Final Report on Video-based Tracking

Erhan Bas, Deniz Erdogmus

From October 2007 to September 2008, we have investigated the use of regular perspective cameras as well as fisheye-lens cameras (mounted on the ceiling) to track multiperson activity in a room. The main purpose of our investigation had been to identify what combination of relatively cheap cameras to use, as well as to develop computationally efficient algorithms that could identify and track multiple (known) people in a given interior space using multiple cameras using such algorithms. Consequently, while our algorithm selections have been primarily motivated by the real-time executability constraint given computational restrictions of our desktop computers, we made every effort to engineer a software package that works together effectively in order to achieve high performance and accuracy for identification and tracking in the context of unobtrusive monitoring. To this end, we have experimented with a variety of candidate techniques for various parts of the solution; however, in this report, we focus on the components that were finally deemed to be the best choices given our experimental results obtained from various recordings at different rooms including our OGI-Central office and POCL at CHH. For the real time implementation of the developed algorithms, programs available in the OpenCV libraries are utilized when possible and custom additions and modifications are made at will based on performance and speed limitations of our particular application. To handle 2 cameras, OpenCV's *cvcam* library, which processes 2 simultaneous camera views consecutively, is employed. Our computer was also limited to 2 camera inputs by hardware constraints. Therefore, at this stage, the mentioned solution has been sufficient; however, further software engineering is necessary to handle more than 2 cameras simultaneously. For each camera, functions 1-3 (detailed below) are completed separately. Function 5 is recently included to merge information from multiple cameras and convert tracking results to 3D real-world coordinates. Function 4 (particle filter) is expected to be an important component for future improvements where tracking of more complicated bodily state variables (such as gait parameters or postures etc) will be desired. For tracking of position alone, in the absence of a good dynamic model, this function does not really add much to performance while it creates computation overhead; the reason for this is that position states are directly measurable from the images and for dynamic state estimation to be most useful an accurate dynamic model is essential (that is the acceleration model for the maneuvers of the bodies being tracked). Nevertheless, this component would still be useful in smoothing impulsive noise patterns that might arise from the image processing portions of the overall solution that could especially occur during occlusions or sudden lighting changes in the environment. A model that is less accurate than the measurement itself serves only to distort estimates, while in a scenario where unmeasured states will be sought, the utility of a dynamical model, even if not perfect, will help improve performance.

Function 1: Background Subtraction

The purpose of background subtraction (BGS) is to segment each frame of the captured video into two clusters: background regions/objects and foreground (e.g. moving people or displaced furniture) objects. In order to prevent displaced furniture and other changes in background to be perceived permanently as a moving object, the BGS module is adaptive; that is, anything that does not move within a preset period of time will slowly be merged into the background model for those pixels. Problems associated with this will be discussed below after the basic algorithm description.

An adaptive Gaussian Mixture Model (GMM) based background subtraction method [1] is combined with a shadow elimination algorithm [2]. These algorithms employ Red-Green-Blue (RGB) and Hue-Saturation-Value (HSV) coordinate systems. GMs are created for each channel of RGB frames to form background model. Background model has the same size and depth as the grabbed images. The *background image* consists of the selection of most probable (largest weighted) RGB values for each pixel in the background model; i.e. it is the maximum likelihood image of the background scene. For a more detailed explanation of BGS, please see our paper published in EMBC 2008 (appended to this report for convenience) [3]. Foreground regions are obtained by subtracting the grabbed image from the background image. Obtained difference vector (in RGB) for each pixel is weighted by (currently) manually selected multipliers (linearly projected to one dimension that best suppresses noise across the whole image in a given scene under certain lightening conditions).[1] Selection of these multipliers is scene dependent and needs to be considered in conjunction with the difference threshold. Pixels exceeding the threshold value are further checked to decide whether they are shadow regions or not. Different shadow elimination algorithms are available in the literature; the one that we used is similar the proposition of Prati et al. [2], where pixel values are mapped to the HSV domain from the RGB domain, and ratios of H-S-V values are compared for both the background image and the grabbed frame. This whole procedure can be summarized as follows:

Let r-g-b, R-G-B, h-s-v, and H-S-V be the image vectors for grabbed image, background image, and their corresponding values in HSV space respectively. **W** is the linear projection vector for the RGB deviation from background. *Thr* is the threshold for this projection to be considered as background or foreground. Alpha, betha, t_s, and t_h are manually selected shadow elimination thresholds. Then for each pixel;

```
if | W*( r-g-b – R-G-B ) | > Thr
   if alpha<=v/V<=beta & s-S<=t_s & h-H<=t_h
   shadow(background)
else
   foreground
else
   background
end
```

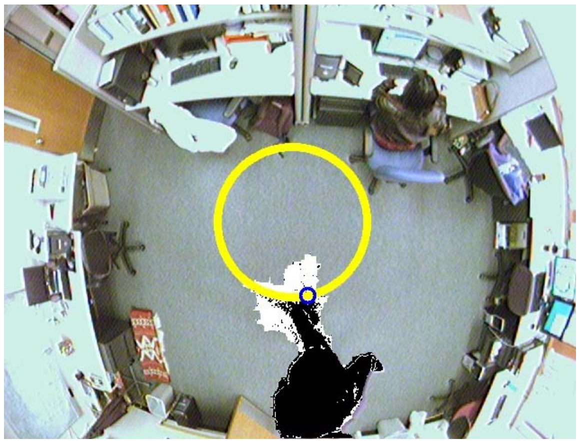
Figure 1. Foreground detection with shadow elimination. Initial foreground regions (black and white) are segmented into shadow (white) and foreground (black).

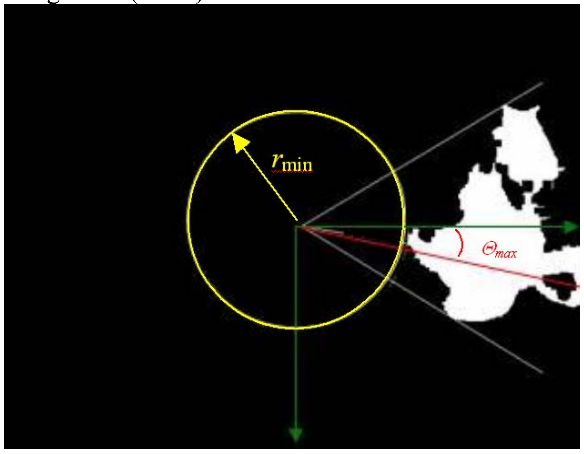

Figure 2. Estimation of foot polar coordinates from the foreground cluster pixels for a human that recently moved a chair.

[1] A learning algorithm that automatically calculates these weights and the corresponding optimal threshold value locally would be a better solution for this calculation. For this local noise models and dispersion of fish-eye cameras need to be calculated.

Once each pixel is classifier using the logic above, foreground objects are detected and labeled using connected component analysis based on the foreground image obtained. A sample of foreground detection with shadow elimination result can be seen in Figure 1.

Function 2: Object Detection

During the course of the past year, the decision to employ fisheye lenses on cameras mounted on the ceiling looking vertically down was made. This selection was motivated by two advantages: wide view angle (almost a complete hemisphere) and reduced possibility of occlusions when multiple people are tracked using multiple cameras. With the restriction that the fisheye cameras are positioned on the ceiling mounted vertically downwards, simplifies the nonlinear mapping of the environment to the image plane: the floor plane is warped onto the surface of a sphere using radial projections resulting essentially in a 2-dimensional polar coordinate system in the image plane that represents the floor plane in a natural manner (see Figure 1). As a consequence of this geometry, a standing human will be projected as a radially extending foreground object where the feet will be located at the minimum radius pixels of this cluster. For a given foreground cluster (assuming this consists of only one person in a nonocclusion situation), the pixel locations are expressed in polar coordinates (black pixels in Figure 1). The mode of the angle distribution is assumed to be the orientation estimate (Figure 2). Based on the histogram of the radius, r, smallest radius value which exceeds 10% of the peak histogram value, is selected as the object displacement from the center (Figure 2).

Function 3: Occlusion Elimination

As in all surveillance systems, occlusions must be considered since multiple moving objects and humans will be typically present in the scene. To resolve the problem of separating a foreground cluster into the appropriate number of tracked objects, the watershed algorithm [4], a non-parametric marker-based segmentation algorithm, is used. The watershed algorithm can also be understood as a connected component clustering approach that utilizes features of choice e.g. color or texture). Seed points are taken from the line that passes through the previous frame's foreground objects where occlusion did not yet occur (red line in Figure 2). Linear Discriminant Analysis (LDA) is then employed to separate pixels into two object segments in the frame that presents the occlusion. For subsequent frames previous occlusion resolving solutions can be used to initialize the seeds and for more than two objects occluding, the LDA algorithm can be employed multiple times. At this point, we have to mention a crucial issue: if the frame rate is not high enough, the seed points from the previous frame will not be useful for the current image. The assumption behind our algorithm described above is that for a fast capture rate, the objects will overlap with their previous locations significantly (more than 50%) in consecutive frames. Although the occlusion elimination algorithm works as a support function for the whole process, it has a direct effect on the performance of the whole system. It is particularly crucial to keep track of all identities and tracked objects through phases of occlusion. This task is made relatively easier when multiple fisheye cameras are employed, since with proper positioning of these cameras, it can be made impossible to create an irresolvable occlusion situation in all cameras simultaneously (e.g. a small child can completely disappear behind an adult for a few frames in at most one camera). As a sanity check, if a segmented region is too small at the end of watershed-LDA segmentation, then input region is passed as the output.

Another issue is the number of objects present in the scene. Currently, the algorithm decides whether an occlusion occurred or not by comparing the number of the detected objects to the previously tracked ones. So if an object leaves the room the algorithm currently decides that there is an occlusion (and tries to split one foreground cluster into two to make up for the missing object). In the future, this problem will be automatically resolved as the information of total number of objects will be known to the algorithm by keeping track of identities of people entering and leaving the room. A separate perspective camera facing the door is to detect, identify, and count the number of people in the room.[2]

Function 4: Partical Filter

A basic position and velocity tracking particle filter model is designed using an adaptive linear autoregressive model of accelerations in the image plane. The position estimate (r,θ) obtained from each camera view is utilized as the measurement equation of a dynamic state-model of the moving object to be tracked with a particle filter. The particle filter technique is a sequential Monte Carlo approach to recursive Bayesian state estimation in dynamic systems and its computationally convenient and efficient algorithms apply to linear and nonlinear dynamic systems corrupted by Gaussian and non-Gaussian noise distributions [5]. For more detailed explanation on partical filter and our noise model refer to our EMBC 2008 paper [3]. While we have developed experimental data derived noise models and incorporated this in the position tracking algorithms under this framework in the 2-dimensional image-plane tracking paradigm, we have not extensively studied the use of particle filters for tracking position in the 3-dimensional actual room coordinates. The reasons for this are indicated earlier in this report; primarily, a dynamic state estimation paradigm is useful if its dynamical model contributes additional and/or more accurate information to the estimation problem. In the case of tracking position alone, without an accurate acceleration model, the dynamic component might hurt performance rather than helping. In future work where more states will be tracked, the use of particle filters will become essential to resolve multiple inverse solutions as well as to impose temporal smoothness of state trajectories.

Function 5: Multiple Person Tracking with Multiple Fisheye Cameras

In the last quarter, we focused on the use of 2 fisheye cameras mounted on the ceiling as described and developed algorithms that can track multiple people (implementation hard-coded to handle a maximum of 10 people/objects) in the given interior environment. Functions 1 to 3 are employed on each image sequence separately and a nonlinear mapping of object positions from the image planes to real world coordinates (via a calibrated projection model) is implemented. Consequently, the current system can handle two cameras that can track individually up to ten people while resolving occlusions in each camera recording independently, as much as possible. The two estimates from the cameras for each person can then be mapped to the 3D room coordinates and fused. At this time, the occlusion resolving and camera confidence levels in their position estimates are not optimally designed and the short term future work in this respect should be to achieve information exchange between cameras to obtain jointly a better occlusion resolving solution and also achieve optimal estimation fusion in the room coordinates based on relative camera confidences.

Spherical Fisheye Camera Model

[2] A better approach is to use previous frames. Velocity and object trajectories are some good features to predict or to decide occlusion. We have started working along this direction, but did not incorporate these ideas in the current code. These ideas have been tested successfully in hand/face tracking for medication adherence assessment.

Calibration of the transformation between a fisheye camera image plane and real world lateral coordinates of an object is implemented in Matlab and the optimization toolbox of Matlab is employed to find the best fit to the model parameters using a snapshot of a grid of points (we unroll a 6’x12’ vinyl mat on the floor and take snapshots with both fisheye cameras that are mounted on the ceiling such that their image plane is parallel to the floor). There are five variables to be calculated: camera orientation (α), camera height (h), camera image plane radius(r), and camera center coordinates in room floor plane (Sx, Sy). Figure 3 visualize the experimental setup. Calibrations of fisheye cameras are calculated based on the geometrical relationship of the camera with the room and the main task is to find the transformation of mapping a pixel location on the image plane to its corresponding 3-D location on the floor. This mapping can be viewed as 3 consecutive operations.

1) Scaling from image plane onto camera plane,
2) Rotation around camera normal, and
3) Translation on the floor.

We assumed that scaling of object dimensions are same in both x and y directions in the image plane. A pixel location on the image, (i,j), can be rewritten on the camera plane as:

$$\begin{bmatrix} i \\ j \end{bmatrix} \rightarrow \begin{bmatrix} ai \\ aj \end{bmatrix} = \begin{bmatrix} x_c \\ y_c \end{bmatrix} \rightarrow u = \begin{bmatrix} x_c \\ y_c \\ \left(r^2 - x_c{}^2 - y_c{}^2\right)^{1/2} \end{bmatrix} \tag{1}$$

where a is scaling factor, (x_c,y_c) is the object location in camera plane, and r is the distance from the origin to a given object location P. Position of the object location on the floor with respect to the camera plane is given in spherical coordinates as

$$q_c = \begin{bmatrix} r\sin\phi_c \sin\theta_c \\ r\sin\phi_c \cos\theta_c \\ r\cos\phi_c \end{bmatrix} \tag{2}$$

where ϕ_c is the angle between the positive z-axis and the object location, and θ_c is the angle that defines the position of the object on the camera plane. So the transformation from 2-D space to 3-D space can be written as

$$R_{ec}q_c + \begin{bmatrix} S_x \\ S_y \\ S_z \end{bmatrix} = \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} \tag{3}$$

Here $S = [S_x\ S_y\ S_z]^T$ and $P = [P_x\ P_y\ P_z]^T$ are camera and object location on the floor, and R_{ec} is the rotation matrix that represents the misalignment between camera plane and 3-D world coordinate system (floor).

Displacement between object location on the floor to camera position can be written as

$$r = \frac{h\|R_{ec}u\|}{[0,0,1]R_{ec}u} = \frac{hr_0}{[0,0,1]R_{ec}u} \tag{4}$$

We assumed that camera normal is orthogonal to floor (parallel to gravity), so the rotation matrix, R_{ec}, and the displacement r can be written as

$$R_{ec} = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}, r = \frac{hr_0}{\left(r_0^2 - i^2 - j^2\right)^{1/2}} \tag{5}$$

By substituting (2) and (5) to (3), the real world coordinates corresponding to a pixel and an object on the floor (that is imaged at this pixel) can be given as

$$P = \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} = \begin{bmatrix} S_x \\ S_y \\ -h \end{bmatrix} + \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \frac{hr_0}{\left(r_0^2 - i^2 - j^2\right)^{1/2}} \begin{bmatrix} \left(x_c{}^2 + y_c{}^2\right)^{1/2} r_0^{-1} x_c \left(x_c{}^2 + y_c{}^2\right)^{-1/2} \\ \left(x_c{}^2 + y_c{}^2\right)^{1/2} r_0^{-1} y_c \left(x_c{}^2 + y_c{}^2\right)^{-1/2} \\ \left(r_0{}^2 - x_c{}^2 - y_c{}^2\right)^{1/2} r_0{}^{-1} \end{bmatrix}$$

If we simplify this equation

$$\begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} = \begin{bmatrix} S_x \\ S_y \\ -h \end{bmatrix} + \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} ai \\ aj \\ \left(r_0 - a^2 i^2 - a^2 j^2\right)^{1/2} \end{bmatrix} h\left(r_0 - a^2 i^2 - a^2 j^2\right)^{-1/2}$$

Which also equals to

$$\begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} = \begin{bmatrix} S_x \\ S_y \\ -h \end{bmatrix} + \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ \left(\bar{r}_0 - i^2 - j^2\right)^{1/2} \end{bmatrix} h\left(\bar{r}_0 - i^2 - j^2\right)^{-1/2}$$

Letting $P_z = 0$ will result in

$$\begin{bmatrix} P_x \\ P_y \end{bmatrix} = \begin{bmatrix} S_x \\ S_y \end{bmatrix} + \begin{bmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} h\left(\bar{r}_0^2 - i^2 - j^2\right)^{-1/2}$$

or

$$\begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix} \left(\begin{bmatrix} P_x \\ P_y \end{bmatrix} - \begin{bmatrix} S_x \\ S_y \end{bmatrix} \right) h^{-1}\left(\bar{r}_0 - i^2 - j^2\right)^{1/2}$$

Thus we can summarize the filter measurements

$$\begin{bmatrix} i \\ j \end{bmatrix} \rightarrow \begin{bmatrix} \rho \\ \gamma \end{bmatrix} = \begin{bmatrix} \left(i^2 + j^2\right)^{1/2} \\ \arctan(j/i) \end{bmatrix}$$

$$\rho = \|P - S\| \bar{h} \left(\bar{r}_0^2 - i^2 - j^2\right)^{1/2} + n_\rho$$

$$\gamma = a\tan\left(\frac{P_y - S_y}{P_x - S_x}\right) - \alpha + n_\gamma$$

$$\rho^2 = \|P - S\|^2 \bar{h}^2 \left(\bar{r}_0^2 - \rho^2\right)$$

$$\rho = \frac{\|P - S\| \bar{h} \bar{r}_0}{\left(1 + \|P - S\|^2 \bar{h}^2\right)^{1/2}}$$

where (ρ, γ) is the pixel value in polar coordinates, n_ρ is the observation noise for radius, and n_γ is the noise term for angle, and $\bar{h} = h^{-1}$.

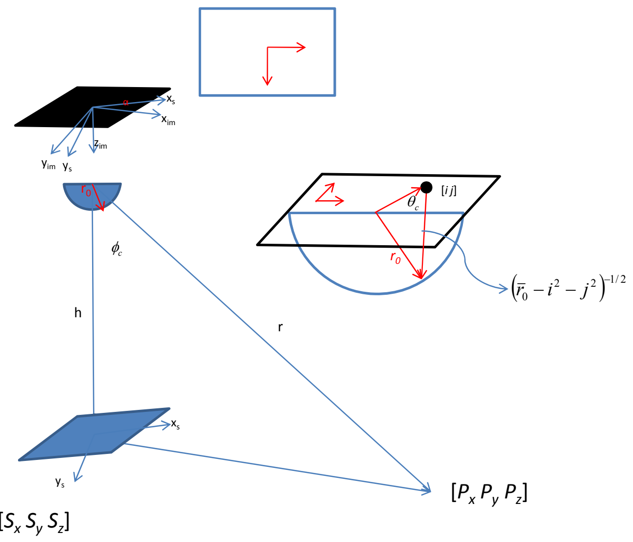

Figure 3.Illustration of the coordinate systems involved in developing the calibration model that is assumed in the implementation.

References

[1] C. Stauffer, W.E.L. Grimson, “Adaptive Background Mixture Models For Real-time Tracking”, Proceedings of CVPR, pp. 246-252, 1999.
[2] Prati, et al., “Detecting Moving Shadows: Algorithms and Evaluation,” IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 25, pp. 918-923, 2003.
[3] E. Bas, U. Ozertem, D. Erdogmus, M. Pavel, “Towards Fish-Eye Camera Based In-Home Activity Assessment”, EMBC 2008.
[4] Meyer, F. “Color image segmentation”. In Proceedings of the International Conference on Image Processing and its Applications, pages 303—306, 1992.
[5] A. Doucet, N. de Freitas, N. Gordon (eds), “*Sequential Monte Carlo Methods in Practice”*, Springer-Verlag, 2001

Multiple Person Tracking

This is a brief description and explanation of the demo's code. The demo constitutes of one main **intel.cpp** file and three additional header files **backg.h, multicam.h, my_aux.h**. The main part, intel.cpp, includes parameter settings and function calls. Functions are patched to the demo as header files and divided into three main categories. 1) functions.h, 2) my_aux.h, and 3)backg.h. Camera calibration parameters are computed in the Matlab environment and used in the main part of the C++ code.

When demo.cpp is executed, after initializations in the main part, the program will call the multicam function. Multicam is an Opencv library function that processes consecutive operations simultaneously for both camera views. The order of categories in this multicam function for each camera view are:

1) Image processing function **back** (in backg.h): This function, executes consecutive image processing functions to calculate the necessary information, e.g. object position. A more detailed explanation of this function is in the following section.

2) Object - Observation assignment: To track objects, objects locations from the previous frame are held and they are compared with the newly observed ones to keep track of object trajectories. This operation is under subfunction **multicam**.

3) Based on the tracking results calculated object locations are back projected to 3D world coordinates and plotted at the end of the subfunction **multicam**. To calculate 3D position, we used the Matlab optimization toolbox to for the optimal fitting of the model with the following parameters: camera orientation (α), camera height (h), camera image plane radius(r), and camera center coordinates in room floor plane (Sx, Sy). These optimal parameters are currently manually entered into the C++ code.

Descriptions of important functions

Intel.cpp

Main part of the demo. It calls the multicam function which is located in the cvcam library. This is an Opencv library that is built for multicam operations. Parallel computation of image processing operations is held for both cameras. Before processing any images, we set camera parameters in this part of the demo. We have 5 parameters for each camera settings. For camera1 (and then similarly for camera2):

```
cam1params.beta = 0.95;
cam1params.alpha = 0.75;
cam1params.t_s = 0.35;
cam1params.t_h = 100;
cam1params.pixelTHR = 25;
```

First 4 parameters are used for shadow detection, and the fifth one is the pixel threshold that decides whether a cluster of pixels belongs to foreground or background (to omit small misdetections).

Another important function under this file is:

```
bg_model1 = cvCreateGaussianBGModel( frame,0,&cam1params );
```

This creates the background model for the first camera. It gets an initial image(frame), BG parameters and cam1params as inputs. BG parameters are set with default values, and cam1params are set as above. This function is under my_aux.h. Here are the default values for background model parameters and camera parameters:

```
#define CV_BGFG_MOG_BACKGROUND_THRESHOLD 0.7 /* threshold sum of weights for
background test */
#define CV_BGFG_MOG_STD_THRESHOLD 2.5 /* lambda=2.5 is 99% */
#define CV_BGFG_MOG_WINDOW_SIZE 2500 /* Learning rate; alpha = 1/CV_GBG_WINDOW_SIZE */
#define CV_BGFG_MOG_NGAUSSIANS 3 /* = K = number of Gaussians in mixture */
#define CV_BGFG_MOG_WEIGHT_INIT 0.05 /* Initial weight for new GMM components */
#define CV_BGFG_MOG_SIGMA_INIT 30 /* Initial variance for new GMM components */
#define CV_BGFG_MOG_MINAREA 500.f /* Area criteria(we don't use this value) */
#define CV_BGFG_PIXEL_THRESHOLD 25 /* default value for pixel thr */
/* camera parameters */
#define CV_BGFG_SHADOW_ALPHA 0.75 /* alpha*/
#define CV_BGFG_SHADOW_BETA 0.95 /* beta */
#define CV_BGFG_SHADOW_T_S 0.35 /* saturation threshold */
#define CV_BGFG_SHADOW_T_H 100 /* hue threshold */
#define CV_BGFG_MOG_NCOLORS 3 /* channel number */
```

For BG model parameters and their usage refer to original paper **[1]** or our EMBC paper[2]. Multicam function calls the **back** function under backg.h (all image processing functions are called in this backg function).

Backg.H

There are some important issues and functions in this file that have to be taken into consideration. First, Opencv has its own object formats. For example, images are represented (and created) as IplImage. The best way to adopt notation is to use OpenCv's own documentation. Second, there are some important functions in this backg.cpp

List of important functions:

```
cvCreateStructuringElementEx( 3, 3, 0, 0,CV_SHAPE_CROSS, NULL );
```

This function creates morphological operators. Different types of operators can be used in this function(I used `CV_SHAPE_CROSS`)

```
cvUpdateBGStatModel( image, bg_model );
```

This function updates the current background model(BGM) with the current pixel value. This function is under my_aux.h, and for usage refer to [1] and [2].

As mentioned images are in IplImage format, and bg_model is the background model that is created previously. It is important to know bg_model's content. Most important objects in bg_model are `bg_model->foreground`, `bg_model->background`, and `bg_model->foreground_regions`. Both `bg_model->foreground`, `bg_model->background` are IplImages that have the same size with the input images. `bg_model->foreground` is a binary image and is computed in the `cvUpdateBGStatModel` function. `bg_model->background` has the same dimension as the input image. `bg_model->foreground_regions` is a sequence and keeps the segmented object clusters' locations.

```
cvFindContours(bg_model->foreground, bg_model->storage,
&first_seq, sizeof(CvContour), CV_RETR_EXTERNAL,
CV_CHAIN_APPROX_SIMPLE );
```

This function finds the contours in a binary image, where more details can be found in Opencv documentation.

```
procesObj = occlusionHandler(image,bg_model,objectLocations,
&num_of_objects_ex,VV);flag = 1;
```

This part segments occluded objects into 2 clusters; it uses the watershed algorithm and LDA to decide the boundary between occluded objects.

```
rectangle(bg_model,image,procesObj,objectLocations,objectSizes,f
irst_seq, VV);
```

This function calculates the object location, object size, orientation, etc. But currently we are using only the location of an object as an output. This function is called as many times as the number of clusters in `bg_model->foreground_regions.` Outputs of this function are processObj and objectLocations.

ObjectLocations[0*processObj] : radius
ObjectLocations[2*processObj] : angle

Refer to [2] for the calculation of object location. We calculate the camera calibration parameters by using optimization toolbox in Matlab. There are 2 m-files for registering two image views. Register.m and myfun.m.

register.m

In order to register two images first we select some points from one view and match them with the corresponding ones in the other view and try to find a transformation that maps each point to the corresponding one on the other. To select coordinates we used the cpselect tool from Matlab. We call 2 views with this function and select pixel coordinates. For example, the following code is used to calculate the camera calibration using 64 points for each views.

```
cam2 = (imread(strcat(direc,'Cam2_00040.jpg')));
cam2=mat2gray(cam2);
cam1 = (imread(strcat(direc,'Cam1_00038.jpg')));
cam1 = mat2gray(cam1);
[cam1_points,cam2_points] = cpselect(cam1,cam2,'Wait',true);
% [[cam1_points,cam2_points] =  cpselect(cam1,cam2,cam1_points,cam2_points, 'Wait',true);
% load registeredPoints.mat
```

The excluded part adds or adjusts the previously selected and stored pixel locations. Or you can directly load a previously saved mat file. Then the world coordinate that matches with this selection is created.

```
n = 64; % number of points used
l = [1:n]';
ly =  (floor((n-l)/8)-4);
lx = rem((n-l),8)-4;
D = [lx(:) ly(:)];
wordVal = D.*repmat([4.5 4.5],n,1);
```

Our selection was an 8x8 matrix. So we created a matrix with the same dimensions. The distance between each vinyl dots on the ground is around 4.5 inches. Then we change the pixel origin from left upper corner to image center. And flip the image (upside down), so that +y will be downward of the image whereas +x will be towards right. (This is just the convention we used in [2], any direction as long as consistent with the same convention will result in same result.)

```
CAM1 = [];
CAM2 = [];
wordCenter = 28;
CAM1 = cam1_points;
CAM2 = cam2_points;
% CAM1(:,2) = 480 - CAM1(:,2);
% CAM2(:,2) = 480 - CAM2(:,2);
pixelVal1 = (CAM1-repmat([320 240], length(CAM1),1));
pixelVal1(:,2) = -pixelVal1(:,2);
pixelVal2 = (CAM2-repmat([320 240],length(CAM1),1));
pixelVal2(:,2) = -pixelVal2(:,2);
```

Finally, fminsearch calculates the set of parameters that minimizes the overall error. We have 5 parameters:

```
%% Defination for parameters
% x is the unknown variables
% pixelVal = [x y]
% x(1) = alpha   % direction of camera (angle between the camera normal and z axis)
% x(2) = 1/h    % h is the height of the camera
% x(3) = r0^2    % camera image plane radius. It has to be more than 320^2 + 240^2
% x(4) = Sx    % Position of (arbitrary)selected center on the word coordinate
% x(5) = Sy
% Initialization
% x(1) ~ 0
% x(2) ~ 1/100 (height ~ 100 inch)
% x(3) ~ 400 : [640 480] -> R = 800, radius of the image [320 240] ->400
% x(4) ~ Cam1 : -4 * 12, Cam2 : 5 * 12
% x(5) ~ Cam1 : -3 * 11, Cam2 : 1 * 11
% clc
%%
angle1 = 0;
angle1 = 2*pi/360*angle1;
x1 = fminsearch(@(x) myfun(x, pixelVal1(1:2:end,:), wordVal(1:2:end,:)),[angle1 1/110 400 -22.5 -13.5])    %x=5 y=3
angle2 = 0;
angle2 = 2*pi/360*angle2;
x2 = fminsearch(@(x) myfun(x, pixelVal2(1:2:end,:),wordVal(1:2:end,:)),[angle2 1/130 500 22.5 -27])      %x=-5 y=6

function f2 = myfun(x,pixelVal,P)
% x is the unknown variables
% pixelVal = [x y]
% x(1) = alpha
% x(2) = 1/h
% x(3) = r0
% x(4) = Sx
% x(5) = Sy
Px = P(:,1);
Py = P(:,2);
i = pixelVal(:,1);
j = pixelVal(:,2);
% fx1 = (i - (cos(x(1))*(Px - x(4)) - sin(x(1))*(Py-x(5))).*x(2).*sqrt(x(3).^2-i.^2-j.^2)).^2;
% fy1 = (j - (sin(x(1))*(Px - x(4)) + cos(x(1))*(Py-x(5))).*x(2).*sqrt(x(3).^2-i.^2-j.^2)).^2;
% f1 = sum(fx1.^2 + fy1.^2);
% fx2 = (Px - ( x(4) + (cos(x(1))*i+sin(x(1))*j) ./ (x(2)*sqrt(x(3)^2 - i.^2 - j.^2)) ) ).^2;
% fy2 = (Py - ( x(5) + (-sin(x(1))*i+cos(x(1))*j) ./ (x(2)*sqrt(x(3)^2 - i.^2 - j.^2)) ) ).^2;
fx2 = abs(Px - ( x(4) + (cos(x(1))*i+sin(x(1))*j) ./ (x(2)*sqrt(x(3)^2 - i.^2 - j.^2)) ) );
fy2 = abs(Py - ( x(5) + (-sin(x(1))*i+cos(x(1))*j) ./ (x(2)*sqrt(x(3)^2 - i.^2 - j.^2)) ) );
f2 = sum(fx2 + fy2);
end
```

About x(4) and x(5): Selection of arbitrary center means just pick a coordinate both on 2 view and write an estimate position for this coordinate for each view. For example in the example (which is the setup in POCLat CHH) 28th pixel location is selected to make it simple (center of world coordinate grid). Their distance to first camera is Sx = -22.5 Sy = -13.5. These lengths are in inches and are just rough estimates. If fminsearch converges to the global optimum we will get better estimates for these numbers eventually. We have observed that the fitting criterion might have some local minima, therefore, proper initialization of the parameters (as close as possible to actual physical values) is helpful to improve optimization performance. In implementation, we simply initialized the rotation angle to 0 in demo.cpp, since cameras are aligned and camera plane is parallel to floor.

As a last comment myfun is the mapping function that minimizes the cost function. Mean absolute error is used as the cost function.

[1]C. Stauffer, W.E.L. Grimson, "Adaptive Background Mixture Models For Real-time Tracking", Proceedings of CVPR, pp. 246-252, 1999.
[2] E. Bas, U. Ozertem, D. Erdogmus, M. Pavel, "Towards Fish-Eye Camera Based In-Home Activity Assessment", EMBC 2008

Face Detection and Tracking

In addition to tracking of people we previously implemented face detection and recognition system that uses Haar object classifiers to detect faces. The training part is included into the code so that it can be updated with test images. Detected faces are mapped to a lower dimension with PCA + LDA and classification is based on the output of the LDA algorithm.

There are 3 files for 2 applications. First application:

faceDetect.cpp

It is used for capturing training images and saving modified images under a directory that will be used for testing. It uses Haar classifiers and finds face regions. Calculated regions are interpolated to 240x240 images, and saved to train directory under the project directory. There are no special parameter to be set for this part. Possible modifications to this function:

1) Size of the training images(currently 240x240)
2) Haar classifier type: You can select side or frontal view. In our case frontal view is used since our camera is facing the door. For possible selections, refer to opencv references.
3) Default directory locations: Directory for the Haar classifier.
4) This code can be used for both camera or avi files. To select between these, simply comment out

```
capture = cvCaptureFromCAM( 1 );
```

so that the program automatically tries to grab an avi file.

The second application is face classification (face_track.cpp and functions.h).

Face_track.cpp

This part is a combination of training + testing. Before running this part you need to manually select the training images (under the directory that previously set in faceDetect.cpp) and set their name in train%2d format (train01.jpg, train02.jpg...). Here is a short description for each part.

face_track: Main part. First it trains the images under ..../'projectname'/train/ directory, then in testing; it finds faces and applies PCA + LDA. Lastly, program classifies the faces based on the distance score of each detected face with the training set.

functions.h

All math (PCA, LDA, scoring, etc) calculations and most parameters are under functions.h.

Parameters:

```
const double eigenThr = 0.85;
CvSize faceSize = cvSize(240, 240);
const int numOfTrainNumber = 21;
const int nEigens = numOfTrainNumber-1;
const int numOfClasses = 3;
const bool histEq = true;
int numOfTrainperClass = numOfTrainNumber/numOfClasses;
```

eigenThr : threshold to cutoff the number of eigenvectors(values). (in PCA)
faceSize : final size of the training face images(same as in faceDetect.cpp)
numOfTrainNumber : number of training images used in training (total number)
numOfClasses : number of persons that are used in training
histEq : To select histogram equalization (or to disable it)

numberOfTrainNumber and numOfTrainPerClass are calculated from other parameters.

Under face_track.cpp there is also

```
bool train = true;
```

This boolean parameter is used to adjust training images. Default : bool train = false;

In the training part, while labeling faces, one thing is important. Let’s say you have 30 images for 3 people so label will be train01.jpg - train10.jpg for person 1, then 11 to 20 for person 2, and 21 to 30 for person 3. Algorithm will read the first **numOfTrainNumber** images and label them based on the **numOfClasses.**

Directories:
.../'project name'/train/ : training images.
Where you manually rename, and select training face images.
.../'project name'/out/ : out images will be saved here.

One needs to define the names of each class (person) in functions.h under outputResults function(the last function). For example for the above example, let’s say first 10 images are Erhan’s, later 10 are Deniz's, and last 10 are Misha's then code will be

```
...
  case 0:
    sprintf(nameOfThePerson, "Erhan");
    scalar = cvScalar(0,0,255);
    break;
  case 1:
    sprintf(nameOfThePerson, "Deniz");
    scalar = cvScalar(0,0,255);
    break;
  case 2:
    sprintf(nameOfThePerson, "Misha");
```

```
        scalar = cvScalar(0,0,255);
        break;
    case 3:
        sprintf(nameOfThePerson, "Other");
        scalar = cvScalar(0,0,0);
        break;
...
```

Other is decided based on the distance score. Distance score is highly dependent on the environmental conditions such as lighting (since eigenvalues depend on lightning). Because of this, threshold for rejection needs to be decided manually in LDAtest function to use this option. For simplicity this option commented out.

```
//if(min>150000000)  res=numOfClasses; //numOfClasses = others
```