This section contains the details of the face detector functionality that comes with CMUcam3. The first section outlines the general face detection algorithm while the next section is more CMUcam3 specific, dealing with various implementation issues and code description.

1 Face Detection System

1.1 Overview

The Face Detector implementation provided with the CMUcam3 is based on the well-known paper "Robust Real-Time Face Detection" by P. Viola and M. Jones, accepted at International Journal of Computer Vision 2004 [1]. Earlier version of the same paper was submitted to ICCV 2001.

The paper introduces a novel technique to detect faces in real-time and with very high detection rate. It is essentially a feature-based approach in which a classifier is trained for Haar-like rectangular features [6] selected by AdaBoost. The test image is scanned at different scales and positions using a rectangular window, and the regions which pass the classifier are declared as faces. One of the major contributions of this paper is the extremely rapid computation of these features using the concept of Integral Image, which enables the detection in real-time. Additionally, instead of learning a single classifier and computing all the features for all the scanning windows in the image, a number of classifiers are learnt which are put together in a series to form a cascade. The classifiers in the beginning of the cascade are simpler and consist of smaller numbers of features. However, as one proceeds in the cascade, the classifiers become more complex. A region is reported as detection only if it passes all the classifier stages in the cascade. If it is rejected at any stage, it is discarded and not processed further. This way, the easier patches in the image which the "cascade of classifiers" is sure of not being a face, are rejected very early while the difficult regions are operated on by more complex classifiers. This greatly speeds up the detection process without compromising on the accuracy and provides high detection rate. This overall system provides performance comparable to the existing best face detector systems (Rowley et al., 1998 [2]; Schneiderman and Kanade, 2000 [3]; Roth at al., 2000 [4]) but with orders of magnitudes faster than any of these systems. On a conventional desktop, it can detect faces at 15 frames per second.

In the next sections, the whole detection system is described in brief. Section 1.2 introduces the features used in detection, section 1.3 talks about basic method to learn a classifier using AdaBoost while section 1.4 describes the extension to cascade.

For more detailed information, kindly consult the paper [1] which is available freely on the web.

1.2 Features:
The features used for face detection are simple Haar-like rectangular features as shown in Fig 1. Three versions of these features are used in the paper: two *two-rectangle* features, and one *three-rectangle* feature and *four-rectangle* feature each. The value of these features is the difference of the sum of the pixels lying in the white and the gray regions.

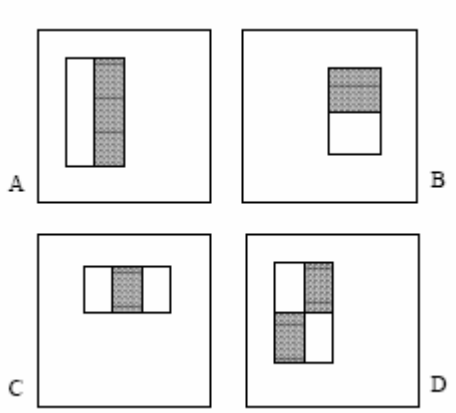

Fig 1. (A) and (B) are the two Two-Rectangle features, (C) shows the Three-Rectangle feature and (D) the Four-Rectangle feature. (Source: IJCV'04 Viola-Jones)

For a base window of 24x24, the exhaustive set of all such possible features is about 160,000. Obviously, a classifier shouldn't be trained on such a large number of features, for two reasons. One, it'll render the system incapable of processing images in real-time, atleast not with today's conventional desktops. Secondly, the set of rectangular features is overcomplete many times over, hence a lot of them are simply redundant. The paper puts forth the hypothesis that it is possible to select a smaller number of "good" features than can be fewer enough to retain real-time functionality but at the same time, discriminative enough to detect faces with high accuracy. The next section describes how to select these features.

A major advantage of using these rectangular features is that they can be computed very quickly using the concept of integral image. The value in the integral image at the pixel (x, y) is the sum of all the pixels to the left and above (x, y) in the original test image:

$$ii(x,y) = \sum_{x' \leq x, y' \leq y} i(x', y'),$$

where *ii(x,y)* is the integral image and *i(x,y)* is the actual image.

The integral image can be computed in one pass for an image and thereafter, the sum of pixels for any rectangular region can be computed with just four array references (Fig 2).

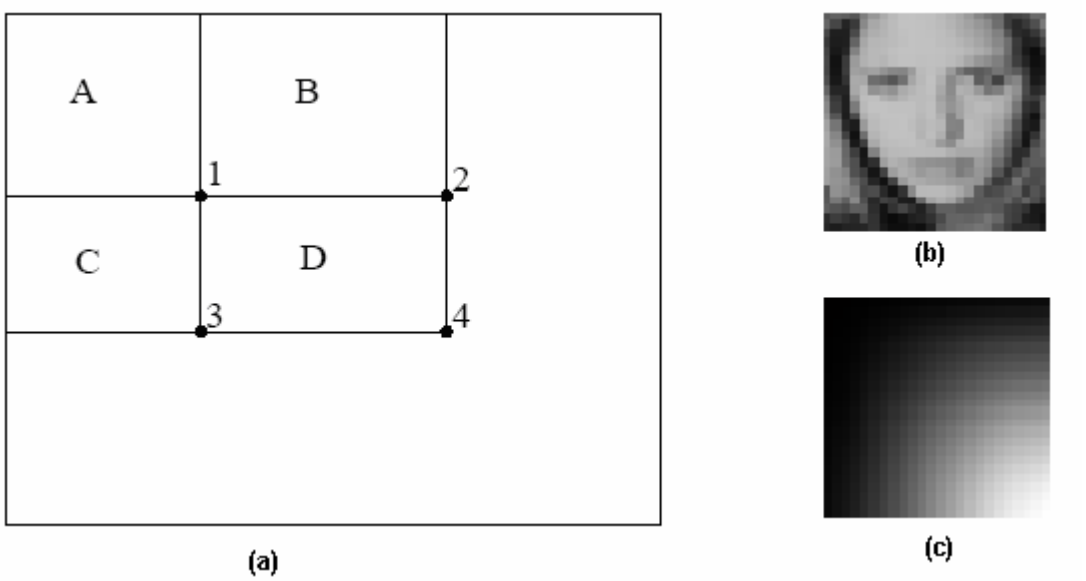

Fig 2. (a) The value of the integral image at 1 is the sum of pixels in rectangle A. The value at location 2 is A+B, at location 3 is A+C, and at location 4 is A+B+C+D. The sum within D can be computed as 4+1-(2+3). (Source: IJCV'04 Viola-Jones)
(b) shows a face image from the database and (c) is its corresponding integral image.

1.3 Learning Classifier Functions

This is the main learning stage of the system which accomplishes two things simultaneously. Firstly, it selects "good" discriminative features out of a pool of thousands of possible candidates. Secondly, it learns a classifier using these features that decides whether a region is a face or not. Both the objectives are achieved using a well-known learning algorithm, AdaBoost [5]. In a nutshell, AdaBoost uses a combination of simple weak classification functions to build a strong classifier. The main idea behind AdaBoost is to boost the performance of a simple weak learning algorithm. A weak learner is one which performs only slightly better than chance. However, a number of such weak learners can be used to boost the overall performance.

The weak learner used in the paper is based on a single feature. For all the training data, labeled as faces and non-faces, all the possible features are computed, along with their corresponding optimal thresholds that minimize their individual misclassification errors. Among all these features, the one that has the least error is selected as a "good" feature and its threshold acts as the separating boundary between faces and non-faces. Thus, a weak classifier consists of a feature (f), its threshold (θ), polarity (p), and the following hypothesis (h):

$$h_j(x) = \begin{cases} 1 & \text{if } p_j f_j(x) < p_j \theta_j \\ 0 & \text{otherwise} \end{cases}$$

Here, *x* is a 24x24 pixel image region. Since no single classifier can achieve desirable classification, a number of such weak classifiers are selected. At every stage, a single classifier is culled out, as discussed earlier, and before continuing for the next stage, each of the misclassified training images are re-weighted proportional to the classification error. The selected classifier also has an associated confidence (*alpha*) which is inversely proportional to the classification error. The process is continued till sufficient features have been chosen that can give overall low error. Initially, all the images of the same label are weighted equally, partitioned equally between the face and the non-face data. The global threshold is the half of the sum of confidence values of each of the selected features. A sub-window is assigned the confidence value of a classifier only if it passes its hypothesis. The sum is accumulated over all the classifiers, and if the final value is greater than the global threshold, the sub-window is declared as a face.

Two of the first features selected by AdaBoost are shown in fig. 3. These features make sense since eyes, nose and cheeks are the most discriminate parts of a face.

Fig 3. First two features selected by AdaBoost. Source: IJCV'04 Viola-Jones)

1.4 The Detection Cascade

In practice, no single strong classifier is used. Instead, a series of many such classifiers are learnt to form a cascade of classifiers. The simpler classifiers come earlier in the cascade and they can reject majority of non-face like sub-windows while retaining almost all the regions containing a face. The sub-windows that pass these earlier simpler classifiers are tougher to distinguish from faces and require more complex analysis. This is where the later stages of the cascade prove useful (Fig 4).

The final desirable false positive and detection rate governs the individual accuracy values for each of the stages. For example, in order to get a detection rate of 0.9, 10 stages can be trained with the individual detection rates of 0.99 (0.99 ^10 = 0.9). The number of selected "good" features or each of the stages in the cascade is determined on the basis of desirable false positive rate for that stage. Lower false positive rate would require more features but the increased accuracy comes at the expense of higher computation time. So, for earlier simpler classifiers, the false positive rate can be chosen to be high while maintaining the false negative rate to be close to zero. For the later stages when very few "easy" sub-windows will be encountered, false positive rate should be set much lower accompanied with a leeway in the detection rate so that the classifier is able to discriminate between faces and tougher face-like regions in the image.

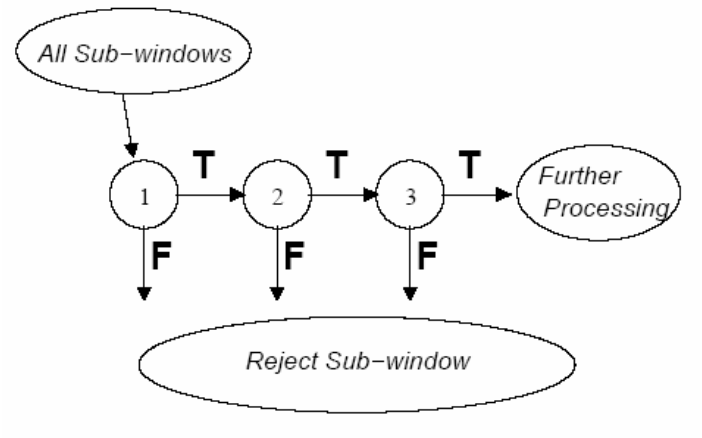

Fig 4. Schematic description of the detection cascade. (Source: IJCV'04 Viola-Jones)

The implementation described in the paper uses two features for the first stage which can discard 50% of the non-face sub-windows while retaining close to 100% of the faces. The next stage has ten features which can reject 80% of the false positives of stage 1, while correctly classifying all the faces. The next two classifiers have twenty-five features each followed by three fifty feature classifiers. The complete cascade consists of 38 stages with a total of 6060 features. It in important to note here is that every particular stage in the cascade is trained only on the non-face images which are not correctly classified by the partial cascade upto that stage. The maximum number of non-face images used in the paper at any stage was 6000 while the same database of faces was used at every stage.

1.5 Performance

A comparison of the performance of this face detector system with the existing best performing systems clearly highlights the speed with which it can detect faces. On a conventional desktop with 700 MHz Intel Pentium III, it can detect faces in 384x288 pixel images at 15 fps with very high accuracy.

For detailed information regarding the performance and overall detection system, the reader can consult the original paper.

2 Implementation on CMUcam3

2.1 Training Stage

The training for the face detector was done in Matlab using 4916 face images and about 150,000 non face images which were randomly cropped from non face images. All the training images were scaled to 24x24 pixels. The face database can be downloaded from the link: http://www.cs.ubc.ca/~pcarbo/viola-traindata.tar.gz and it is claimed that the same was used by Viola and Jones in their paper.

The minimum size of any feature was chose to be 8x8 which reduces the total number of features to about 50,000. The final detection cascade consists of five stages – first stage has 9 features, the next one has 10, followed by 25, with the last two comprising of 100 features. The number of stages in the cascade can be easily increased with a few simple modifications to the C code.

Even though the total number of stages in the cascade and the size of any individual stage in terms of the number of features are not critical to the performance of the system with respect to the accuracy, however, a smart selection of these parameters can boost up the overall speed of detection. Initial stages have less number of features so that most of the "easy" non face sub-windows can be rejected early without much computation. The tougher sub-windows have to go through more involved computations but they are fewer in number too.

2.2 CMUcam3 Implementation Issues

Available RAM in the CMUcam3 processor is a major bottleneck in the implementation of the face detector. Accounting for 8K stack, only 54K of ROM is available for computations. Considering a lower resolution gray scale image of 176x144, the integral image alone requires about 76K of memory (with 24 bit per pixel). Besides, there are no floating point operations available on the board. The clock speed at 60 MHz isn't terribly fast either. Even with all these restrictions, it is possible to get baseline performance for face detector in a controlled environment, in terms of both detection rates and speed. It takes about 5-6 seconds for CMUcam3 to detect faces. It can report multiple detections for a single face with an offset of few pixels.

In order to minimize the effect of variations in illumination, each candidate sub-window is standard deviation normalized before processing as described in the paper. Ideally, if enough memory is at the disposal, a squared integral image can be maintained on the lines of integral image but with sum of square of pixel values. This would bypass the need to compute standard deviation explicitly for each of the sub-windows as it is trivial to calculate it using integral image and squared integral image. Even though, due to memory constraints maintaining such an array isn't feasible, a similar version of the same idea is implemented details of which are described later.

In order to further speed up the process, an initial check is performed on the sub-windows before feeding them to the actual face detector. If the sub-window is too homogeneous

(std < 14) or too dark/bright (mean < 30 or mean > 200), it is discarded straight away and not processed further.

2.3 CMUcam3 C code description

The C code implementation of the face detector basically consists of two files – viola-jones.h and viola-jones.c.

viola-jones.h - contains the declaration of various constants, parameters and the feature values to be used in the main C file. The header file is well documented and is self-contained with respect to understanding the usage of all the variables.

viola-jones.c – This C file contains the actual implementation of the face detector. Apart from main(), it contains two helper functions – *cc3_get_curr_segment()* and *cc3_get_feat_val()*.

- *cc3_get_curr_segment()* reads the image from the FIFO and calculates the integral image simultaneously. At the first call of this function (for a particular frame) 61 rows (default value of *CC3_INTEGRAL_IMG_HEIGHT*) are read at once, the integral image is computed and stored in *cc3_integral_image* array. On the following calls, only a single row is read and the corresponding location in *cc3_integral_image* updated. There is an optional functionality to save the actual image to the MMC card too, for debugging or further processing.

- *cc3_get_feat_val()* computes the value of a particular feature for a particular stage in the cascade for the current sub-window. The global variable *curr_cascade* keeps track of the current stage in the cascade (*0-CC3_NUM_CASCADES-1*), *feat_num* tells which feature needs to be evaluated and *(x, y, curr_scale)* are the parameters for the current sub-window. If the feature value evaluated for the sub-window exceeds the feature threshold, the function returns the feature confidence (*alpha*), else it returns zero. In order to add another cascade stage, all that is required is adding/updating required parameters in the header file and replicating an "if" loop in the function by changing the corresponding feature values.

Saving to MMC: The coordinates of the detected face windows can be saved to a text file on the MMC card by specifying a pre-compiler primitive: *SAVE_IMAGES*

Matlab Files: Besides the C code, additional Matlab files have been provided to help the user while playing around with the face detector.

- ***get_rows_to_eval_feat.m***: saves a txt file containing a matrix *cc3_rows_to_eval_feat* which describes which rows need to be check for a face at which scales (details in the next section)
- ***generate_feat_in_struct_for_C.m***: saves a txt file containing the format in which features for a cascade need to be "copied" into a header file

2.4 Parameters and Constants

This section lists various predefined parameters/constants used in the code. Majority of these constants are declared in the header file *viola-jones.h*.

- *CC3_INTEGRAL_IMG_HEIGHT*: Height of the integral image

- *CC3_INTEGRAL_IMG_WIDTH*: Width of the integral image (same as that of the actual image in FIFO)

- *CC3_IMAGE_HEIGHT*: Height of the cropped image used for face detection (see *top_offset* and *bottom_offet* and header file for more details)

- *top_offset*: Number of rows cropped from the top of the actual image in FIFO (mainly due to memory considerations and speed-up)

- *bottom_offet*: Same as *top_offset* but it's the number of rows cropped from the bottom

- *cc3_integral_image*: [*CC3_INTEGRAL_IMG_HEIGHT*] x [*CC3_INTEGRAL_IMG_WIDTH*] matrix to store the integral image

- *CC3_NUM_SCALES*: Number of scales for the scanning window

- *CC3_SCALES*: Scale values for the scanning window

- *CC3_WIN_STEPS*: Steps for the scanning windows for each scale (Same length as *CC3_SCALES*)

- *CC3_NUM_CASCADES*: Number of cascade stages for face detector system

- *CC3_GLOBAL_THRESHOLD*: Global threshold for each of the cascade stages

- *CC3_NUM_FEATURES*: Number of features at each stage of the cascade

- *cc3_rows_to_eval_feat*: Matrix of size [*CC3_INTEGRAL_IMG – top_offset – bottom_offset]* X *[CC3_NUM_SCALES]*. A '1' in the matrix means that a particular row has to be checked for that scale for a face. Since the image size and all the different scales and step sizes are fixed beforehand, it is know which all sub-windows need to be evaluated. A Matlab script, get_rows_to_eval_feat.m, is provided that generates this matrix automatically for given scales/steps (if the user wants to play around with different values)

- *CC3_FACE_FEATURESi*: The matrix (see: *cc3_feature*) that stores the feature values and their different parameters for each of the cascade stages. 'i' takes the

value from 0 to *CC3_NUM_CASCADES*-1. The number of rows in each feature matrix is the number of features for that stage and the number of columns is the number of scales. The feature values are computed for all the scales, beforehand, and stored so save the computations at the runtime. A Matlab script, generate_feat_in_struct_for_C.m, is provided that does this too.

Struct cc3_feature: The features are stored in a matrix where each entry is a structure of the type cc3_feature. Description of the structure is below:

```
typedef struct
{
  uint8_t x[9];
  uint8_t y[9];
  int8_t val_at_corners[9];
  int8_t parity;
  int32_t thresh;
  int32_t alpha;
} cc3_feature;
```

- *x* & *y*: coordinates of the features. Even thought not all features have nine coordinates (two-rectangular features have six), same size has been chosen to maintain a generic structure for features. For those with lesser coordinates, undefined values are made zero.
- *val_at_corners*: Value of the features at the coordinated specified by (*x,y*) (zero for undefined coordinates)
- *parity, thresh, alpha*: Feature parameters (refer to the detailed algorithm for details)

Important Note about Features: Since there is no floating operation possible on CMUcam3 board, all the feature values have been scaled by a factor (default: 100) so that they cover the required variation is the values. Also, when features are scaled for different scales, only *x* and *y* coordinates change and rest all the values remain the same.

2.5 Code Flow

The flowchart in fig.5 describes the basic flow of the algorithm as implemented in CMUcam3.

Due to limited RAM, the whole integral image can't reside in the memory. So, only a block of the image is present in the memory at any given time. In fact, the actual image is never stored since the face detector operates on the integral image and not the original one. *cc3_integral_imgage* contains the actual integral image while *cc3_image_t* is used as buffer to read the image, row by row, from the FIFO. This functionality is taken care of by the function *cc3_get_curr_segment()*.

Once the integral image is available, in the main() function, we iterate over the rows of cc3_integral_image, starting from 0. For the current row, cc3_curr_row_ii, if sub-windows need to be computed at a particular scale (given by cc3_rows_to_eval_feat matrix), its mean and variance are computed. If these values exceed a certain threshold,

the face detector algorithm is operated on it and if is passes all the cascade stages, the sub-window is reported as a detection. The results (coordinates and size of the detected sub-windows) can be saved to a MMC card too in addition to sending over the serial port to the computer (see: Section 2.3).

Every iteration, a read is read from the FIFO (if we haven't read all the rows) and the integral image is updated. During the same iteration, a row is read from the integral image, and checked for faces. The process is repeated till the whole image has been check for faces.

Mean and variance: In order to make the face detection robust to illumination changes, it is required to normalize the sub-windows to make their variance unity. To compute the variance, we need the sum (basically the mean) of the pixels and the sum of the square of the pixels in the sub-window. The sum can be easily computed using the integral image while computing sum of the square of the pixels is very expensive. However, since most of the sub-windows overlap substantially, we can use the results of the past window for next immediate sub-window.

The sum of the square of the pixels is calculated explicitly for the first sub-window for each of the scales only once (top left corner of the image). Knowing the value of step provides the amount of overlap between the sub-windows. These are stored in horz_past_sum_sq_pix for the horizontally shifted windows (i.e. shifted windows for the same row) and vert_past_sum_sq_pix for the vertical shift (first windows at consecutive rows, as defined by CC3_WIN_STEPS). The values are updated for every evaluation of a sub-window and the same procedure is applicable across all scales.

This technique is very helpful in boosting the over-all detection speed. Without this scheme, i.e. computing the sum of square of pixels independently for each of the sub-windows, it takes about 18 seconds to run face detection on a 176 X 144 gray scale image. On the other hand, exploiting the overlap between windows provides a speed-up of upto 3X and face detection takes about 5 seconds for the same image.

In the implementation, the image sub-window is never explicitly divided by the standard deviation. Since the feature value evaluated for a particular region is just the difference of sum of pixels, it is sufficient to just divide the feature value by the standard deviation. This saves expensive division and prevents propagation of undesirable rounding-off errors.

2.6 Performance

The face detector on CMUcam3 works well in a controlled environment without too much background clutter. Some of the test results from CMUcam3 board are shown in Fig 6. The detected face windows and the captured images were saved to the MMC card and a Matlab script was used to draw window patches around reported detections. Currently, it takes around 5-6 seconds to detect a face.

For any questions or feedback, kindly email …

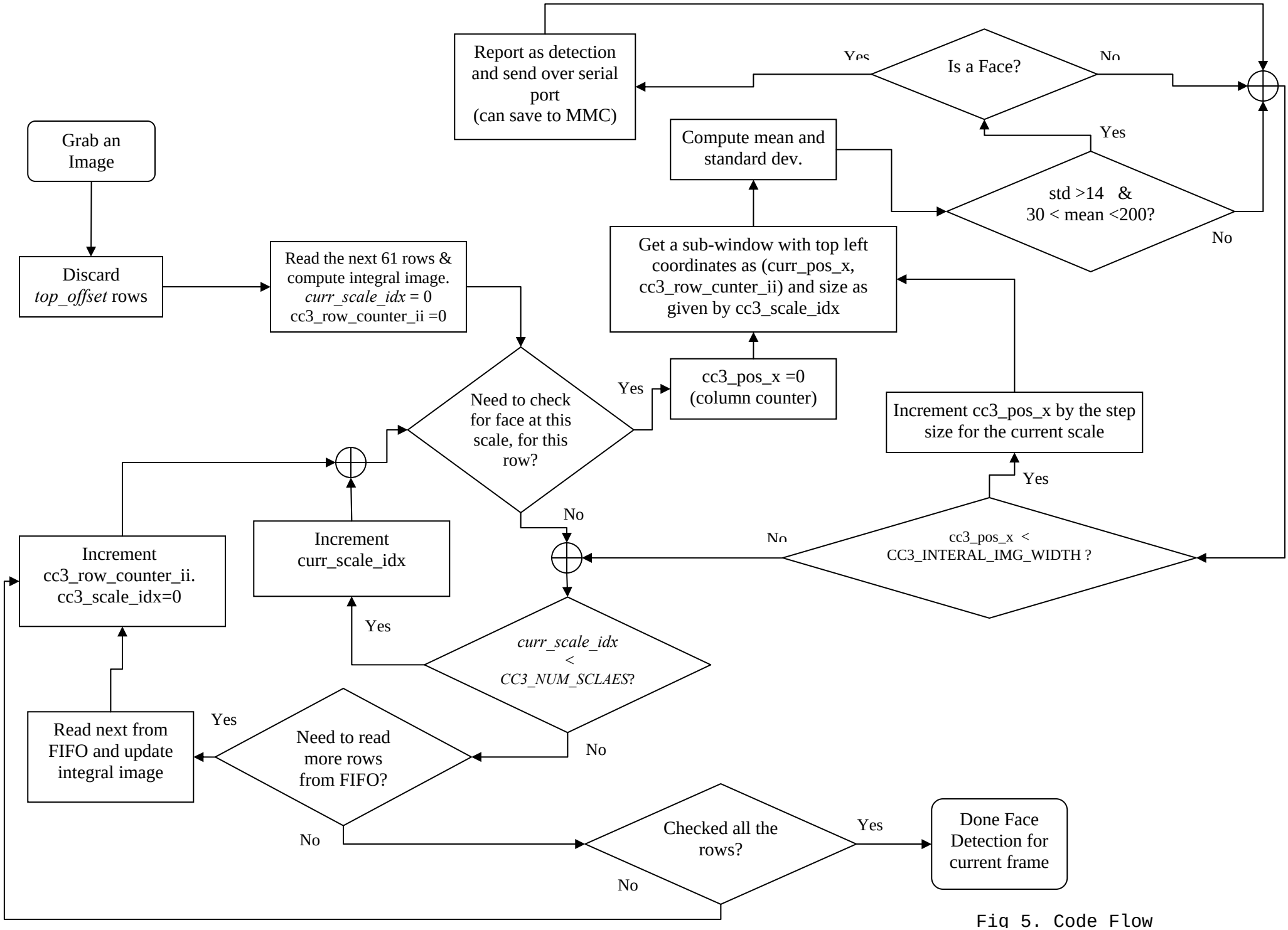

Fig 5. Code Flow

Fig 6. Detected Faces from CMUcams3 Face Detector

References:

[1] "Robust real-time face detection", P. Viola and M. Jones, *International Journal of Computer Vision*, 2004

[2] "A statistical method for 3D object detection applied to faces and cars", H. Schneiderman and T. Kanade, *International Conference on Computer Vision*, 2000.

[3] "A snowbased face detector", D. Roth, M. Yang and N. Ahuja, *Neural Information Processing*, 2000

[4] "Neural network-based face detection", H. Rowley, S. Baluja and T. Kanade, *IEEE Pattern and Machine Intelligence*, 1998

[5] "A decision-theoretic generalization of on-line learning and an application to boosting", Y. Freund and R.E. Schapire, *Computational Learning Theory: Eurocolt 95, Springer-Verlag, pp. 23-37*, 1995

[6] "A general framework for object detection", C. Papageorgiou, M. Oren, and T. Poggio, *International Conference on Computer Vision*, 1998.