REAL TIME FACE AND FACIAL FEATURE DETECTION AND TRACKING

George Arceneaux IV, Allison Katherine Schedin, Andrew John Willson White
Department of Computer Science
St. Olaf College
Northfield MN 55057
Schedin@stolaf.edu

Abstract

In this paper, we present a means of detecting and tracking a face and its features. There are many possible applications for our findings such as gaming, conferencing, and handicapped assistance. Our method of detection and tracking uses OpenCV to load boosted Haar classifier cascades which allows an initial detection of the face and facial features. Then, by using hue histogram thresholding via OpenCV's Camshift feature and then template matching, our program is able to track the face and its features. Our method of detection achieves great speed and generality at a slight cost to accuracy.

Introduction

Face detection and tracking has been an endeavor in computer vision for over two decades. Programmers have devised numerous methods for accomplishing this task quickly and accurately, including the use of Haar-like feature classifier cascades, skin detection, motion detection, segmentation and background removal, and geometric modeling. Support for face detection and tracking is also available through the OpenCV library, an open-source library for use in computer vision programming.

Our ideal goal is to fully implement a face detection program that can find a face in a variety of conditions and track it throughout a diverse sequence of movements, expressions, and rotations. In one month, we have achieved high speed and good accuracy under general conditions. Our program has been tested with successful results, but certainly not perfect ones. For example, rotated and tilted faces are not always detected or may misrepresent the actual location of the features.

Referenced Work

Before beginning our project, we did research on previously implemented methods for facial tracking and related tasks. (Articles read are listed in References [1]-[7].) From these articles we took the methods for implementing boosted Haar classifier cascades and template matching. We also researched the general ideas of skin detection, eye tracking, and expression recognition.

In addition we used code from the OpenCV wiki (facedetect.c, [9]) as well as from Robin Hewitt and Nash Ruddin ([10][11]).

Overview of our proposed method

Here a broad overview of our proposed method is given. The specifics of each step will be presented in the following section.

We begin our process by capturing a frame from a webcam video feed and searching it with a pre-processed boosted Haar classifier cascade for faces, implemented as proposed by Viola and Jones [5]. The program continues to capture frames and search them until a face is found, at which point a color histogram is taken from the rectangular area surrounding the detected face. The tracking of the faces in all subsequent frames is then handled by hue thresholding, wherein a probability value for each pixel of the image is calculated based on the color histogram and its proximity to the current detection. The area with the biggest cluster of high probability pixels is determined to be the facial region, and an ellipse is drawn around that area. This process is accomplished by using OpenCV's Camshift.

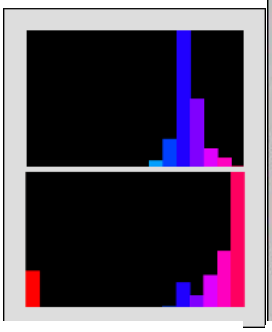

Figure 1: Two examples of the color histogram that Camshift creates upon detecting a face. The highest bars represent the most common hues in the facial region.

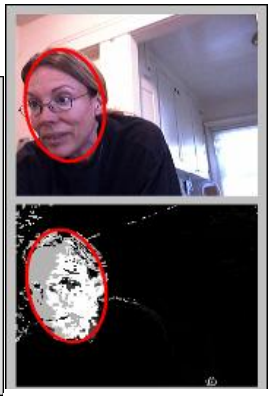

Figure 2: The normal and face-probability views from Camshift. In the face-probability view, black pixels have the lowest value, and white, the highest.

If the tracking fails at any point, the Haar cascade is run again until the face is found. Then the Camshift process is repeated.

Once the facial region is determined, another series of Haar classifier cascades is run in the frame but restricted to certain areas within the face to detect facial features. For instance, the cascade corresponding to the mouth is only applied within the lower portion of the determined facial region. This way, detection is sped up considerably and false detection is much less likely to occur.

After a facial feature has been detected, a template is taken of the area inside the rectangle created by the cascade. The template can be of any specified size and placement, so some templates consist of the entire feature region while others, such as irises, are a small portion of them. The newly-created templates are displayed in separate windows for viewing.

From this point on, template matching is used to track each feature. If a user has a particularly unique facial feature, as long as it can be detected once, then it will still be easy to track because the templates are unique for each person. The template matching, like the cascades, is restricted to certain areas of the facial region to greatly improve the speed of the program.

Proposed Method

OpenCV

Our program depends heavily on the open-source library OpenCV. Features such as Camshift and a number of Haar classifier cascades are used as well as drawing functions and object classes like CvRect and CvPoint. Capturing from cameras and from .avi files are also given convenient functions. The beginning of our project consisted of a piece of code named facedetect.c that is both provided and explained on the OpenCV wiki page [8]. When run, it detects all faces within a video frame or given image via Haar classifier cascades and draws red rectangles around them.

Later we implemented the code TrackFaces, which relied on the OpenCV feature Camshift and hue thresholding.

Haar-like Classifier Cascades

Haar-like feature classifier cascades are composed of multiple classifiers, or conditions, that are used to distinguish unique objects (i.e., a face, eye, panda, etc) from anything that is not that object. Classifiers are a set of values representing sums of pixel brightness on a region-by-region basis. These specifically structured regions are called Haar-like features and are pictured below superimposed on a man's face.

Figure 3: Two examples of Haar-like features superimposed on a man's face.
The features' unique sub-regions are indicated by black and white.

Haar-like feature classifiers are created using integral images. Each pixel in the integral image represents the change in brightness of the corresponding pixel in the original image by finding the sum of the derivatives of the pixels above and to the left of the original:

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

Figure 4: The equation used for finding the change in pixel brightness to create an integral image.

After calculating the integral image, any Haar-like features can be computed at any scale or location in real time.

Cascades are trained by running a list of weak classifiers against a dataset of images, some of which have the desired features, called positive images, and some of which do not, called negative images. After a weak classifier sorts the dataset, a process called AdaBoosting gives higher weight to the mis-sorted image data and proceeds to run another classifier. This process is repeated so multiple classifiers are found for detecting increasingly specific data. Thus, multiple weak classifiers are combined to make one strong classifier.

The weak classifiers are implemented in a cascade, so the most general classifiers are run first. Increasingly specific classifiers are then run in the areas which were identified by the previous ones. This way time is not wasted on areas in an image which have little to no chance of being a desired feature. The result is a fast and accurate object detector. This framework was introduced by Paul Viola and Michael Jones in 2001 [3].

Our method begins by using these Haar cascades to identify the locations of different features including the face.

Hue Thresholding

After the first Haar cascade is used and the face is initially detected, hue thresholding is implemented to keep track of the face. OpenCV begins by creating a histogram of which colors are most prominent in the region bounded by the Haar cascade.

The next step is to determine which colors represent the face. The most commonly found hue indicated by the histogram is assumed to be the face. Afterward, OpenCV uses Camshift to continually follow the assumed colors and thus the face. While this assumption can be manipulated by using material with similar colors to the face to "trick" the tracker, it has proven to be very reliable.

Template Matching

The next step is to use template matching. Like the hue detection, each template is initialized by a Haar cascade.

Each feature has its own Haar cascade which is used to identify each feature's location. Upon completion, a snapshot of the area within the cascade is taken. This snapshot is the template.
A template is a base image which is used to locate similar images. For example, a template of a person's eye is taken and then the entire image is searched for the region which looks most like that eye.

While this process is very useful, it can be very inaccurate and slow. Because the entire image is searched, many different areas might be detected as the desired feature and because the whole image is searched, it becomes much slower. These two problems are countered in two different ways. The first is thresholding. By setting a determined threshold and then comparing results found by the template matcher to that threshold, we can determine if the area within the image is accurate enough to be identified with the desired feature.

The second method binds the matching within regions of interest. OpenCV provides a means of cropping an image temporarily. Thus, instead of searching the entire image, the template matcher searches only the cropped region for a certain feature. While this primarily addresses the issue of speed, it also helps improve accuracy by making the matcher look only in the most probable location for detection.

Results

We subjected our program to two tests. The first used our simple eye detection with Haar cascades. Our peers, David Tengdin and Janet Burt, created a program for facial recognition.

They used a dataset of the entire St. Olaf student body and its staff and detected eyes in each picture. The criterion for success was that the picture would return with two successfully detected eyes. If only one eye was detected, or there were false detections, then the test was considered a failure. Of the 3884 pictures tested, 2778 images were successes, resulting in a success rate of 71.85%.

For our full program, we collected data from 12 peers to see how many frames successfully detected each feature: the face, eyes, irises, nose, and mouth. The subjects were asked to move their face to the left, then right, then move forward and out, then turn slightly left and right, smile, frown, look left and look right without moving their heads. After calculating the average of all the individual tests, we found the results depicted in figure b. The eyes and irises were tracked notably well, but the nose had a very low success rate. Though its impact on the data was not large, false detections were also counted positively. We also had great success tracking the face, as redetection was rarely needed.

	Face Count (x detected)	Right Eye	Right Iris	Left Eye	Left Iris	Mouth	Nose
Allison	1	95%	99%	94%	99%	96%	95%
Andrew	1	87%	99%	95%	99%	87%	85%
George	1	94%	99%	95%	99%	99%	99%
Caroline	1	94%	99%	75%	85%	87%	0%
Peter	1	68%	99%	99%	99%	99%	24%
Mary	1	99%	99%	39%	41%	5%	66%
Ryan	1	93%	98%	95%	95%	83%	38%
Patrick	2	94%	99%	99%	99%	23%	10%
Jon	1	23%	99%	4%	60%	99%	75%
Janet	1	94%	99%	95%	99%	70%	60%
David	2	67%	99%	96%	99%	90%	2%
Phineus	1	55%	99%	53%	99%	30%	2%
Total Average	1.17	93%	99%	92%	97%	72%	52%

Table 3: Table of results from the peer test.

Conclusion

The program thus far has generated considerable success. Our program does well under certain conditions; for instance, if a user is squarely facing the camera and does not rotate their head more than slightly. However, the deplorable failure of the nose tracking as well as the false detections due to face rotation indicates a need for further work. We began the implementation of two trackbars dependent on the movements of the user's face and irises which so far have enjoyed limited success. This indicates the possibility for future incorporation of our program in different ways.

Further possibilities include better tracking with more extreme rotations and scales, a smoother user interface, and a more practical application.

References

[1] L. Chen, G. Kukharev, K. Peng, Su Ruan, "A Robust Algorithm for Eye Detection on Gray Intensity Face without Spectacles," *JC&T* vol. 5, (3) pp. 127-132, 2005.

[2] M. Betke, M. Chau, "Real Time Eye Tracking and Blink Detection with USB Cameras," *Boston University Computer Science Techincal Report*, vol. 12, may 2005.

[3] Z. Yao, H. Li, "Tracking a detected face with dynamic programming," *Image and Vision Computing*, vol. 24, (6), pp. 573-580, 2006.

[4] F. Dadgostar, A. Sarrafzadeh, "An adaptive real-time skin detector based on Hue thresholding: A comparison on two motion tracking methods," *Pattern Recognition Letters* vol. 27, (12), pp. 1354-1352, 2006.

[5] Paul Viola, Michael Jones, "Rapid Object Detection using a Boosted Cascade of Simple Features," Accepted Conference on Computer Vision and Pattern Recognition, 2001.

[6] X. Xie and K. Lam, "Facial expression recognition based on shape and texture," *Pattern Recognition*, vol. 42, (5) , pp. 1003-1011, 2009.

[7] Y. Cheon and D. Kim, "Natural facial expression recognition using differential-AAM and manifold learning," *Pattern Recognition*, vol. 42 (7) (2009), pp. 1340-1350.

[8] Y. Cheon and D. Kim.

[9] (2008, Jan.). FaceDetection using OpenCV [Online], Available: http://opencv.willowgarage.com/wiki/FaceDetection?action=fullsearch&context=180&value=localhost&titlesearch=Titles

[10] R. Hewitt. (Mar. 2007) Seeing With OpenCV, Part 3: Follow that Face!. *SERVO Magazine*. [Online]. Available: http://www.cognotics.com/opencv/servo_2007_series/part_3/index.html

[11] N. Ruddin. (Jan. 2009). Real Time Eye Tracking and Blink Detection. Available: http://nashruddin.com/Real_Time_Eye_Tracking_and_Blink_Detection