being disturbed and whether the user is physically present. Installed Cameras are used to capture video and image analysis of the video is done to detect the presence of multiple persons (e.g. for meetings) in a room.

Presence of more than one person in a room is assumed to be an indication that the user (individual) is most likely to be busy. The software aims in generating presence data and influence call behavior, so that incoming calls are automatically redirected to another party or voice mail.

1.2. Motivation

On many occasions it has been found that telephone calls create disturbances in between some important meetings going on in a room. The software aims to automatically detect the status of the user in a room (busy/idle) and correspondingly update the Presence Server with the status of the user so that appropriate action is taken for an incoming call. The proxy can request the status of an user from the Presence Server and decide to allow the incoming call or may decide to route the incoming call to another phone number or the voicemail.

The field of computer vision is concerned with problems that involve interfacing computers with their surrounding environment through visual means. One such problem, object recognition, involves detecting the presence of a known object in an image, given some knowledge about what that object should look like. As humans, we take this ability for granted, as our brains are extraordinarily proficient at both learning new objects and recognizing them later. However, in computer vision, this same problem has proven to be one of the most difficult and computationally intensive of the field. Given the current state of the art, a successful algorithm for object recognition requires one to define the problem with a more specific focus.

2. Software Architecture

The software architecture is shown in the figure below:

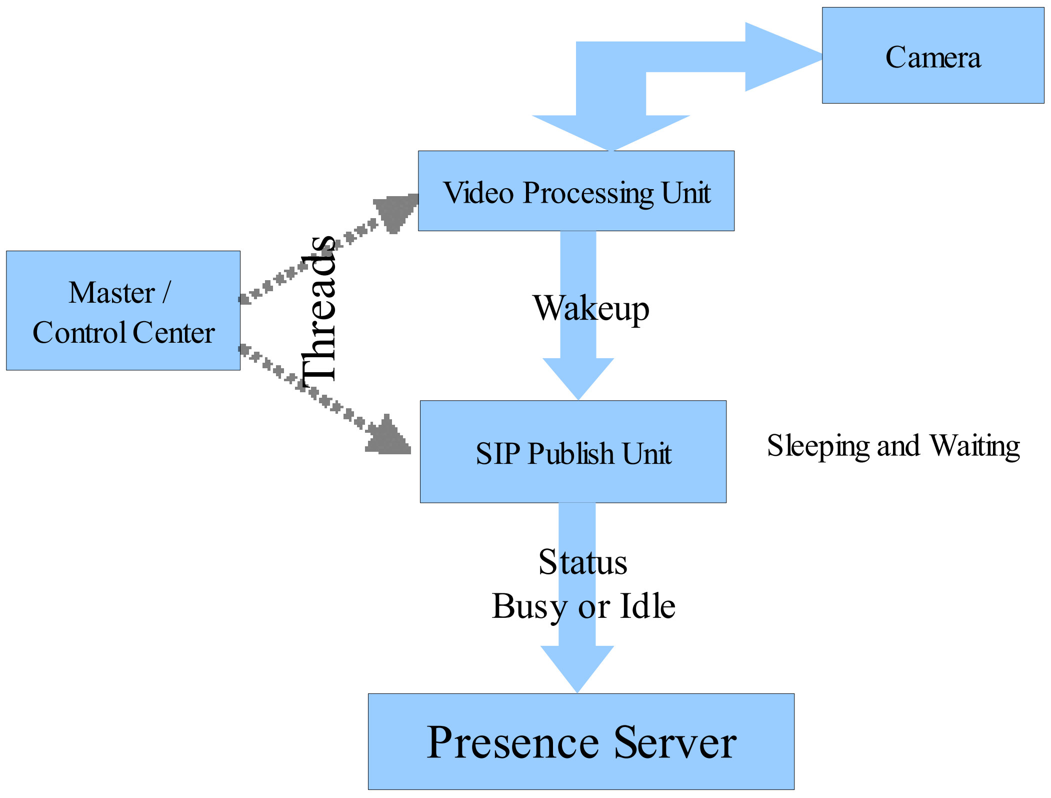


Figure – 1: Software Architecture

The software consists of three main components:

1. **The Master Control Unit**
2. **The Video Processing Unit**
3. **The SIP Publish Unit**

Upon execution of the program, the Master Control Unit is initialized and it creates two threads:
a) The Video Processing Unit Thread: This thread is responsible for the processing and analysis of the video captured. It detects the objects in a scene by analyzing every frame of the video. The algorithms used in this process will be discussed in detail in later sections. When the Video Processing Thread finds that there are more than equal to two humans in the room, it signals the SIP Publish Unit thread to publish a pidf/xml on the Presence Server.

b) The SIP Publish Unit Thread: This thread is created by the Master Control Unit and it remains in a listening mode. It initializes the communication with the Presence Server and waits for the signal from Video Processing Thread to publish an event on the Presence Server. As soon as it receives the signal from the Video Processing Thread it publishes an event on the Presence Server.

2.1. The Video Processing Unit

The primary task of the Video Processing unit is to find out how many humans are there in a continuous video. We present an operational computer vision system for real-time detection and tracking of human motion. The system captures monocular video of a scene and identifies those moving objects which are characteristically human. This serves as both a proof-of concept and a verification of other existing algorithms for human motion detection. The Video Processing Unit derives its results on the basis of two major components – Object Detection and Object Tracking, each of which has its own algorithms for artificial automatic detection and tracking respectively. The results of experiments with the system indicate the ability to minimize both false detections and missed detections.

2.2. The SIP Publish Unit

The primary task of the SIP Publish Unit is to publish a pidf/xml (Presence Information Data Format -RFC 4480) [3] on a presence server when asked to do so by the Video Processing Unit. The software uses an open-source SIP library for the SIP communication. The library used is Sofia-SIP from Nokia. (Download the library from http://opensource.nokia.com/projects/sofia-sip/index.html).

3. Object Detection Algorithms

In this section we will highlight some of the object detection algorithms we have implemented for our experiments and will also analyze them for their advantages and disadvantages on the basis of the real-time results obtained.

In order to avoid wasting time on the background image, we use image segmentation to separate foreground portions of the scene from the background. Image segmentation is the action of any algorithm that separates regions of an image in a way that resembles how a human would naturally perceive them. Since we are interested in motion, a natural approach is to segment those regions of the image that are moving relative to the background. This process is called background subtraction, and is discussed further in Section 3.1.

3.1 Image Segmentation

Our goal in image segmentation is to separate background areas of the image from foreground regions of motion that are of interest for human tracking. In this project, we make the fundamental assumption that the background will remain stationary. This assumption necessitates that the camera be fixed and that lighting does not change suddenly.

A naive description of the approach[4] can be detecting the foreground objects as the difference between the current frame and an image of the scene's static background:

$$|\mathbf{frame}_i - \mathbf{background}_i| > \mathbf{Th}$$

The basic method[12] is Frame difference given by:

$$|\mathbf{frame}_i - \mathbf{frame}_{i-1}| > \mathbf{Th}$$

The important factors attached with it are:

1. The estimated background is just the previous frame
2. It evidently works only in particular conditions of objects' speed and frame rate

3. Very sensitive to the threshold Th

Another way of Background modeling is by finding the running average [4]. It is mathematically represented by:

$$B_{i+1} = \alpha * F_i + (1 - \alpha) * B_i$$

1. α, the learning rate, is typically 0.05
2. There is no more memory requirements
3. The background model at each pixel location is based on the pixel's recent history. In many works, such history is:
   a) just the previous n frames
   b) a weighted average where recent frames have higher weight
4. In essence, the background model is computed as a chronological average from the pixel's history.

It is possible to achieve accurate image segmentation without this assumption, but such generality would require more computationally expensive algorithms. Given our assumption, the algorithm of choice is background subtraction, in which we compute a model of the image background over time. For any given frame of video, we can subtract this background image from it. Those pixels with a result near zero are treated as background and those pixels with a larger result are treated as foreground. Thus, once we have the model of the background image, this algorithm is simple, efficient, and easy to implement.

Acquiring the background model, on the other hand, is more complicated. The most straightforward approach would be to simply set up the camera, empty the scene of any moving objects, and take a snapshot. Although this approach is simple, it is always impractical in real scenes because backgrounds can change over time, it can be difficult to empty a scene, lighting can change subtly, and the camera position can drift due to any movement of the camera which can be happen due to manual interventions or a gust of wind from the window.

A more practical approach is one that can adapt to a slowly changing background in real-time, which we will now describe. [12]

Consider the time-varying value of a pixel at position *(x, y)* of a grayscale video sequence.

We will refer to this value as $V_{x,y}(t)$. We can treat the value as a random process of variable

$$X_t = V_{x,y}(t)$$

Now, suppose we can model the probability of observing the current pixel value as a mixture of K Gaussian distributions.[4, 12] This probability is,

$$P(X_t) = \sum_{i=1}^{K} \omega_{i,t} \, \eta(X_t, \mu_{i,t}, \Sigma_{i,t})$$

where $\omega_{i,t}$ is an estimate of the weight of the i^th^ Gaussian, and η is the evaluation of a standard Gaussian with mean $\mu_{i,t}$ and covariance matrix $\sum_{i,t}$. [12]

$$\eta(X, \mu, \Sigma) = \frac{1}{(2\pi)^{\frac{n}{2}} |\Sigma|^{\frac{1}{2}}} e^{-\frac{1}{2}(X-\mu_t)^T \Sigma^{-1} (X-\mu_t)}.$$

Since the background is assumed to be static, the value of pixels which are part of the background can

be represented by one or more Gaussians with a small variance due to image noise alone. More than one Gaussian is a possibility for bimodal scenes such a objects swaying in the scene or a flashing light. Furthermore, in most scenes, the background will be visible more often than foreground at any given pixel, so the Gaussian with the largest weight ω is likely the background.
These ideas now enable the following approach to background subtraction:

```
for(i = 0 to all pixels in a video frame) {
  find k Gaussians and weights that best fit the sample of the last N values taken by the
  pixel using an algorithm such as K-Means or Expectation Maximization ;
  $\mu$ = the mean of the Gaussian with the largest weight $\omega$ among the k Gaussians ;
  value of background pixel for this frame = $\mu$ ;
}
Difference Image = current_Frame – background_image;
for(i = 0 to all pixels in a video frame) {
   if ( Pixel(i) > 3 standard deviations from the mean)
        Pixel(i) = foreground;
   else
        Pixel(i) = background;
}
```

The preceding algorithm is too computationally intensive for real-time use, especially the step of fitting K Gaussians to the data for each pixel and every frame. To simplify, the background image itself need only be recomputed every N frames. Thus, for most time steps, values of each pixel are simply collected and stored for later processing that only occurs once every N frames. The disadvantage of this approach is some lag time before the background can adapt to new stationary objects.

There are few other methods of background modeling like mixture of Gaussians (MoG), texture based methods and eigen-backgrounds. Each of these methods have their own advantages and disadvantages and they behave differently according to the video/scene they analyze.
However for our purposes since our software aims to detect objects/humans typically in a conference room or a lab, we can safely make a few assumptions like:

- The background is static and does not undergo frequent changes due to lighting or other issues.
- Hence we can use some time initially to build a good robust background and then use that background for comparison with the current frame to get the foreground model.
- Also since our setup is simplistic we will like to adopt an algorithm which should be fast and less complex.

In our software we have two methods implemented:

1. Running Average
2. Gaussian Background Model

3.2. Contour Detection

After background subtraction is complete, the morphological dilate and erode operation [13] is applied in sequence to the foreground mask. Briefly, the dilate function returns the dilation of image by the structuring element. This operator is commonly known as "fill", "expand", or "grow." It can be used to fill "holes" of a size equal to or smaller than the structuring element. Used with binary images, where each pixel is either 1 or 0, dilation is similar to convolution. Over each pixel of the image, the origin of the structuring element is overlaid. If the image pixel is nonzero, each pixel of the structuring element is added to the result using the "or" operator. Letting $A \oplus B$ represent the dilation of an image A by structuring element B, dilation can be defined as:

$$C = A \oplus B = \bigcup_{b \in B} (A)_b$$

where $(A)_b$ represents the translation of A by b. Intuitively, for each nonzero element $b_{i,j}$ of B, A is translated by i,j and summed into C using the "or" operator. [13]

Erosion is the dual of dilation. It does to the background what dilation does to the foreground. Briefly, given an image and a structuring element. The erode function can be used to remove islands smaller than the structuring element. Over each pixel of the image, the origin of the structuring element is overlaid. If each nonzero element of the structuring element is contained in the image, the output pixel is set to one. Letting $A \otimes B$ represent the erosion of an image A by structuring element B, erosion can be defined as:

$$C = A \otimes B = \bigcap_{b \in B} (A)_{-b}$$

where $(A)_{-b}$ represents the translation of A by b. The structuring element B can be visualized as a probe that slides across image A, testing the spatial nature of A at each point. If B translated by i,j can be contained in A (by placing the origin of B at $i,j)$, then i,j belongs to the erosion of A by B. [13]

Dilation generally increases the sizes of objects, filling in holes and broken areas, and connecting areas that are separated by spaces smaller than the size of the structuring element. With binary images (applicable in our case), dilation connects areas that are separated by spaces smaller than the structuring element and adds pixels to the perimeter of each image object.
Erosion generally decreases the sizes of objects and removes small anomalies by subtracting objects with a radius smaller than the structuring element. With binary images, erosion completely removes objects smaller than the structuring element and removes perimeter pixels from larger image objects.
Applying dilation followed by erosion helps us fill in broken areas of a blob and then helps us sharpen the edges of the blob. This is extremely useful for accurate contour detection.
Contours are distinguished from edges as follows. Edges are variations in intensity level in a gray level image whereas contours are salient coarse edges that belong to objects and region boundaries in the image. Since we have a binary image of foreground and background, the connected components of the blobs are computed to get the contour of an object which is basically the boundary of the blob.

4. Object Tracking

The image segmentation step allows us to separate foreground objects from the scene background. However, we are still working with full images, not the individual points of motion desired for human motion detection. The problem of computing the motion in an image is known as finding the optical flow of the image. There are a variety of well-understood techniques for doing so, but the Kanade-Lucas-Tomasi [6] method stands out for its simplicity and lack of assumptions about the underlying image.

The Kanade-Lucas-Tomasi algorithm [6] uses the image's gradients to predict the new location of the feature already detected — iterating until the new location is converged upon. Since this approach is based on a Taylor series expansion, it makes no assumptions about the underlying image.
The following derivation summarizes the iterative step of the Kanade-Lucas-Tomasi algorithm[6]. Consider two images, I and J, represented as continuous functions in two dimensions. We want to track a feature of known location $x' = [x, y]^T$ in image ***I*** to image ***J***, finding its displacement $d = [dx, dy]^T$. Given a window W, we can compute the dissimilarity ε between the new and old feature as:

$$\epsilon = \iint_W \left[J(\mathrm{x}') - I(\mathrm{x}' - \mathrm{d})\right]^2 d\mathrm{x}'.$$

We can make this relationship symmetric by making the substitution x' = x + d/2

$$\epsilon = \iint_W \left[J(\mathrm{x} + \frac{\mathrm{d}}{2}) - I(\mathrm{x} - \frac{\mathrm{d}}{2})\right]^2 d\mathrm{x}.$$

Given this expression for dissimilarity, we want to solve for the value of d that minimizes ε.
Thus, we find the value of d that solves the equation,

$$\frac{\partial \epsilon}{\partial \mathrm{d}} = 0 = 2\iint_W \left[J(\mathrm{x} + \frac{\mathrm{d}}{2}) - I(\mathrm{x} - \frac{\mathrm{d}}{2})\right]\left[\frac{\partial J(\mathrm{x} + \frac{\mathrm{d}}{2})}{\partial \mathrm{d}} - \frac{\partial I(\mathrm{x} - \frac{\mathrm{d}}{2})}{\partial \mathrm{d}}\right] d\mathrm{x}.$$

<u>Equation – 4.1</u>

In order to make it possible to solve for d, we can express the value of the displaced images by their Taylor series expansion, approximating terms of second-order or higher derivatives as zero in

$$J(\mathrm{x} + \frac{\mathrm{d}}{2}) \approx J(\mathrm{x}) + \frac{d_x}{2}\frac{\partial J}{\partial x}(\mathrm{x}) + \frac{d_y}{2}\frac{\partial J}{\partial y}(\mathrm{x})$$

and,

$$I(\mathrm{x} - \frac{\mathrm{d}}{2}) \approx I(\mathrm{x}) - \frac{d_x}{2}\frac{\partial I}{\partial x}(\mathrm{x}) - \frac{d_y}{2}\frac{\partial I}{\partial y}(\mathrm{x}).$$

Equation (4.1) can now be approximated as:

$$\frac{\partial \epsilon}{\partial \mathrm{d}} \approx \iint_W \left[J(\mathrm{x}) - I(\mathrm{x}) + \frac{1}{2}\mathrm{g}^T(\mathrm{x})\mathrm{d}\right] \mathrm{g}(\mathrm{x})\, d\mathrm{x} = 0,$$

where

$$\mathrm{g} = \begin{bmatrix} \frac{\partial}{\partial x}(I + J) \\ \frac{\partial}{\partial y}(I + J) \end{bmatrix}.$$

Terms can be rearranged as follows:

$$\begin{aligned}
\iint_W \left[J(\mathrm{x}) - I(\mathrm{x}) + \frac{1}{2}\mathrm{g}^T(\mathrm{x})\mathrm{d} \right] \mathrm{g}(\mathrm{x})\, d\mathrm{x} &= 0 \\
\iint_W [J(\mathrm{x}) - I(\mathrm{x})]\mathrm{g}(\mathrm{x})\, d\mathrm{x} &= -\iint_W \frac{1}{2}\mathrm{g}^T(\mathrm{x})\mathrm{d}\mathrm{g}(\mathrm{x})\, d\mathrm{x} \\
\iint_W [J(\mathrm{x}) - I(\mathrm{x})]\mathrm{g}(\mathrm{x})\, d\mathrm{x} &= -\frac{1}{2}\left[\iint_W \mathrm{g}(\mathrm{x})\mathrm{g}^T(\mathrm{x})\, d\mathrm{x} \right] \mathrm{d}.
\end{aligned}$$

Thus, we have simplified the expression to a 2 × 2 matrix equation,

$$\mathrm{Zd} = \mathrm{e},$$

Equation – 4.2

where Z is a 2 × 2 matrix,

$$\mathrm{Z} = \iint_W \mathrm{g}(\mathrm{x})\mathrm{g}^T(\mathrm{x})\, d\mathrm{x}$$

and e is a 2 × 1 vector,

$$\mathrm{e} = 2\iint_W [I(\mathrm{x}) - J(\mathrm{x})]\mathrm{g}(\mathrm{x})\, d\mathrm{x}.$$

Equation (4.2) allows us to solve for the approximate displacement of a feature, given its starting location and the two images. Furthermore, the computed displacement has subpixel accuracy. Since we are dealing with a discrete image composed of pixels, the above definitions for Z and e are computed with a summation over the window rather than an integral. The x and y image derivatives are approximated by convolving the images with a Sobel operator.

Since the above computation for displacement is only an approximation, it is useful to repeat the procedure for more than one iteration. If the displacement does not converge towards zero after several iterations, the feature is considered lost. For features displaced by a large amount, the approximation also breaks down because the Taylor series approximation becomes less accurate. To handle such a case, it is best to perform several iterations on versions of the images re-sampled to a coarser resolution, followed by several iterations on the full-resolution images. [3,4,5,6]

A final consideration with the Kanade-Lucas-Tomasi [6] algorithm is the choice of initial features. It is wasteful to track all pixels of the starting image to the destination image. A more useful approach is to track only those pixels which represent sharp, well-defined features. In our case we track the contour of the detected Objects.

5. Summary

We have explained our approach to each of the three stages of the human motion detection system:

1. Image segmentation achieved with Running Average or a mixture of Gaussians approach to background subtraction.
2. Point feature tracking utilizing the Kanade-Lucas-Tomasi method.

The next section will show how each of these stages was implemented in a real-time working system.

6. Implementation

In the previous chapter, we discussed the choice of algorithms for each stage of our human motion detector. Although this discussion provides a good theoretical overview of how the detector works, it is not enough information to implement the system. In this chapter we discuss the practical details of how each algorithm is implemented and how they come together to form a complete system.

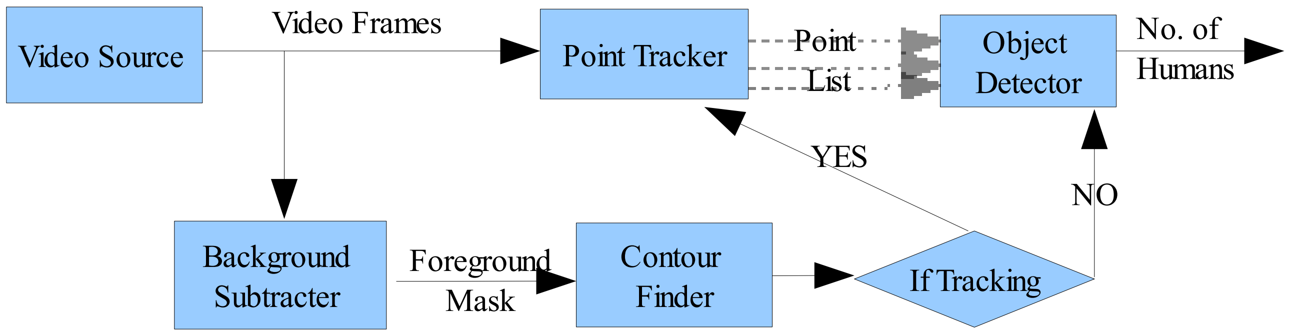


Figure – 2: Block Diagram of Video Processing Unit

6.1 Software overview

The block diagram in Figure-1 gives a high-level overview of the software architecture of the number of human detector. Input to the system is provided by a video source which can be either a IEEE 1394/ USB digital camera or a sequence of still image files in raw RGB format.

The USB digital camera (a logitech USB Digicam is used in our experiments) input allows for live video to be processed in real-time as it is captured by the camera.

Video data from the video input is made available to the background subtraction algorithm, which is responsible for differentiating between foreground and background image regions. The implementation of the background subtracter is discussed in more detail in Section 3.1. The output of the background subtracter for each video frame is an 8-bit per pixel bitmap that serves as a foreground mask: Those pixels which are foreground have value 255 and those pixels which are background have value 0.

The foreground mask provided by the background subtracter is processed to build a data structure that enumerates the boundaries for each distinct foreground object. This enumeration is achieved by the contour finder, a simple algorithm that finds each "connected component" of the foreground mask. A

connected component is defined as a region of an image whose pixels all have the same value and are adjacent to other pixels in the same connected component.
For our contour finder, we used an implementation of the algorithm provided by the Intel Open Computer Vision Library [9]. Once these contours are located, those with small geometric area are ignored. Such small contours are likely to be noise or small image disturbances that are probably not human. In our experiments, we capture video frames at a resolution of 640 x 480 and we find contour area greater than 5000 pixels is a good representation of a human being in the scene. However this value is configurable and can tend to vary according to the location, the zoom factor of the camera and the resolution of the captured video frames.
For each contour, a rectangular bounding box is computed which is used as the region of interest for the Kanade-Lucas-Tomasi point feature tracker, whose implementation is discussed in Section 4. The feature tracker outputs a list of point coordinates within this bounding box.

6.2 Background subtraction

Our implementation of background subtraction follows the general algorithm described in Section 3.1. However, there are a number of approximations and simplifications that have been made to increase speed. Since successive frames tend to be very similar, only every fourth frame is used for statistics collection.(this again is configurable).
We can express background subtraction in terms of three basic operations:
background update, which is run once every 4 frames, gathers statistics for the background model.
background subtraction, which generates a foreground mask for every frame. This step is performed by subtracting the background image from the current frame, taking the absolute value of the difference, and thresholding it with the value of three standard deviations of the average image noise.

There are different ways to estimate the standard deviation of Gaussian noise in an image. Olson(1993) showed that that average method was best and also the simplest [12]. This method consists of filtering the image *I* with the average filter and subtracting the filtered image from I. Thus a measure of noise of noise at each pixel is computed. To keep image edges from contributing to the estimate, the noise measure is disregarded if the magnitude of intensity gradient is larger than some threshold *T.* The threshold value may be found from the accumulated histogram of the magnitude of the intensity gradient.

As discussed in the approach, we maintain a set of Gaussians for each pixel of the background. Each Gaussian has a mean, variance, and weight. To simplify, the variance is assumed to be fixed, equal to the variance of the image noise. In our algorithm, the weight of a Gaussian is simply equal to the number of frames for which the pixel has taken a value within three standard deviations of that Gaussian's mean. Furthermore, we keep track of the sum of these pixel values, so that the Gaussian's mean is simply this sum divided by the number of frames. Also, each pixel has exactly five Gaussians associated with it to simplify data structures.

The procedure to update each pixel during the Background Update phase is as follows:

```
 mean of each of the five Gaussians = divide each sum of the pixel by the frame count (weight).
 if(the current pixel value is within three standard deviations of any of the means of five gaussians)
    increase that Gaussian's weight by 1 and add the current value to its sum.
else {
    replace the Gaussian of lowest weight by a Gaussian with weight equal to 1;
```

sum equal to the value of the current pixel.
}

This procedure will tend to collect a pixel's past values into the five highest weighted Gaussians that represent them. Although it is only an approximation of the exact mathematical specification in Section 3.1, it balances accuracy and efficiency. This background subtraction algorithm is easily extended to color images by applying the procedure separately to each of the red, blue, and green channels. If any one of the three channels is determined to be foreground for a given pixel, the entire pixel is marked as foreground.

6.3 Contour Finder

After background subtraction is complete, the morphological dilate and erode operation [13] is applied in sequence to the foreground mask.

Once this step is complete, the foreground mask is run through a contour finder. We use OpenCV contour finder functions for this purpose [9]. We specifically use the function *cvFindContours and cvApproxPoly.*

The function *cvFindContours* retrieves contours from the binary image and returns the number of retrieved contours.

The function *cvApproxPoly* approximates one or more curves and returns the approximation result(s). It approximates with desired precision which can be configured and for our purposes we have attained an optimum value by experimentation.[9]

6.4 Kanade-Lucas-Tomasi Feature tracking

The Kanade-Lucas-Tomasi algorithm was implemented almost exactly as described in Section 4. In fact, the code was based on a reference implementation of the KLT algorithm written by Stan Birchfield [8], although heavily modified to be optimized for speed and the details of this particular application. We use the OpenCV Library Tracking function *cvCalcOpticalFlowPyrLK* in our implementation. In order to be robust against large displacements of features, the algorithm is first run on a sub-sampled version of the image. The sub-sampled image is computed by first feeding the original through a Gaussian filter and then removing the odd-numbered rows and columns. Tracking is first performed on an image sub-sampled twice to get an approximate displacement.

The tracking formula, Equation (2.2), is used to compute the displacement for each iteration of the algorithm. The origin of the starting image is then shifted by this displacement so that the tracking equation can be reapplied. Once the displacement converges near zero, tracking is complete. If it does not converge in a few iterations, the algorithm fails.

6.5 Number of Humans in the Scene – Algorithm

The number of humans in the scene is calculated on the basis of the following algorithm.

- Detected Contours of greater than 5000 pixels of area is assumed to be a human being. (the value 5000 pixels is configurable)
- The number of humans in a scene is calculated after every 60 frames. (again the value 60 frames is configurable)
- For every 60 frames, the maximum, minimum and median number of humans detected in the scene is calculated. Presently we find the minimum as a good approximate of the number of humans detected in the scene. This may be because we are running the algorithm for every 60

frames which is fairly quick considering the frame capture rate is 30 frames/second, so we are effectively calculating for every 2 seconds and it is unlikely that there is much change in the number of humans in the scene for every 2 seconds.

7. SIP Publish Unit

We have used the Sofia-SIP library from Nokia for publishing a pidf/xml to the Presence server.
When there are more than one individual detected in the room, then the user is assumed to be busy and the pidf/xml SIP message contains:

<status><basic>closed</basic>\n</status>

otherwise:

<status><basic>open</basic>\n</status>

Sofia-SIP is an open-source SIP User-Agent library [10], compliant with the IETF RFC3261 specification [1]. It can be used as a building block for SIP client software for uses such as VoIP, IM, and many other real-time and person-to-person communication services. The primary target platform for Sofia-SIP is GNU/Linux. Sofia-SIP is based on a SIP stack developed at the Nokia Research Center. Sofia-SIP is licensed under the LGPL.
Sofia-SIP implementation follows RFC3261 [1] and related key RFCs. It supports simple presence and instant messaging, with the MESSAGE, SUBSCRIBE/NOTIFY and PUBLISH methods. NUA and NTA are the primary interfaces Sofia-SIP provides to application developers.
NUA, the User-Agent API (a higher layer interface) is ideal for implementing SIP clients such as VoIP and IM applications, and server elements implemented as SIP clients. NUA hides many complex tasks, such as dialog management, offer/answer negotiation, and registration management, from the application developer. We use NUA interface towards building the client for publishing a pidf/xml on the Presence Server.

7.1 NUA Concepts

7.1.1 NUA Stack Object

Stack object represents an instance of SIP stack and media engine. It contains reference to root object of that stack, user-agent-specific settings, and reference to the SIP transaction engine, for example. A NUA stack object is created by nua_create() function and deleted by nua_destroy() function. The nua_shutdown() function is used to gracefully release active the sessions by **nua** engine.engine.NUA stack object has type nua_t. [10]

7.1.2 NUA Operation Handle

Operation handle represents an abstract SIP call/session. It contains information of SIP dialog and media session, and state machine that takes care of the call, high-level SDP offer-answer protocol, registration, subscriptions, publications and simple SIP transactions. An operation handle may contain list of tags used when SIP messages are created by NUA (e.g. From and To headers).

An operation handle is created explicitly by the application using NUA for sending messages (function nua_handle()) and by stack for incoming calls/sessions (starting with INVITE or MESSAGE). The handle is destroyed by the application using NUA (function nua_handle_destroy()).

Indication and response events are associated with an operation handle.

NUA operation handle has type nua_handle_t. [10]

7.1.3 Stack Thread and Message Passing Concepts

The stack thread is a separate thread from application that provides the real-time protocol stack operations so that application thread can for example block or redraw UI as it likes.

The communication between stack thread and application thread is asynchronous. Most of the NUA API functions cause a send of a message to the stack thread for processing and similarly when something happens in the stack thread it sends a message to the application thread. The messages to the application thread are delivered as invokes of the application callback function when the application calls su_root_run() or su_root_step() function. [10]

7.1.4 SIP Message and Header Manipulation

SIP messages are manipulated with typesafe SIPTAG_ tags. There are three versions of each SIP tag:

- SIPTAG_<tagname>() takes a parsed value as parameter.
- SIPTAG_<tagname>_STR() takes an unparsed string as parameter.
- SIPTAG_<tagname>_REF() takes a reference as parameter, is used with tl_gets() function to retrieve tag values from tag list.
- SIPTAG_<tagname>__STR_REF() takes a reference as parameter, is used with tl_gets() function to retrieve string tag values from tag list.

For example a header named "Example" would have tags names SIPTAG_EXAMPLE(), SIPTAG_EXAMPLE_STR(), and SIPTAG_EXAMPLE_REF().

When tags are used in NUA calls the corresponding headers are added to the message. In case the header can be present only once in a message and there already exists a value for the header the value given by tag replaces the existing header value. Passing tag value NULL has no effect on headers. Passing tag value (void *) -1 removes corresponding headers from the message. [10]

7.2 The SIP PUBLISH Unit

7.2.1 Data Structures and Defines

/* type for application context data */
```
typedef struct pua_s pua_t;
#define NUA_MAGIC_T pua_t

/* type for operation context data */
typedef struct ssc_oper_s ssc_oper_t;
/* define type of context pointers for callbacks */
#define NUA_IMAGIC_T    ssc_oper_t
#define NUA_HMAGIC_T    ssc_oper_t

struct ssc_oper_s {
  /**< Remote end identity
   *
   * Contents of To: when initiating, From: when receiving.
   */
  char const   *op_ident;

  /** NUA handle */
  nua_handle_t *op_handle;

  sip_method_t  op_method;            /** REGISTER, INVITE, MESSAGE, PUBLSIH or SUBSCRIBE */
  char const   *op_method_name;
  /** Call state.
   *
   * - opc_sent when initial INVITE has been sent
   * - opc_recv when initial INVITE has been received
   * - opc_complate when 200 Ok has been sent/received
   * - opc_active when media is used
   * - opc_sent when re-INVITE has been sent
   * - opc_recv when re-INVITE has been received
   */
  enum {
    opc_none,
    opc_sent = 1,
    opc_recv = 2,
    opc_complete = 3,
    opc_active = 4,
    opc_sent_hold = 8,             /**< Call put on hold */
    opc_pending = 16               /**< Waiting for local resources */
  } op_callstate;

  int           op_prev_state;     /**< Previous call state */
  unsigned      op_persistent : 1; /**< Is this handle persistent? */
  unsigned      op_referred : 1;
  unsigned :0;
};

typedef struct ssc_conf_s ssc_conf_t;
/**
 * Configuration data for pua_s_create().
 */
struct ssc_conf_s {
  const char   *ssc_aor;        /**< Public SIP address aka AOR (SIP URI) */
  const char   *ssc_certdir; /**< Directory for TLS certs (directory path) */
  const char   *ssc_contact;              /**< SIP contact URI (local address to use) */
  const char   *ssc_media_addr;       /**< Media address (hostname, IP address) */
  const char   *ssc_media_impl;       /**< Media address (hostname, IP address) */
  const char   *ssc_proxy;  /**< SIP outbound proxy (SIP URI) */
  const char   *ssc_registrar;            /**< SIP registrar (SIP URI) */
  const char   *ssc_stun_server;/**< STUN server address (hostname, IP address) */
 };
/**
 * PUA context information structure
 */
typedef struct pua_s {
  su_home_t    ssc_home[1];            /**< Our memory home */
  char const   *ssc_name;   /**< Our name */
  su_root_t    *ssc_root;       /**< Pointer to application root */
  nua_t        *ssc_nua;        /**< Pointer to NUA object */

  ssc_oper_t   *ssc_operations;        /**< Remote destinations */
  char        *ssc_address;    /**< Current AOR */
  ssc_conf_t   conf[1];      /**< Config settings for ssc_sip.h */
} pua_t;
```

7.2.2 Initialization and Deinitialization

The following code snippet shows how the system is initialized, enters the main loop for processing the messages, and, after message processing is ended, deinitalizes the system

7.2.2.1 Initialization

```
pua_t pua[1] = {{{{sizeof(pua)}}}};
global_pua = pua;
pua->ssc_name = win->m_ssc_name/*"ss3295"*/;
//Initialize system utilities
su_init();
//initialize memory handling
su_home_init(pua->ssc_home);
//Initialize the root object
pua->ssc_root = su_root_create(pua);
pua->ssc_nua = nua_create(pua->ssc_root,
                                        event_callback, pua,
                                          TAG_NULL());
if (pua->ssc_nua) {
                        nua_set_params(pua->ssc_nua, TAG_IF(pua->conf->ssc_proxy,
                        NUTAG_PROXY(pua->conf->ssc_proxy)),  NUTAG_ENABLEMESSAGE(1),
                        NUTAG_ENABLEINVITE(1),   NUTAG_AUTOALERT(1),   NUTAG_SESSION_TIMER(3600),
                        TAG_IF(pua->conf->ssc_aor, SIPTAG_FROM_STR(pua->conf->ssc_aor)), TAG_NULL());
            }
```

7.2.2.2 DeInitialization

```
void free_thread2(void *msgp)
{
            su_root_break(global_pua->ssc_root) ;
            nua_shutdown(global_pua->ssc_nua);
            if(publish_flag == 1)
                        nua_destroy(global_pua->ssc_nua);
            su_root_destroy(global_pua->ssc_root);
            su_deinit();
}
```

7.2.2.3 Entering Main Loop For Processing of Messages:

```
            su_root_run(pua->ssc_root);  /*Enters the main loop */
```

The main loop handles any incoming messages from the Presence server, it again goes on a listening mode waiting for the Video Unit to signal for another Publish. Upon getting such a signal, it publishes a message again on the Presence server.

7.2.2.4 Publishing SIP message on Presence Server:

```
pl = sip_payload_format
   (pua->ssc_home,
    "<?xml version='1.0' encoding='UTF-8'?>\n"
    "<presence xmlns='urn:ietf:params:xml:ns:cpim-pidf'\n"
    "          entity='%s'>\n"
    "  <tuple id='%s'>\n"
    "    <status><basic>%s</basic>\n</status>\n"
    "%s"
    "  </tuple>\n"
    "</presence>\n",
    pua->ssc_address, pua->ssc_name,
    state_busy ? "closed" : "open",
    xmlnote ? xmlnote : "");

address = su_strdup(pua->ssc_home, pua->ssc_address);

 if ((op = ssc_oper_create(pua, SIP_METHOD_PUBLISH, address,
SIPTAG_EVENT_STR("presence"),TAG_END())))
 {
        global_op = op;
   printf("%s: %s %s\n", pua->ssc_name, op->op_method_name, op->op_ident);
   nua_publish(op->op_handle,
                SIPTAG_CONTENT_TYPE_STR("application/cpim-pidf+xml"),
                SIPTAG_PAYLOAD(pl),
                SIPTAG_EXPIRES_STR("3600"),
                TAG_END());
 }
```

7.2.2.5 Callback Event:

```
void event_callback(nua_event_t event, int status, char const *phrase, nua_t *nua, nua_magic_t *pua, nua_handle_t *nh,
nua_hmagic_t *op, sip_t const *sip, tagi_t tags[])
{

        tag_type_t tag = NULL;
        tag_value_t value = 0;
        switch (event) {
        case nua_r_get_params:
                printf("Event %s status %d %s\n", nua_event_name(event), status, phrase);
        break;
        case nua_r_set_params:
                printf("Event %s status %d %s\n", nua_event_name(event), status, phrase);
        break;
        case nua_r_publish:

                ssc_r_publish(status, phrase, nua, (pua_t *)pua, nh, (ssc_oper_t *)op, sip, tags);
                pthread_mutex_lock(&BusyState_mutex);
                pthread_cond_wait(&BusyState_cond, &BusyState_mutex);
                pua_publish(global_pua,g_busy_state, NULL);
                pthread_mutex_unlock(&BusyState_mutex);
        break;
        }
}
```

8. Results and Conclusions

The performance of the software is highly dependent on the accuracy and efficiency of the Video Processing Unit.
In the software, we have implemented few algorithms and one can actually evaluate and compare between them. The options available are:

1) Gaussian Background Model
   1. With Object Tracking enabled
   2. Without Object Tracking
2) Running Average Background Model
   1. With Background Model being continuously being updated

2. With Background Model being generated only once and all comparisons are with that background.

Each of these methods have their own advantages and disadvantages.

a) Gaussian Background Model
The algorithm is fairly complex and it is good in terms of modeling the background. However it is a bit slow and also since the environment in which the software operates is fairly static and stable, it does not show significant benefit over the other methods. It is around 10% slower than the other method (Running Average) used in our experiments.

b) Tracking:
Since the Gaussian Method is a continuous background update module, it makes it absolutely imperative to do tracking along with the object detection because otherwise if a person remains in the scene for a long amount of time, he/she will eventually become merged into the background and the software will lose the information about the presence of the human in the scene. By tracking, the software knows that the person is still in the scene even if the foreground model does not detect a contour for the person.
However, tracking can become really tricky when many people enter and leave the scene at the same time. Since tracking happens on feature points, it becomes difficult to determine if a point being tracked was part of a contour in a previous frame is part of the same contour in the next frame. Points from different contours can get overlapped due to occulsions and in all the cases tracking does not provide very accurate results.

c) Running Average: Background creation by running average of the pixels is a fairly simple algorithm and hence it is fast. Though it is not one of the most sophisticated algorithm it still does pretty well in the environment primarily because our environment is stable, static and invariant to much lighting or such changes. Running Average is a simple algorithm in which the the background model is created on the basis of the pixel values of the last N frames, hence in more complex dynamic backgrounds with considerable lighting changes, it may at times not provide the best background model. However our background being simple, static and less prone to lighting changes, Running Average creates a good background model. (visually verified)

d) Creation of Background Once: Using the background once definitely improves the performance as background modeling is then not done for every frame. However, to get the best out of this process, we have to make sure that the software gets a good amount of initial time to build the background in which the conference room or the lab is empty without any individuals inside it. Also it may require an occasional re-creation of the background after a pre-determined amount of time or at a specified time for instance the software may want to create a background everyday twice, once in the morning and once in the evening.

e) The software presently takes an frame buffer from the camera, processes it and then frees the frame buffer memory. This slightly affects the performance and we can find that the software operates at around 6-10 frames/second. It can possibly be made more efficient with a frame buffer queueing mechanism.

Below are a few screenshots showing how the software looks like:

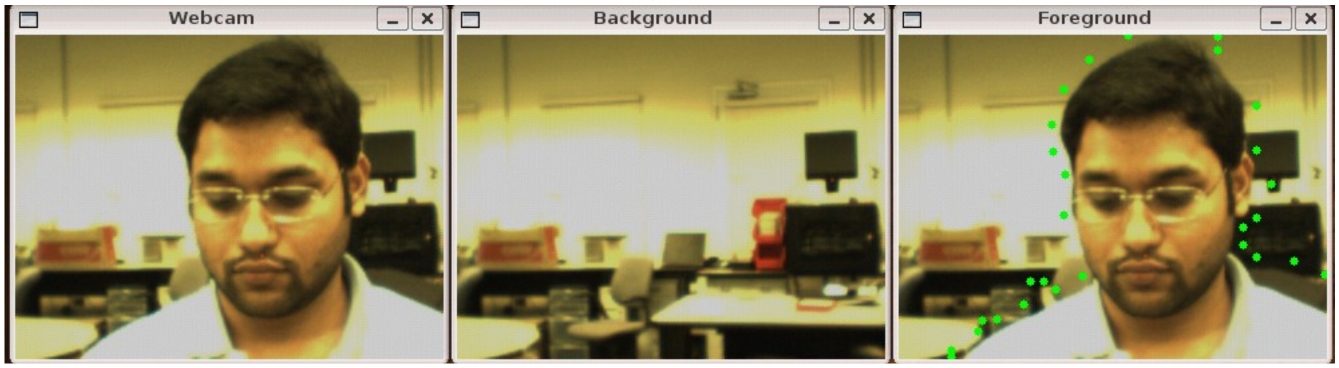


Figure-3: The green spots in the Foreground depicts the points that are being tracked.

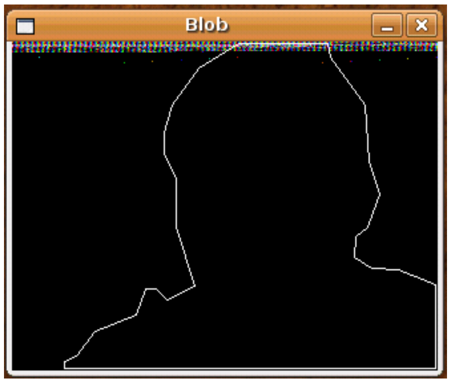
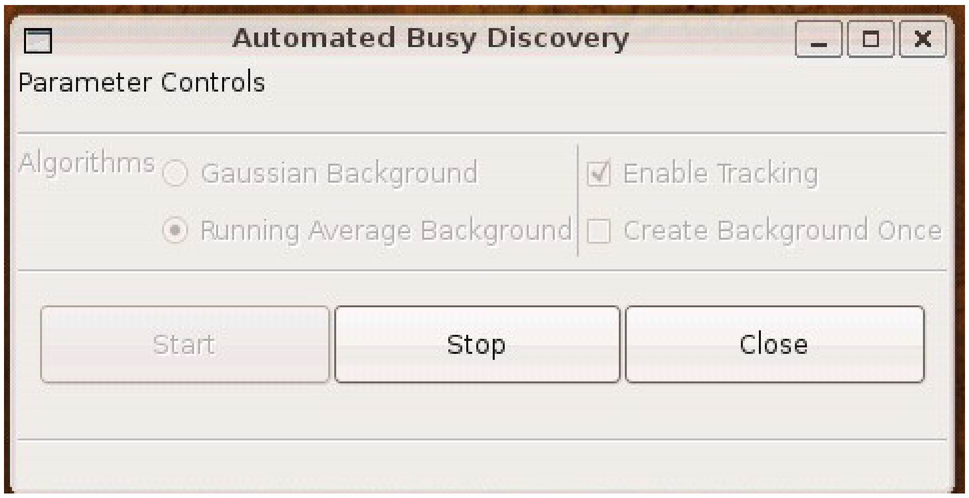


Figure-4: The Blob contour



Automated Busy Discovery

Parameter Controls

Algorithms ( ) Gaussian Background [✓ Running Average Background]

[✓ Enable Tracking]

☐ Create Background Once

Figure-5: The GUI as it looks while the software is running

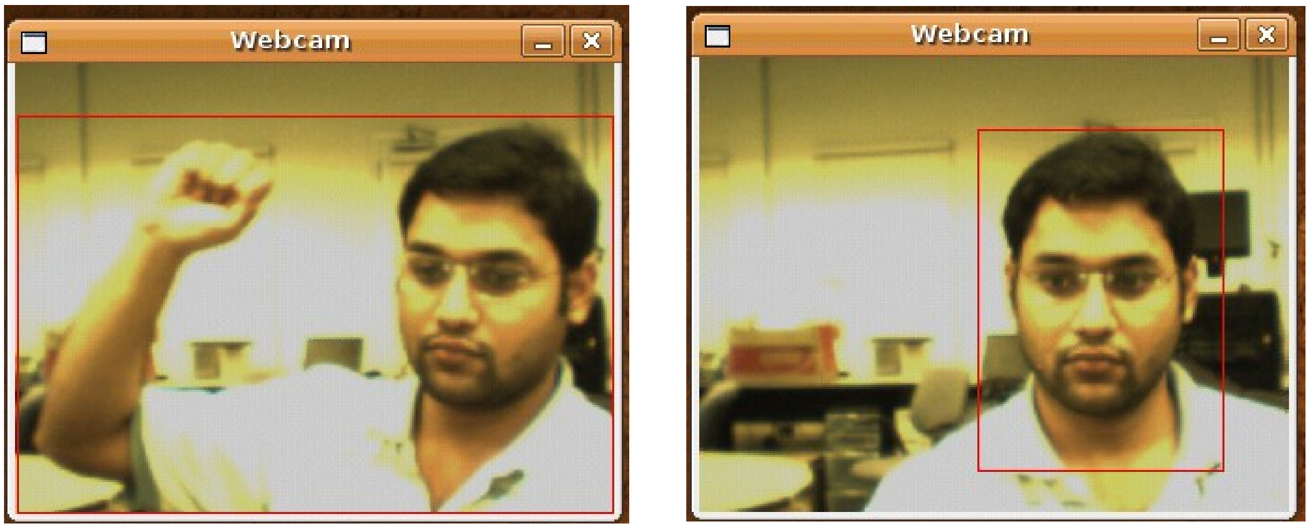


Figure-6: The red rectangle shows the rectangular block drawn on the contour that is found.

Note: Movement of hand does not lead to the finding of more than once contour, it still can acurately detect the hand as a connected component.

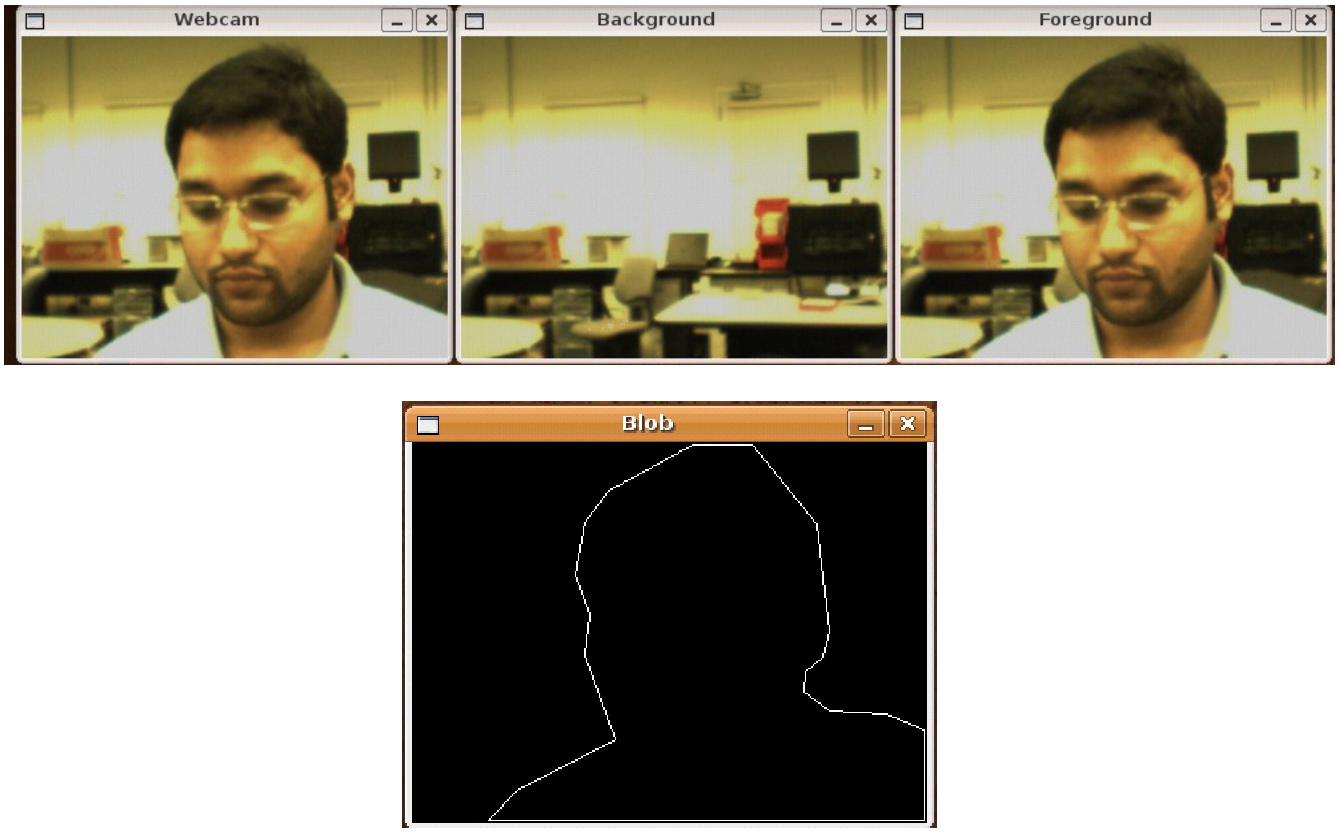


Figure- 7: Without tracking

9. Future Directions

We need to test the software in real-time actual situations and observe its performance. We need to play around with the configurable parameters of the software and observe the results.
There are couple of changes which can possibly improve the performance of the software if we find that in actual real-time situation the performance of the software is not up to the mark. In the continuous Background Model, we can aim to process every nth frame for the update process instead of doing it for every frame. That will improve the performance. Also we can do the same for the Object Detection. Also we may look at using Intel's IPP library of OpenCV to find if that improves the performance significantly.
We may try to improve the tracking system further and aim to track contours as a whole rather than just points on the contour. We can also improve some performance by having a frame buffer queueing mechanism.

10. Software requirements

1. OpenCV (Intel's Computer Vision Open Source Library)
2. GTK 2.0 (for the GUI development on Linux)
3. Linux Kernel 2.6.xx
4. Drivers of the Webcam installed in the Linux kernel.
5. Sofia-SIP library (Nokia's Open Source SIP Library)

12. References:

1. RFC 3261 - SIP: Session Initiation Protocol
2. RFC 3903 - Session Initiation Protocol (SIP) Extension for Event State Publication
3. A real-world system for human motion detection and tracking – David Moore, California Institute of Technology, June 2003.
4. Object Tracking: Survey – ACM Computing Surveys, Vol 38, No.4 Article 13, December 2006
5. C. Tomasi and T. Kanade, “Detection and tracking of point features,” Tech. Rep. CMUCS-91-132, Carnegie Mellon University, 1991.
6. S. Birchfield, “Derivation of Kanade-Lucas-Tomasi tracking equation.” http://robotics.stanford.edu/~birch/klt/derivation.ps, 1997.
7. J. Shi and C. Tomasi, “Good features to track,” Proc. IEEE Conf. Comput. Vision and Pattern Recogn., pp. 593–600, 1994.
8. S. Birchfield, “KLT: An implementation of the Kanade-Lucas-Tomasi feature tracker.” URL:http://robotics.stanford.edu/~birch/klt/, 1998
9. “Intel open source computer vision library.” http://www.intel.com/research/mrl/research/opencv/.
10. Sofia SIP Library – Nokia . http://sofia-sip.sourceforge.net/
11. RFC 4480 : RPID: Rich Presence Extensions to the Presence Information Data Format (PIDF) http://www.rfc-editor.org/rfc/rfc4480.txt
12. C. Stauffer, W.E.L. Grimson, “Adaptive background mixture modelsfor real-time tracking”, Proc. of CVPR 1999, pp. 246-252.

13. Digital Image Processing (3rd Edition) - Rafael C. Gonzalez, Richard E. Woods. (Authors)

13. <u>Acknowledgements</u>

I wish to thank Professor Henning Schulzrinne for guiding me throughout the project, helping me whenever I was stuck and providing me with valuable tips whenever required. Also I wish to thanks Vishal Singh who helped me with using his SIP Library. Though I finally did not end up using the library, yet I am grateful to him for his time and patience.