Face Detection and Recognition in Office Environments
by
Jeffrey S. Norris

Submitted to the
Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science at
the Massachusetts Institute of Technology

May 21, 1999
[June 1999]

Author_____
Department of Electrical Engineering and Computer Science
May 21, 1998

Certified
by_____
Paul Viola
Thesis Supervisor

Accepted
by_____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Face Detection and Recognition in Office Environments
by
Jeffrey S. Norris

Submitted to the
Department of Electrical Engineering and Computer Science

Friday, May 21, 1999

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

The Gatekeeper is a vision-based door security system developed at the MIT Artificial Intelligence Laboratory. Faces are detected in a real-time video stream using an efficient algorithmic approach, and are recognized using principal component analysis with class specific linear projection. The system sends commands to an automatic sliding door, speech synthesizer, and touchscreen through a multi-client door control server. The software for the Gatekeeper was written using a set of tools created by the author to facilitate the development of real-time machine vision applications in Matlab, C, and Java.

Thesis Supervisor: Paul Viola
Title: Assistant Professor, MIT Artificial Intelligence Laboratory

Acknowledgements

I would like to thank:

Paul Viola, my thesis advisor, for the opportunity to work on such a great project, Mike Ross, for being the very first IVCCLIB user and his help on IVCCLIB for Java, my wife, Kamala, for her undying support and invaluable editing help, and my parents, for their constant encouragement.

CONTENTS

Chapter 1

Introduction

Bridging the gap between the world of the computer and that of its user has always been one of the chief goals of computer science research. Graphical interfaces, input devices, speech generators, and handwriting recognition systems are just a few examples of how we have made computers more accessible. Machine vision is a more recent thrust in this direction, and represents a major step in bringing the computer into our world. Humans rely upon vision more than any of our other senses, and the human brain has evolved in a way that indicates the necessity and complexity of observing our world through sight. The primary visual, striate, and visual association cortexes occupy over 25% of the neocortex, the region of the cerebral cortex which developed latest in our evolution [1],[2]. Given that our brains seem to devote such resources to the process of sight, it's not a surprise that machine vision is a very compelling, though challenging, field of research in computer science. My thesis focuses entirely on two specific challenges in machine vision: the tasks of detecting and recognizing human faces in real time, and the applications of these technologies in a modern office environment.

The last decade has been a time of incredible advances in face detection and recognition. Computers have become fast enough to perform computationally intensive

image processing tasks, and storage devices have grown to allow the accumulation of large databases of facial images. A rebirth of interest in neural networks has fueled many learning-based approaches to these problems. In addition, sadly, increases in terrorism and espionage have created a demand for a more effective means of human identification.

Many applications await each major advance in face detection and recognition, particularly in the field of biometrics. Biometrics is the statistical measurement of biological characteristics, primarily for the purpose of identification [3]. Popular biometric devices include fingerprint, palmprint, iris, and retina scanners, but all of these devices have the disadvantage that they require that the user initiate the identification process. Faces are the holy grail of biometrics because humans rely primarily on faces to identify one another, and because a person's face can be detected and recognized without the person initiating the process.

Several companies have already dedicated themselves to marketing biometric products which make use of the most recent advances in face detection and recognition. In November 1998, Visionics installed the "Mandrake" system in Needham, Scotland, where it attempts to recognize faces of known criminals in real-time footage from 144 CCTV cameras dispersed throughout the city. When a criminal is recognized, a control room operator is notified who can then contact police [4].

Another interesting application was annouced by Viisage last month. Viisage is spinning off Biometrica, a company that will begin marketing a product to casinos to help them protect themselves against known cheats and professional card counters. The system will allow casino operators to photograph cheats and enter them into a national casino database, allowing other casinos to deny admission to people who have been

banned elsewhere [5]. In addition, numerous systems are on the way from companies like these to prevent voter registration fraud, identify terrorists in airports, prevent unauthorized logins at computers, and allow ATM check cashing (see figure 1-1).

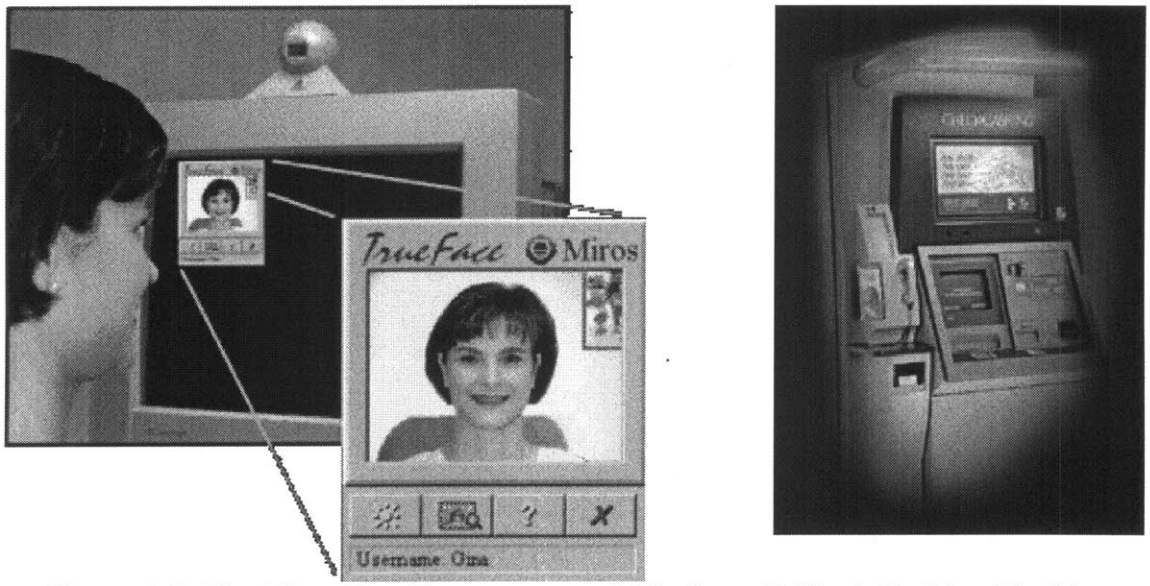


Figure 1-1: The Miros TrueFace system and Mr. Payroll Check Cashing Machine

The particular application that I created for my thesis is a doorway security system (shown in figure 1-2). This system, called "The Gatekeeper," detects the faces of people approaching an automatic door and compares them to a database of previously collected faces. If the person is recognized, they are greeted verbally through a speech synthesizer and visually through a touchscreen, and the door is opened for them. I designed the Gatekeeper system to serve both as a test environment for my research in face detection and recognition, and as a platform for further vision research by members of the Learning and Vision group in the MIT Artificial Intelligence Laboratory. To accomplish the latter goal, I developed an extensive set of tools, called IVCCLIB (Intel Video Capture Card Libraries). IVCCLIB greatly reduces the effort required to write machine vision applications in Matlab, C, or Java. It consists of a heavily modified

Figure 1-2: The Gatekeeper

driver for the Intel Video Capture Card, and a set of useful functions that allow a user to concentrate more on image processing, and less on the details of image digitization. I found IVCCLIB extremely useful while working on my thesis, and several other students in my laboratory have begun using IVCCLIB for their work as well.

The first two chapters of my thesis focus on my early work in face detection and recognition, and the development of the systems that would eventually form the bulk of the Gatekeeper system. I discuss several recently published techniques for face detection and recognition, and the results I achieved after implementing them. I also describe a lightweight algorithmic approach to face detection that performs very well in a stable application environment.

The third chapter is devoted to a detailed description of the Gatekeeper system. I explain each of the components of the system in detail, including the hardware used for capturing images and controlling the door. I also discuss the IVCCLIB API and some of the software developed using it.

In the fourth chapter, I describe two applications of my work. First is the contest I designed for the 1999 AI Olympics, in which teams of AI Lab researchers designed competitive recognition systems using an early version of IVCCLIB. Second is the door security system I mentioned above. Finally, I discuss some retrospective conclusions on my work, and some possible extensions.

Chapter 2

Background Work

Before attempting to construct any working face detection and recognition applications, I dedicated several months to researching and implementing classic and state-of-the-art approaches to these problems. Since I had defined my application environment as indoor and relatively stable, I was able to read papers with a critical eye to what would likely perform well in that environment. Implementing these approaches helped me to understand them much more thoroughly and allowed me to accumulate a code base that I would eventually use in some applications of my research.

This chapter focuses on the recent research in face detection and recognition that motivated my work, as well as the systems I implemented to attack these problems. I combined the final versions of these systems to make the Gatekeeper, which is described in detail in Chapter 3.

2.1 Face Detection

I first focused my attention on the problem of locating a face in a still image. My search of the literature in this field produced several early approaches relying on fairly straightforward density estimation techniques [6], some neural network systems [7], and a few feature-based systems as well. I decided to begin with a neural network approach to the problem. After I had a simple neural network system working, I implemented an

approach developed at Carnegie Mellon University that uses a more complex network topology. Finally, I experimented with some algorithmic approaches that didn't rely on learning and achieved remarkably good results.

A simple neural network produces mediocre face detection results

After reading several papers on the topic of face detection, I investigated the performance of a very simple neural network-based face detector in Matlab with the hope that it could serve as a benchmark for my later work. The network I implemented had 400 inputs (one per pixel in an image), 5 hidden units with a sigmoidal transfer function, and one output that I trained to indicate that a face was present (see figure 2-1). The basic steps followed by the system as it searched for a face in an image are as follows:

1. Select every 20x20 region of the input image and use the intensity values of its pixels as the 400 inputs to the neural network.
2. Feed the values forward through the network, and if the value is above .5, the region likely represents a face.
3. Repeat steps 1 and 2 several more times, each time on a resized version of the original input image in order to search for faces at different scales.

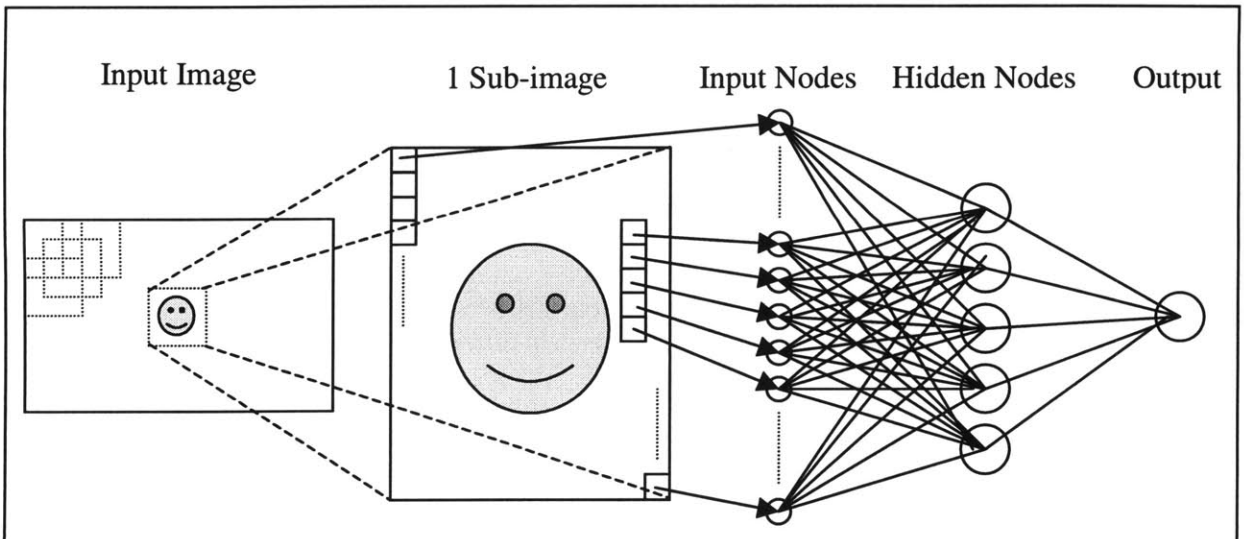
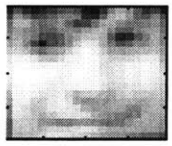


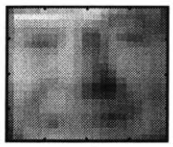
Figure 2-1: A simple neural-network approach to face detection

Once I had finalized the network topology, I needed to locate a data set to train it. I selected a face database developed by Kah-Kay Sung at MIT [6] during his graduate work with Tomaso Poggio. I used the first 700 pictures in Kah-Kay Sung's data set of 1488 faces to train the network, along with 700 random noise pictures as negative examples. The network converged to 1.2% error after training for 50 epochs, at which point I decided to perform some tests.

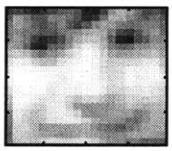
I tested the network using the remaining 788 faces in Kah-Kay's data set, followed by 788 random noise pictures. On this set of 1566 examples, 35 misclassifications were made (2.23% error). Out of these, 32 were false positives on the noise data. Figure 2-2 shows the three face images that were not correctly identified as faces. It is worth noting that two of the pictures are of the same person. I was surprised that the network performed so well with only 5 hidden units, but I suspected that the task of distinguishing faces from noise wasn't as challenging as distinguishing faces from more typical image patterns.



#104



#117



#105

These are the faces that were not detected by the 5 hidden node network out of a set of 788. (Note that two are the same person at different rotations).

Figure 2-2: Problem

To confirm my suspicions, I tested the system on the picture in figure 2-3. There are 1305 20x20 patches in this picture of Savannah, GA (after downsampling to ¼ size), and the system found 327 faces where there are clearly none (25.06% error). Rather than strictly detecting faces, the network had been trained to differentiate between noise and more "typical" images, with a weighting towards faces.

Figure 2-3: A test image

In order to try an improve the system's rejection of non-face images, I next used the landscape picture in figure 6 along with half of Kah-Kay's face database to train a new network, also with 5 hidden units. This network also converged in around 50 epochs. I then tested the system on the picture in figure 2-4 (a waterfall in Nice, France). The system had a false positive rate of 11.3%, which is much better than the 25.06% error achieved by the face/noise discriminator. However, when I tested the system on the rest of Kah-Kay's data set and some noise images (the same test as I performed on the first network), there were 142 misclassifications—an error rate of 9.07%, and 18 of these were on the face images (as opposed to only 3 before)

At first glance, I was impressed with the performance of such a simple network on this problem, but when I tried to apply the system to a more difficult problem, it quickly became apparent that a more complex approach was

Figure 2-4: A second test image

necessary. However, merely increasing the number of hidden units would likely produce a system that would not be trainable in a reasonable amount of time. Instead, I turned my efforts towards more creative network topologies.

Rowley, Baluja, and Kanade use a more complex topology and achieve better detection results

In "Neural Network-Based Face Detection", Henry Rowley, Shumeet Baluja, and Takeo Kanade outline an upright-face frontal face detection system that relies on a receptive-field, or retinally-connected neural network [7]. Their system is very similar to the simple neural network approach I described above. It also works by examining many windows of an input image and uses a neural network to decide whether each window contains a face. The chief difference between these approaches lies in how the inputs to the neural network are generated.

The CMU system, rather than simply making each pixel of a sub-image an input to the neural network, divides the sub-image into regions of various shapes and sizes and averages the intensity values in each of these regions to produce inputs for the neural network. These regions are constructed as follows (see figure 2-5):

1. Divide the sub-image into 4 square regions, and use the average intensity in each of these regions as inputs for 4 of the neurons.
2. Divide the sub-image into 16 square regions and use the average intensity in each of these regions as inputs for 16 of the neurons.
3. Divide the sub-image into 10 20x2 rectangular regions, and use the average intensity in each of these regions as inputs for 10 of the neurons.

This approach has two benefits. First, by averaging the intensity values in each of these regions, the number of inputs in the neural network is reduced from 400 to 30. Second, the different shapes and sizes of the regions allow the network to learn different kinds of facial features. Large square regions can detect eyes and noses, while long, thin rectangles are more suited to horizontally arranged features like an eyebrow or mouth.

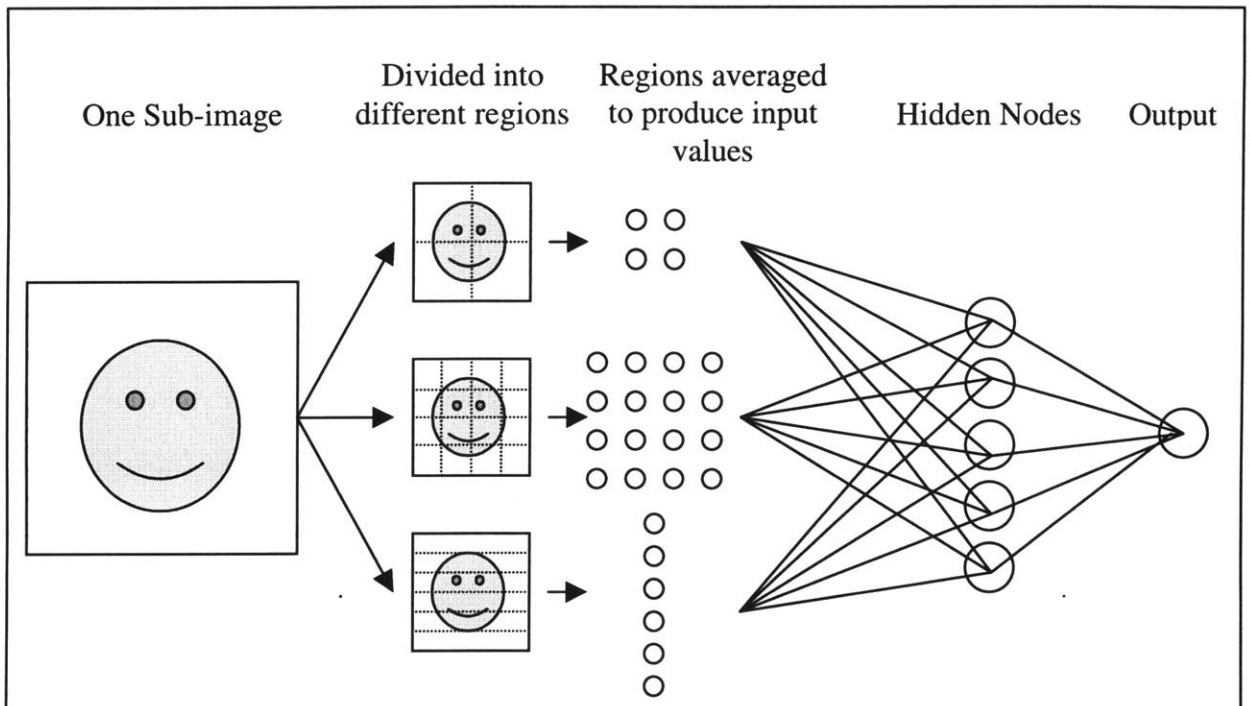


Figure 2-5: A more complex neural network approach to face detection

I was pleased with the accuracy of the CMU algorithm- after training it on half of Kah-Kay Sung's database, it achieved approximately 6% error on the rest of the database. However, in terms of its running time, the CMU approach was not as impressive. The Gatekeeper needed to be able to detect a face in a video stream, recognize it, verbally greet the user, and open the door before the user was able to walk approximately 25 feet, the distance from where they first became visible to the door. Ideally, it needed to be able to do the first two steps multiple times, in case the user happened to cover their face for a moment or turn their head to the side. My implementation of the CMU algorithm required 3-4 seconds to locate a face in an image, which was simply not fast enough without further optimization.

What was really needed was a way to focus the CMU's search space— perhaps in the form of a quick algorithm to tell it where to start searching. After finishing my work

on the CMU system, I began designing a lightweight, algorithmic solution to face detection that could support the CMU system.

An algorithmic approach to detection uses background subtraction and image morphology

Learning-based systems are attractive because they can exploit patterns in data that are difficult to discern. Unfortunately, they usually require extensive training, large data sets, and may never converge to a truly optimal solution. Furthermore, they are often highly sensitive to subtly different training methodologies. For this reason, lightweight algorithmic solutions to problems are often more desirable.

To be fair, the CMU system solves a much more difficult problem than the one I faced. They designed a system capable of detecting faces in still images with no prior knowledge of the likely scale or location of those faces. Clearly, my application environment allows me to make several assumptions that the CMU people weren't able to make:

1. The environment is still and unchanging most of the time, allowing for background subtraction.
2. A rapid change in the environment almost always represents a moving person.
3. Faces will always appear nearly upright
4. I can build a recognizer that has a very low false positive rate. This recognizer will be unlikely to misidentify non-face objects as a face, even at the expense of occasionally rejecting an actual face.

It should be noted that the last assumption heavily influenced my selection of a recognition system, and allowed the detection system to pass some false positives to the recognizer with some confidence that they would be rejected at the recognition phase. This was important because it allowed me to eliminate a costly detection refinement phase, speeding up the detection process immensely.

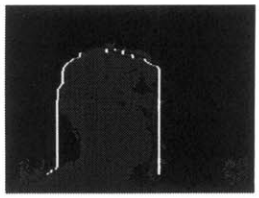
The basic steps that my algorithmic approach follows to detect a face are shown in figure 2-6.

1. At startup, record several frames of the camera's input and average them. Store this average image as the background image for further steps.

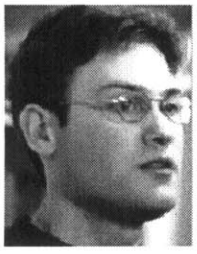
2. Capture an image and determine if a person is likely to be present by estimating how different the image is from the background.

3. Subtract the current image taken by the camera from the background image and apply a threshold to produce a binary difference image where white corresponds to areas that differ greatly from the background.

4. Apply an image morphological "erode" operation to remove artifacts in the difference image due to camera noise. Also remove highly unlikely regions from consideration.



5. Locate the top of the largest white region and trace the contour of the head portion of the region by performing a series of line-based morphological "close" operations and then finding the extents of these lines.



6. Grab the region of the original image that we now believe to be a face.

Figure 2-6: My algorithm for locating and returning a face

Several of the steps in the figure deserve further explanation. First, following Step 3, the difference image can be quite noisy. Regions corresponding to people are often full of "holes" where sections of the person happened to be similar enough to the background to slip through the threshold in Step 3. Many systems attempt to correct for these holes immediately using image morphology, but this is a slow process. Instead, I deal with these holes more efficiently in Step 5, after I've found a face region. In addition, there are often false positives in the images—white areas primarily due to camera noise, often appearing in horizontal line artifacts. Step 4 cleans out nearly all of this noise by using the image morphological erode operation (for an explanation of image morphology techniques, see Appendix C). This removes most of the noise without significantly affecting the contours of any of the regions of interest.

In Step 4, I use my knowledge of the application environment to bias the face detector towards certain regions of the image. For instance, I know that it is unlikely that I will find a face in the upper corners of the image because these regions correspond to positions higher than the tallest person likely to use my system. Consequently, my system assumes that any blobs detected in those regions are likely noise. Currently, the search for face regions simply starts at the top of the image and works downward, line by line, but it should be noted that this process could be made more efficient using this same knowledge about likely locations of faces.

By Step 5, I have found the top of a face region and need to locate the width of the face in order to determine its scale. I trace the contour of the face a line at a time by extracting the current line from the image and finding the horizontal extents of the white portion of the line. Often, the white portion of the line will not be contiguous due to the

holes within white regions mentioned above. At this stage, I can perform an image morphological dilate operation *on this single line*, which eliminates any holes very quickly.

My algorithmic approach produces surprisingly good results

The system outlined above is a simplistic approach to a challenging problem, but it performs quite well in practice. Running entirely in Matlab, an admittedly inefficient environment, it is able to extract and display around 3 faces per second from a series of 320x240 color images. In practice, about 10% of the faces extracted must be discarded because they represent only partial faces, but I believe this is excellent "first pass" performance. This algorithmic approach can focus the attention of more complex and robust methods like the CMU approach, and in systems like mine, the recognition step

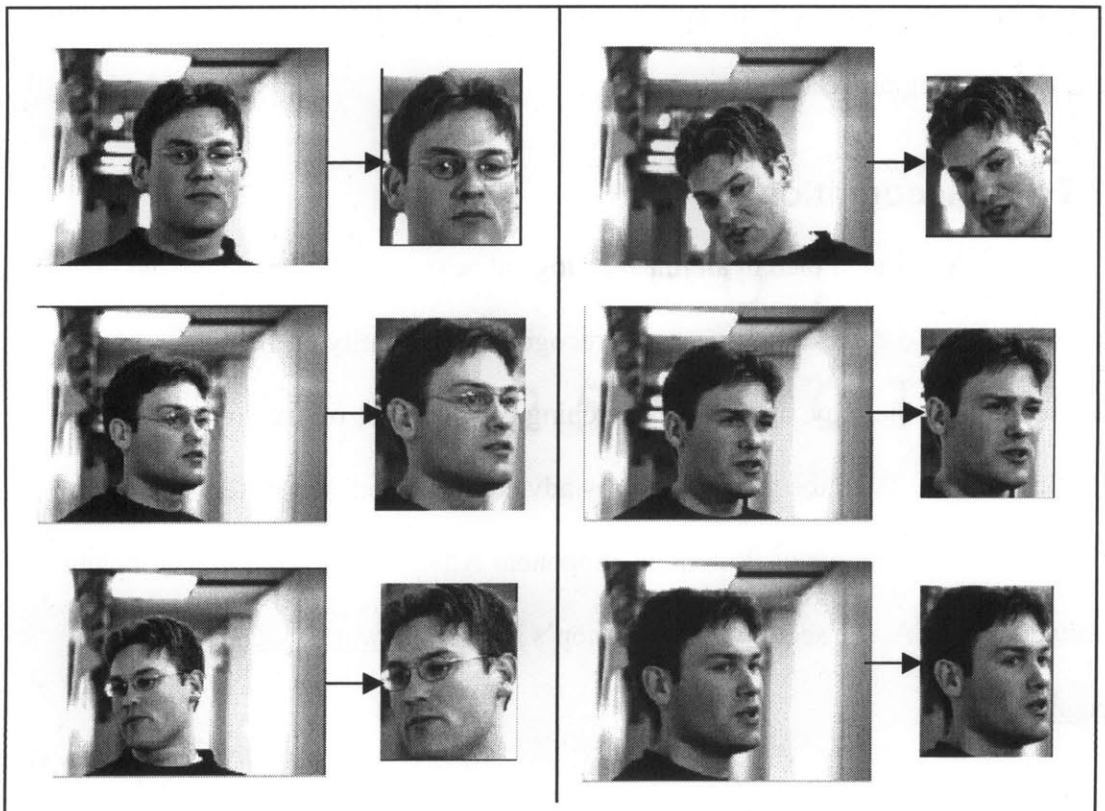


Figure 2-7: Typical performance of my face

can often be trusted to reject the partial faces. Figure 2-7 contains typical results for my face detector on a variety of poses.

One great benefit of a reliable algorithmic approach such as this is that it can be used to bootstrap many learning methods. For instance, a face database generation system can be set up in a novel environment, and begin by relying entirely on this algorithmic approach to find faces of pedestrians. A step following this algorithm can reject all images except those that are *certainly* faces, and attempt to cluster the acquired data into likely face classes. Eventually, the system can begin to attempt to recognize people using the created database. In addition, once a database is collected, a learning approach like the CMU system described above can be trusted with the task of detecting faces in the future if more accuracy is desired. This is an intriguing idea because the system starts "cold", without any painstakingly collected data or human intervention during the training process.

2.2 Face Recognition

After detecting a face in an image, a logical next step is to compare that face with a set of faces in a database and attempt to recognize the identity of the person. I addressed this problem by, again, first searching the literature on face recognition for promising results. With the guidance of my advisor, I located two systems that relied primarily on the technique of Principal Component Analysis (PCA). For a complete explanation of PCA, see section 8.6 of Bishop's Neural Networks and Pattern Recognition[8].

Pentland's Eigenfaces relies on Principal Component Analysis alone

In 1990, Alex Pentland and Matthew Turk published "Eigenfaces for Recognition," a paper describing a straightforward approach to face recognition using Principal Component Analysis [9],[10]. Faces are collected into a database (part of my

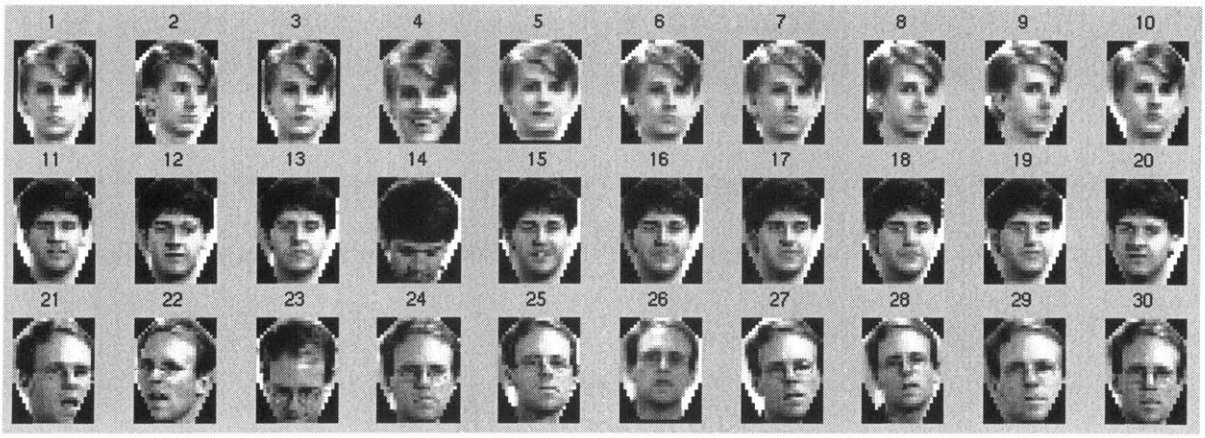


Figure 2-8: Part of my face database

database is shown in figure 2-8), and a singular value decomposition is performed to produce the eigenvectors with the largest eigenvalues, called "Eigenfaces" by Pentland. The first 20 Eigenfaces for my database are shown in figure 2-9. The matrix of these Eigenfaces can then be used to project the face database into a lower dimensional subspace of the originally very high dimensional space of the images. This is valuable because classifiers can work much efficiently in this low dimensional space. When a new image is presented to the system for recognition, it is projected into the low dimensional space using the Eigenfaces, and then a classifier (typically nearest neighbor) is used to determine the person's identity. It is worth noting that nearly all of the papers I read on face recognition use Pentland's system as a benchmark.

I decided to implement Pentland's Eigenfaces system because one of its chief strengths is that it can be used to quickly test if an image is likely to be a face. Since Pentland's algorithm works by determining the largest eigenvectors of the high-

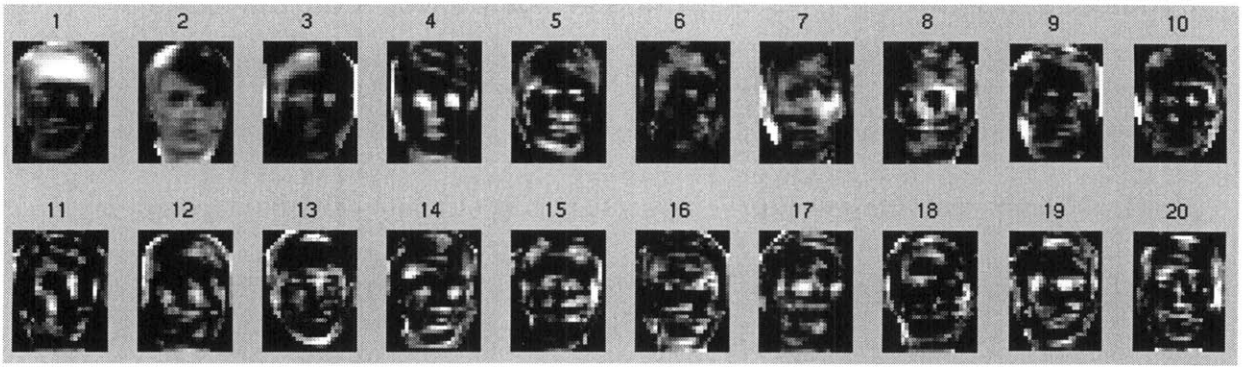


Figure 2-9: First 20 Eigenfaces for my face database

dimensional "face-space," these eigenvectors can be used to determine how "face-like" an image is using the following measurement from a later Eigenfaces paper [11]. For an image I, and eigenvector matrix W_{pca}:

$$dist = \left| I - W_{pca}^{T}(W_{pca} I) \right|$$

If *dist* is large, the image I is unlikely to be a face. I made extensive use of this feature of the eigenface system in the recognition system I created.

Bellhumeur's Fisherfaces extends Eigenfaces with Fisher's Linear Discriminant

In 1998, Pentland's system doesn't seem quite as impressive as it probably was in 1990. It's little more than ordinary Principal Component Analysis, and its shortcomings are quite clear. For one, it's very sensitive to changes in illumination. Also, it endeavors to spread the data as much as possible in the projected space, even if it ends up scattering data that should have stayed close together (ie, data that all belongs in the same class). The reason why is clear: PCA, the heart of Pentland's system, isn't thinking about classification when it does its job.

One paper that tries to make a direct attack at the root of PCA's problems in face recognition is "Eigenfaces vs. Fisherfaces: Recognition Using Class Specific Linear Projection" by Belhumeur, Hespanha, and Kriegman [12]. Belhumeur's system uses PCA

to reduce the dimensionality of the data set, but then applies the multiple class version of Fisher's Linear Discriminant in the PCA projected space to separate the data by class. I decided to first implement this system, and then build a classifier around it.

The Fisherfaces approach, like Eigenfaces, exploits a particular feature of Lambertian surfaces. When viewed from a stationary viewpoint under varying lighting conditions, Lambertian surfaces lie in only a three dimensional subspace of the very high-dimensional total image space. Faces aren't exactly Lambertian surfaces, but I won't go any further into these details here. Suffice it to say that faces are nearly Lambertian surfaces, and this feature makes a principle component analysis approach to this problem more effective [12].

Both the Fisherfaces and Eigenfaces approach to face recognition begin by solving the following PCA expression to maximize the total scatter of the data:

$$W_{pca} = \arg\max_{W} \left| W^T S_T W \right|$$

$$S_T = \sum_{k=1}^{N} (x_k - \mu)(x_k - \mu)^T$$

Since S_T, the *total scatter matrix,* is defined as the difference between every data point and the mean of the entire set, no class information is considered in the projection. Eigenfaces stops at this point and applies a nearest neighbor classifier to identify new faces in this projected, lower dimensional "face space." The Fisherfaces approach to face recognition, however, continues from this point, using the W_{pca} matrix in the next calculation, which is one step away from the Generalized Rayleigh Quotient:

$$W_{fld} = \arg\max_{W} \frac{\left|W^T W_{pca}^T S_B W_{pca} W\right|}{\left|W^T W_{pca}^T S_W W_{pca} W\right|}$$

Note that the solution to the above expression maximizes S_B, defined as the *between class scatter matrix*, while minimizing S_W, the *within class scatter matrix*. S_B and S_W are defined as follows:

$$S_B = \sum_{i=1}^{c} N_i (\mu_i - \mu)(\mu_i - \mu)^T$$

$$S_W = \sum_{i=1}^{c} \sum_{x_k \in X_i} (x_k - \mu_i)(x_k - \mu_i)^T$$

Where μ_i is the mean image of the class X_i, and N_i is the number of samples in that class. Intuitively, it makes sense for Fisherfaces to minimize within class scatter and maximize between class scatter. In order to project the data so that faces of a particular person lie close together, the projection should maximize the distance between examples of faces from different people while minimizing the distance between faces of the same person. The expression for W_{fld} above can be solved using a generalization of Fisher's Linear Discriminant for multiple classes, described in Duda & Hart, pages 118-120[13]. After solving for W_{fld}, Fisherfaces combines the two above expressions into the Fisherfaces projection matrix [12]:

$$W_{opt}^T = W_{fld}^T W_{pca}^T$$

This matrix is then used to project new data points into a low dimensional space, where a non-parametric classifier can be applied to classify the example. I chose to use a nearest neighbor classifier because it performs well with a relatively small face database.

Essentially, Fisher's Linear Discriminant is making the job for a classifier easier. For instance, a nearest neighbor classifier works by testing the distance between the point to be classified and every point in the training set. In a high dimensional space, (each face in my database has 800 dimensions) this is a very expensive calculation. However, in the Fisherfaces space, it is greatly simplified—the dimensionality is typically reduced to about 20.

Implementation of the Fisherfaces system required first implementing the Eigenfaces system, and then adding to it the particular optimizations of this method. I implemented the system in Matlab, and did some early tests on handwritten characters to ensure that my system was working. I then began looking for a face database to test my algorithm.

Unfortunately, the Sung database that I used while working on face detection was inadequate for a face recognition system. There is only one image of each person in Sung's database. He stretched this to three by rotating it slightly in two directions, but still, three images per person is too few to both train and test a face recognition system. I decided on the ORL (Olivetti Research Laboratory) face database for my tests because it had a large number of pictures (10) of each of its 40 subjects, compared with the other databases I was able to locate [14] (see figure 2-10). The poses in the pictures seemed to be both varied and reasonable, and were taken at different times under slightly varied lighting. Ideally, I would have used a data set with more variations in lighting to show

Figure 2-10: Part of the ORL Face Database

off the strengths of the Fisherface method, but none of the five data sets I was able to locate had both significant variation in lighting and a large number of pictures of each subject. The ORL database was available as a series of pgm files, which had to be loaded individually, downsampled, and built into a Matlab vector.

Bellhumeur's Fisherfaces approach performs very well in variable light conditions

I divided the ORL dataset into a training set consisting of seven pictures of each subject, and a test set consisting of three pictures of each subject. A nearest neighbor classifier in the Fisherface projected space achieved an error rate of 3.3%, missing only four pictures, which I've included in figure 2-11. Note the similarities in the backgrounds of these misclassifications, as well as in the faces themselves. The Eigenfaces approach surprised me—it peaked at 4.2% accuracy at 50 principal components, which is slightly

Image 15 mistaken for person 40.

Image 30 mistaken for person 38

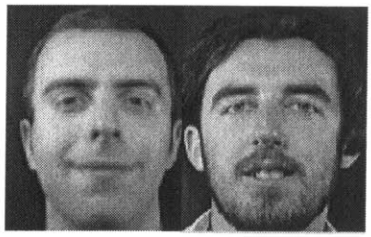


Image 56 mistaken for person 11

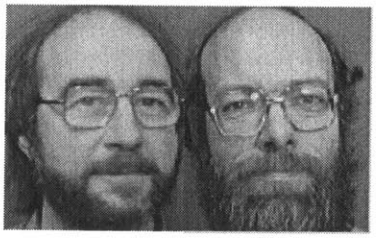


Image 82 mistaken for person 38

Figure 2-11: These are the mistakes made by my system on the ORL database. The left image of each pair was mistakenly identified as the person on the right

better than I expected, and much better than Belhumeur reported in his paper (8.5%). I believe this discrepancy can be explained by the fact that my data set had some variation in lighting, it did not have as much variation as the data set that Belhumeur used in his paper.

Like Pentland's Eigenfaces, Belhumeur's approach is centered around a set of projection vectors. When displayed as images, these vectors appear face-like. The Fisherfaces for my face database are in figure 2-12.

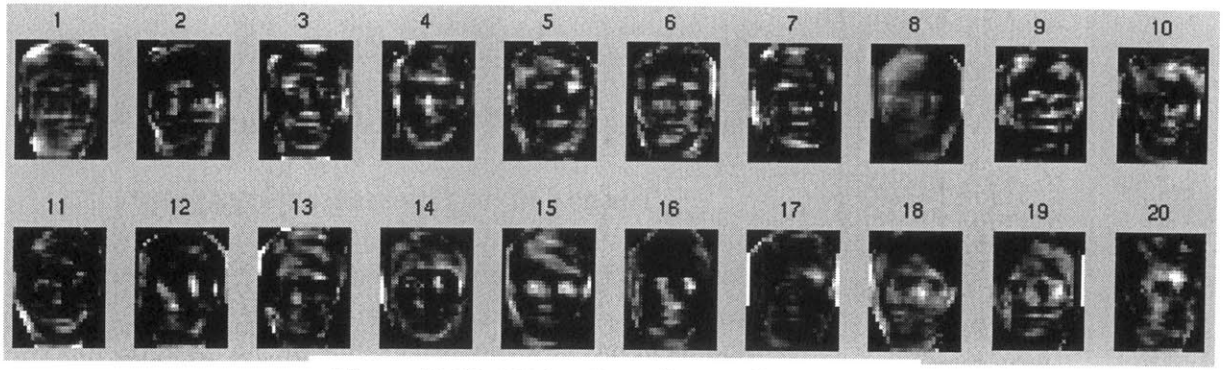


Figure 2-12: Fisherfaces for my face

Chapter 3

The Gatekeeper

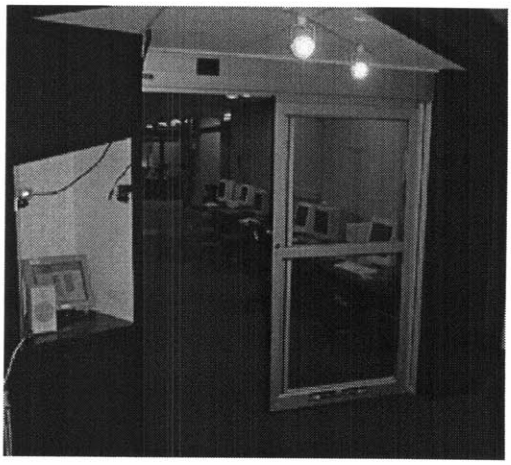


Figure 3-1: The Gatekeeper

Sometime in 1998, the AI Laboratory decided to renovate a large section of the seventh floor and fill it primarily with Eric Grimson's and Paul Viola's graduate students. The new space was intended to encourage collaborative projects and serve as an area where demonstrations of the group's work could be displayed in a common space. Among the features of this new space were a darkroom for light-sensitive vision projects, a large Steelcase grid structure that distributes power and network connectivity throughout the space, and an automatic sliding door that could be incorporated into some experimental work in people detection and recognition. I was given the opportunity to design and implement the first system for this automatic door, which I named the "Gatekeeper."

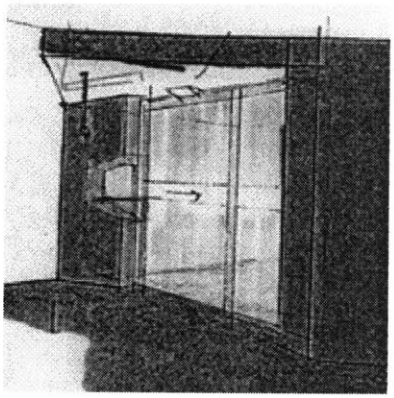


Figure 3-2: Architect's sketch

The goal of the Gatekeeper system was to demonstrate the results of my research in face detection and recognition while also establishing a platform for

further vision research in my lab. I was responsible for selecting and installing all of the necessary hardware as well as creating a system that would allow future researchers to easily control this hardware. In this section I give a detailed description of the hardware and software that make up the Gatekeeper.

3.1 Gatekeeper's hardware

Before I began writing software for the Gatekeeper, I focused my attention on designing a layout for the Gatekeeper's space and selecting appropriate hardware. The Gatekeeper system consists of an automatic door, two cameras, two door computers, spotlights, speakers, a touchscreen, and circuitry to control the door remotely. I combined these devices in a permanent installation at the primary entrance to the vision laboratory. This section describes the specific components that make up the Gatekeeper and the design decisions behind their selection.

The automatic door, touchscreen, and speech synthesizer provide a tangible interface to users

The Stanley automatic sliding door (quite similar to what you'd find at a supermarket) is equipped with motion sensors on both sides of the door. The door was designed so that the outside motion sensor could be disabled by toggling a switch inside the door. This proved to be a very useful feature because it allowed me to place the system in a mode in which the door would open only when my recognition system recognized a person standing outside.

I installed a touchscreen outside the door to provide visual feedback to and input from a user. The touchscreen was invaluable while debugging the system because it allowed me to monitor the system's performance at different stages of the algorithm in real time. For instance, while working on face detection I displayed the last detected face

on the touchscreen. This allowed me to gain an understanding of the scenarios where the system had difficulty detecting a face. The touchscreen was used extensively in the Gatekeeper competition, and I expect that future applications designed for the door will make greater use of it.

Finally, I wanted to allow Gatekeeper applications to greet and instruct users of the system using a speech synthesizer on the door control computer. This also turned out to be a great asset while working on my software for the Gatekeeper. For instance, when the system was acquiring pictures of my face for the database, it verbally informed me when it had stored a face and was ready for me to move to a different location in the environment. I also used the speech synthesizer to report recognition scores in real-time, allowing me to set the system's parameters more accurately.

Two cameras connect to video capture cards in an Intel PC

I considered a number of different cameras when setting up the Gatekeeper system. I wanted to use small cameras that were easily aimed, and provided good image quality. Eventually I decided on two Pulnix cameras that were large enough to support standard camera lenses. I believe that this capability contributes to the reconfigurability of the space because a change of lens can turn a pedestrian, full body camera into a face camera without having to move and remount the camera. In addition, auto-gain and auto-white-balance could be turned off on the Pulnix cameras, which greatly improved background subtraction performance. Both cameras are mounted on Bogen ball-and-swivel mounts that are screwed into aluminum wall mounts. I machined

Figure 3-3: Pulnix camera, mounted

the wall mounts to improve stability and allow the camera mounts to be removed and reattached to the walls many times without damaging the walls. Each camera is connected to a Intel Video Capture Card in an Intel-based Linux PC.

Figure 3-4: Bogen mount

The problem of where to place the aluminum wall mounts was perplexing at first. Many positions provided excellent views of the area outside the door, but were likely to be bumped by people walking through the door. I eventually decided on two positions inside the recessed kiosk to the left of the door because this position shielded the cameras while still allowing them to focus on a variety of positions in the environment.

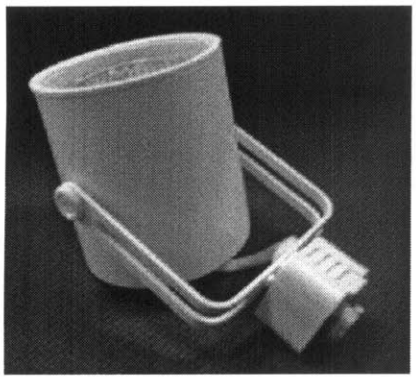


Figure 3-5: Juno track light

Adequately lighting the space proved to be a tricky task. The area was lit fairly uniformly from above by standard fluorescent ceiling lights, but these lights didn't adequately illuminate the faces of people approaching the door. To improve the lighting, I installed two Juno track lights in a track on the ceiling, and focused them on people approaching the door. Even with these lights, there remained places where a person's face was still somewhat shadowed. A light source attached to the wall in some of these locations would likely correct the problem.

The door control interface is a serial-port controlled relay

The Stanley automatic door has a touchplate inside the door that opens the door if the motion detectors fail. In order to allow a Gatekeeper to open and close the door, I decided that I needed a device, controllable from a computer, that would allow me to open and close the touchplate's circuit on command. It's quite remarkable, actually, that devices like this are very hard to find—nearly every one I found was very expensive and had many features that I was not interested in. Finally, however, I decided on the 232DRIO serial-port controllable relay switch from B&B Electronics. The device listens for several particular sequences of characters on the serial port, and opens or closes one of two circuits according to the content of the signal. I connected this device to one of the two door computers and wrote a server to allow multiple users to open and close the door remotely.

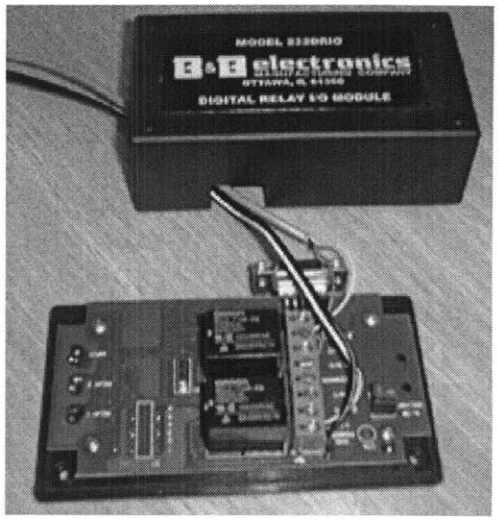


Figure 3-6: The 232DRIO Digital Relay

3.2 Gatekeeper's software

The goal of the Gatekeeper's software is to allow users to write vision-based door control applications in a variety of languages with little effort. The basic design of Gatekeeper's software is summarized in figure 3-7. This goal can be separated into two separate sub-goals:

1. Provide an intuitive, multi-user interface to the door hardware.
2. Facilitate the implementation of applications that depend on real-time video.

I addressed the first goal by creating the Doorman server, a multi-threaded Java server that controls access to the touchscreen, speech synthesizer, and door. Clients communicate with the Doorman by making a socket connection to the server and sending

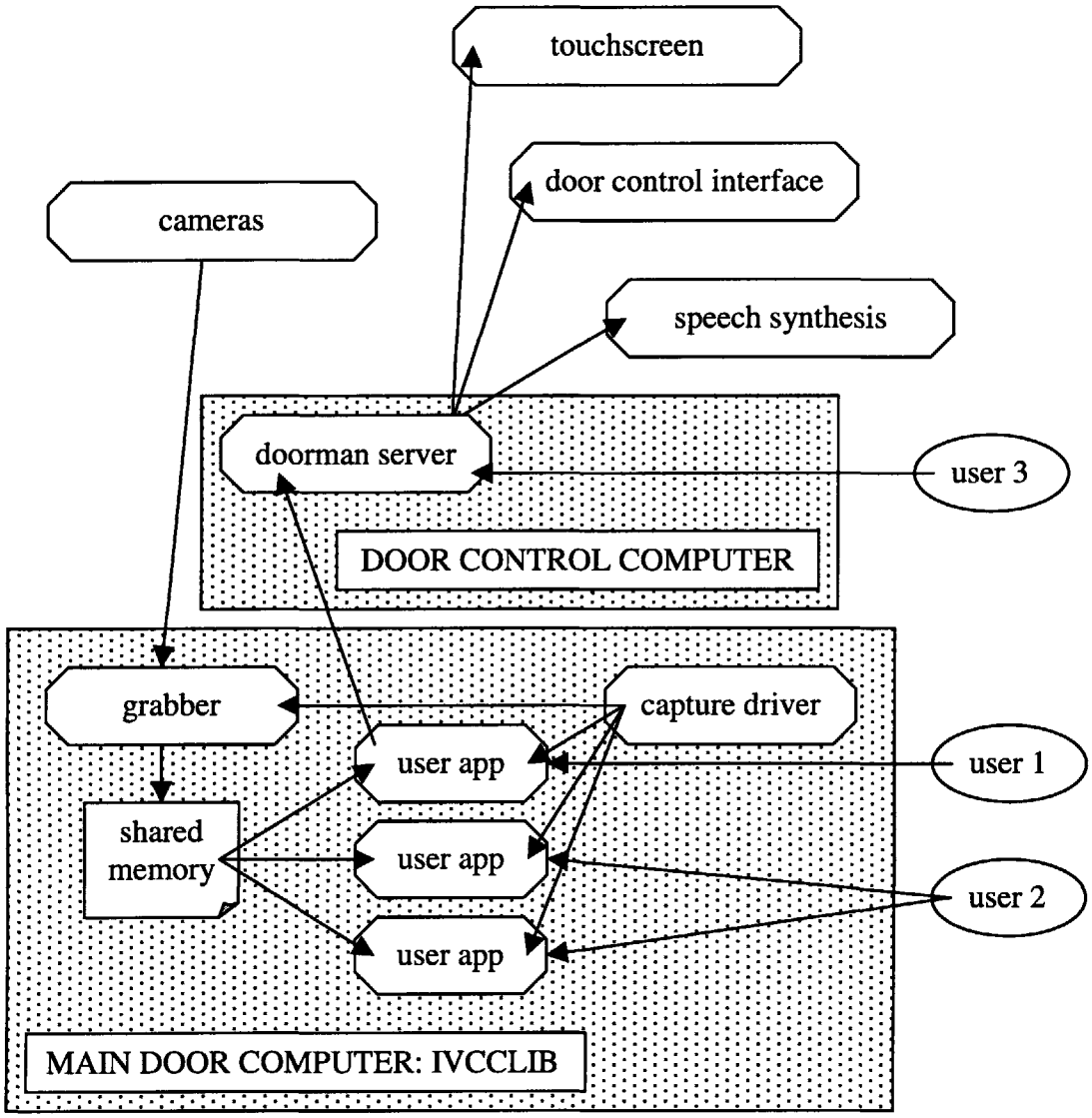


Figure 3-7: Gatekeeper's software

simple text commands. I implemented Doorman as a separate piece of software for two reasons. First, by requiring all door open/close requests to go through a single server, it was easy to control who was allowed to send these commands and resolve conflicts when multiple applications sent different signals to the door. Second, by isolating all of the door-specific software in a single server, I was able to build the rest of Gatekeeper's software into a general set of vision research tools that have been useful to other members of my lab.

These tools, called IVCCLIB (Intel Video Capture Card Libraries), consist of three parts:

1. Modified drivers for the Intel Video Capture Card, which make it possible for multiple users to simultaneously use the device.
2. The grabber program, which handles all of the details of capture real-time video, and makes the images available to multiple user applications simultaneously.
3. A collection of C, Matlab, and Java functions that allow users to easily write vision applications in several languages.

One of the chief goals of IVCCLIB was to make it possible for multiple applications to access the real-time video captured by a single capture card simultaneously. This is a valuable feature for several reasons. First, capture cards aren't cheap—any system that avoids buying a new card for every person who wants to do vision research is certainly beneficial. Second, many users may want to access images from an existing camera in a particularly useful place, as is the case with the Gatekeeper door cameras, the Virtual Viewpoint Reality 12 camera reconstruction space, or Chris Stauffer's window cameras [15], [16]. It shouldn't be necessary to setup additional cameras and computers every time a new user decides to do research in that space, and users shouldn't have to worry that they are inadvertently "blocking" other users' access to these resources. Finally, a single user may want to access the same real-time data from multiple applications simultaneously, for different purposes. For instance, Chris Stauffer runs pedestrian tracking programs on the window cameras for long periods of time. Using a typical capture card, when one of these programs is executing, he can't use the capture card for any other purpose. However, using IVCCLIB, he could start an additional application to record a few frames for offline analysis, or display what was

being captured, or even start a second tracker and have the two programs communicate with one another.

Another interesting feature of IVCCLIB is the way video capture is accomplished. Specialized "grabber" programs run on every IVCCLIB machine constantly, capturing video and providing frames to user applications when they request them. This means that there is no capture card "startup time" when a user decides to start using data from a particular source. In fact, a user can quickly change between several sources with very little overhead. It's not difficult to imagine every computer in the vision lab equipped with a camera, constantly capturing data and providing it to a researcher whenever it is requested.

I have created an extensive website on IVCCLIB to serve as a respository for the IVCCLIB software and documentation of the system. Much of the material on this website is included in Appendix B.

The Doorman server provides multi-user access to door controls and speech synthesis

The doorman server is written in Java and runs on a Windows NT machine. The server listens for connections on port 9790. When a client connects, the server spawns a thread to handle the requests made by that client. Once connected, the user uses the protocol specified in Appendix A to:

1. Open or close the door.
2. Cause the door to say any text aloud.
3. Display a webpage on the touchscreen..

Because the interface with the door is a simple socket protocol, the door can be controlled from any application. Since the server is multithreaded, many users can be connected to the doorman server simultaneously.

Multi-user control of a door has an obvious complication: If one user sends a command instructing the door to open, and another sends a command instructing it to close, what should the system do? Furthermore, if a program in development goes haywire and begins opening and closing the door as often as possible, people trying to enter and leave the lab are likely to become a bit annoyed. To avoid these problems, users aren't usually allowed to actually open and close the door. To provide them with visual feedback while they are testing the system, however, the touchscreen displays a list of the users currently connected to the doorman server. Beside their name is a light that is red if the user's last door request was to close the door, and green if the user's last request was to open the door. (see FIGURE) This allows a user to start their application, walk to the door, and see when their program requests that the door open or close. During the development period for the Gatekeeper competition (described in the next chapter), there were often four people working on the door at once, making this feature very valuable.

Several modifications to the capture card driver allow the grabber processes to supply images to multiple simultaneous users

In this section and the next, I describe parts of IVCCLIB, the Intel Video Capture Card Library, which I wrote to assist me in my work on my thesis. IVCCLIB has grown into a set of tools that aid a researcher in writing applications that need to capture video in real-time. It consists of modified drivers for the capture card, a grabber program which handles all of the details associated with real-time video capture, and a set of tools that facilitate the development of vision applications.

In order to take a video capture card that was intended to support a single user and make it accessible to multiple users, I had to solve two problems. First, the images had to

be placed somewhere in memory that was accessible to multiple applications, and second, the card driver had to be modified so that it signaled all of the active vision applications when a frame was completed. In addition, the modifications I made needed to be backwards compatible, that is, I wanted the capture card to function exactly as it had for applications written before IVCCLIB, but provide all of the above additional functionality when it was desired.

The first challenge was making the images captured from multiple capture cards accessible to multiple applications that were each written in different languages. To accomplish this, I wrote a grabber process which runs constantly in the background, taking images from every video capture card in the computer and placing them in separate POSIX-style shared memory segments. Since the capture cards use direct memory access, the grabber program requires almost no processor time, and is streamlined to require very little memory beyond that which is required for storage of the current image from each card. Consequently, it is able to capture 30 frames per second on several capture cards simultaneously without any noticeable slowdown of the system. The grabber process can be thought of as an image postman—it delivers images from all of the capture cards to mailboxes in memory. IVCCLIB applications running on the system check these mailboxes when they receive a notice stating that a new frame is in a mailbox that they previously expressed interest in.

The next problem, clearly, is informing all of the IVCCLIB applications on the machine that a new frame is waiting in the "mailbox." I accomplished this task by modifying the driver for the capture card. Fortunately, it was already designed to send a POSIX-style signal to one program that is using data from the capture card. I modified

the driver so that it maintained a list of IVCCLIB applications and their process identification numbers. When a new frame arrived, the driver sent a signal to each of these processes. The most difficult problem that arose was that these applications could not be counted on to inform the capture card driver when they exited. Since the capture card code runs in the kernel, sending a signal to a nonexistent process causes a serious kernel error, and must be avoided. A large number of the modifications I made to the driver are for the purpose of carefully keeping track of when IVCCLIB applications start and terminate.

IVCCLIB facilitates development of vision programs in several languages

An unfortunate characteristic of systems developed as a part of a student's research is that, since he works alone, he is the only person that really understands his work. Consequently, when he leaves, no one can use the products of his research without considerable support from him. I was determined to leave part of my work behind in the form of an API that made it easy to write vision applications for nearly any purpose, in several languages. The languages I decided to support first were C and Matlab. Recently, Mike Ross, another student in my group, has written code to make the IVCCLIB system accessible to Java programmers as well.

In the last section, I described the grabber programs, which are responsible for delivering images to a shared location of memory, and the method by which applications are informed that a new frame is waiting in that location of memory. All that remains is an easy way for users to access that location of memory, independent of what language they choose.

Rather than requiring the user to access and manipulate the section of shared memory that holds the captured image themselves, I chose instead to use a callback

model. The user begins by writing a callback function that takes as an argument the most recently captured image. A grabber is then "attached" to the callback by calling a function that is part of the IVCCLIB. Once the grabber is attached to the callback, it begins sending it frames of video as they becomes available, taking care to format them so that they are recognized as an image in their language. For a detailed explanation of how to write callbacks in IVCCLIB, see Appendix B.

I concentrated on making IVCCLIB accessible from Matlab because I've found that Matlab is an excellent tool for vision research. It offers a huge library of functions that reduce many difficult image analysis tasks to single line commands. It is also a comparatively easy environment to learn, which fit with IVCCLIB's goal of making it possible for a researcher without a lot of programming experience to start writing vision applications. I first concentrated my efforts on making it possible to capture video into Matlab at any speed. My earliest approach actually used a second program, written in C, to capture the data and write it to a sequence of files on disk. This program also wrote an information file to inform Matlab which file in the sequence contained the most recent frame. I was able to capture video into Matlab this way at only two or three frames per second, but this first success drove me to find a more efficient solution.

What was needed was a way for Matlab to directly access the images placed in memory by the IVCCLIB grabber, like an IVCCLIB application written in C. Fortunately, Matlab does allow its users to directly call native C code by compiling it into a "mex" file. One way to proceed would have been to write a C program that captured one frame of video and returned it to Matlab. However, there is some overhead associated with starting up that C program from Matlab, so this "one-shot" approach is

inadequate. Instead, a user begins by writing a callback function that takes as an argument the most recent captured frame of video. When they are ready to test their callback, they call a piece of C code from Matlab that begins passing images captured by the grabber into their callback. Since this C program only has to be called once, the aforementioned overhead is avoided, and a capture rate of 30 frames per second is achieved.

Chapter 4

Applications

After developing systems for face detection and recognition, building the Gatekeeper, and developing IVCCLIB, I was eager to use these components in an application to test their performance and find areas that needed improvement. I had planned from the beginning to combine the face detection and recognition systems to transform the Gatekeeper into a doorway security system, but finding an application to test IVCCLIB was more challenging. I needed to generate enough interest to convince multiple people to use IVCCLIB for a project, so I could evaluate its performance in a multi-user environment. The MIT AI Olympics proved to be the ideal setting.

The Gatekeeper competition tested IVCCLIB and illustrated its ease of use

MIT students don't have classes during the month of January. This month is devoted to IAP, the "Independent Activities Period." Each year, at the end of IAP, the MIT AI Laboratory holds the annual AI Olympics. The Lab is organized into four teams which compete in a variety of events ranging from basketball to coding competitions. For the 1999 AI Olympics, I designed a coding competition for the Gatekeeper system. In the Gatekeeper event, each team was responsible for writing a program that opened the door for members of their team, but not for members of other teams. Each team was

provided with an early version of IVCCLIB that allowed them to capture images, open the door, and control the speech synthesizer and touchscreen.

A few rules were specified to keep the competition interesting and fair. First, the only real-time input that a door control program could accept were images from the camera. In other words, it was against the rules to sit a member of your team at a nearby computer whose job was to press enter whenever one of your team members approached the door. People were allowed to carry props—a picture or a light for instance, but a

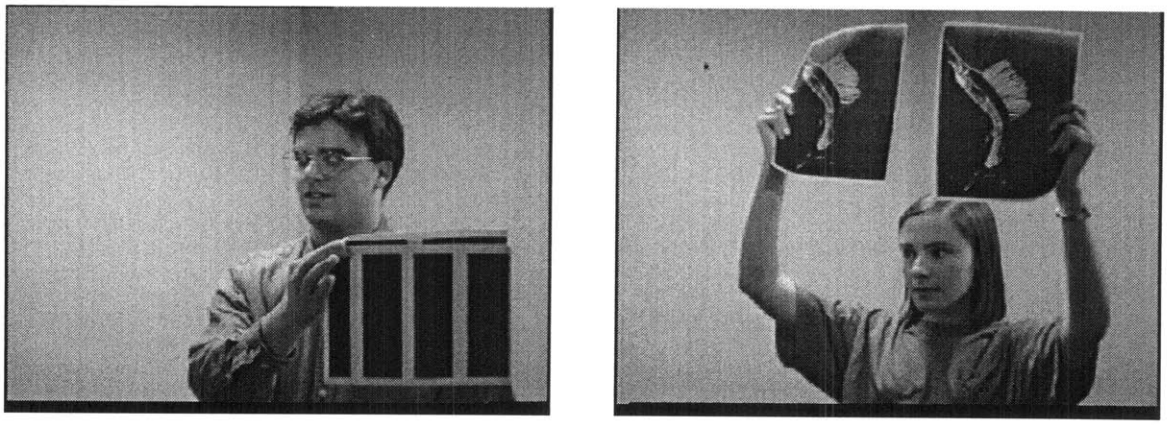


Figure 4-1: Competitors approach the barcode and clockwork entries

particular prop could only be used once. Finally, teams were given a large bonus whenever they identified a person by name, but they were told in advance that people would be required to approach the door in a random order. The exact scoring system is as follows:

Gatekeeper admits a team member	2 points
Gatekeeper admits an opposing team member	-1 point
Gatekeeper identifies someone by name	2 points

In addition to providing a fun Olympic event, I designed the competition to help me evaluate the current state of IVCCLIB. I paid close attention to the progress of each

of the teams, particularly to tasks that seemed to be more difficult with the current implementation of the system. It quickly became apparent that users of the system wanted a higher framerate than the 5 frames per second capture rate in their version of IVCCLIB. This fact led me to make the modifications described at the end of Chapter 3 which made 30 frames per second video capture in Matlab possible. In general, however, IVCCLIB fulfilled the demands of its users, allowing multiple clients to work on code for the door simultaneously.

The competitors surprised me with their ingenuity. One team built a system that only admitted people holding a piece of paper with a particular barcode on it. Another implemented a complex challenge and response system that required users to spell out a code by positioning their arms in a clockwork fashion (see figure 4-1). In the end, though, the team that achieved the best results used a much simpler approach—they only admitted people holding a particular plastic tray that was slightly translucent. The competing teams were unable to find a prop that could fool the system quickly enough, so the simple solution won the competition (see figure 4-2).

Figure 4-2: The winning entry and a valiant attempt to fool it

A Vision-Based Security System provides some access control in a high traffic area

After the AI Olympics competition, I invested my time in refining IVCCLIB. At the beginning of April 1999, I felt that IVCCLIB was at a stage where I could use it to integrate all of the pieces of my research up to that point into a working door security system. The system I designed uses the algorithmic face detector described in Section 2-1 to locate a face in an image from one of the door cameras, and then uses the Fisherfaces recognition system described in Section 2-2 to attempt to identify the face. If the person is recognized, the door is opened and the person is greeted by name.

Before my system could begin recognizing people, I needed to construct a face database to support my recognition algorithm. To expedite this task, I built a tool around my face detector that greatly simplified the process.

The first system I built used the algorithmic face finder described in Chapter 2 to guess at the location of a face, but then allowed the user to more precisely center the

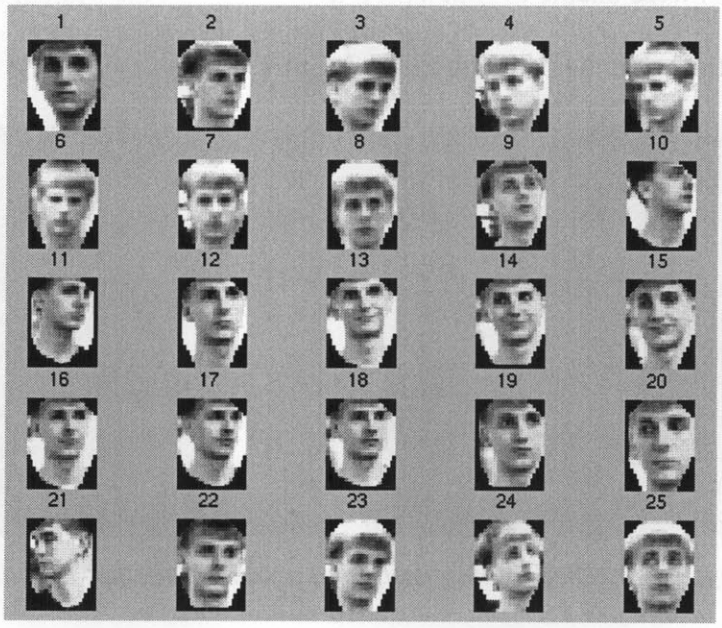


Figure 4-3: Interface to the face database generator

image using a keyboard interface before the face was stored. At first, this seemed to be the best approach, because the algorithmic face finder occasionally grabbed only a portion of someone's face, or missed it entirely. However, when testing the recognition system with this data, I wasn't impressed with the recognition accuracy. I deduced that the problem was that the hand-centered data was significantly different from what the system was likely to see in operation. In retrospect, it seems that I cropped the faces more tightly when I refined the faces by hand than the automatic system did in practice.

The final database generation system didn't allow a user to perform the hand centering operation. Instead, the system grabbed far more faces than was necessary for the database, displayed them all to the user, and allowed the selection of the best images of subjects' faces (see figure 4-3). This avoided the introduction of bad data resulting from mistakes made by the face detector, but ensured that the data entered into the database was more similar to what the system was likely to see in operation.

A clear extension of this system would be to follow up my face detector's guess with a refinement algorithm, and remove the user entirely from the process of data acquisition. This would produce the added benefit that the system could build a face database nearly autonomously.

I wrote the door security system entirely in Matlab using IVCCLIB. Since the face detector and recognizer were both written in Matlab themselves, the system followed these steps:

1. Pass the image through the algorithmic face detector.
2. If a possible face is detected, first measure its distance from the entire face database's space. If this distance is larger than a certain value, discard the face. Otherwise, pass it to the Fisherfaces recognizer.

3. The recognizer will return the most likely identity of the face, and a score, where a lower value indicates a better match. If the score is above a certain threshold, discard the face. Otherwise, greet the user and open the door.

Steps 2 and 3 deserve further explanation. In Step 2, my system attempts to reject incorrect detections made by the face detector, such as partial faces or backs of peoples' heads. To accomplish this, I use the "distance from face space" metric discussed in Pentland and Turk's Eigenfaces paper [9] and summarized in Chapter 2. Since this metric only measures the distance between the input image and the space of all of the faces in the database, this metric is only able to reject non-face images. Faces of people not in the database are likely to slip through. However, in Step 3, the system returns the most likely identity of the face, and a recognition score. Using this score, faces of people not in database that passed the first metric can be rejected because they will not match very well with any particular person in the database. In general, a face of someone not in the database will typically be close to the face space as a whole, but not very close to any particular class.

To test the system, I gathered data from three people: my advisor, my officemate, and myself. The system performed quite well on this small set of people—it identified us by name, and rejected most people not in the database. Execution time was acceptable, but could be improved greatly by moving some of the more time consuming tasks from Matlab into C.

Chapter 5

Conclusions

It's been a short eight months since I began work on my first neural network for face detection, and only three and a half months since the automatic door was installed on the seventh floor of the Artificial Intelligence Laboratory, but it's already time to draw together a few reflections on the lessons I learned during my thesis. At its conclusion, my thesis consists of a complex automatic door setup, a working door security system, and an extensive set of libraries to support further vision research in my group. In the process of completing these products, I learned as much from what didn't work as I did from the successful systems. Also, I uncovered new questions and possible research directions along the way that I hope will be a part of future work on the Gatekeeper.

Unpredictable complications exist in a real-time environment

I expected my algorithms to behave differently in varied environments, but I did not anticipate the extent of the consequences associated with moving an offline system into a real-time, realistic environment. The transition is akin to tossing a plastic model airplane off the roof of a building. For instance, I found that the CMU approach to face detection is very effective when applied to still images, but is poorly suited to real-time video streams. While working on the drivers for the Intel Video Capture Card, I

addressed several tricky synchronicity issues that I could have avoided if I hadn't expected the system to run in real-time. The Doorman server seemed perfect until six people connected to it and all started telling it to say different things all at the same time.

I also couldn't predict exactly how people would respond to the system, and how this would affect its operation. For example, some people approached the system in interest, but ended up too close to the camera for the system to function properly. Others immediately set about fooling the system by trying to alter their appearance to match a person in the database.

On a similar note, I learned a valuable lesson on the importance of using realistic data to train a learning-based system. After training my recognizer on face data centered by hand, I was pleased with the results it achieved when I tested it on more of my hand-centered data. However, its performance in a realistic environment was terrible, because I no longer played a part in the system's input loop. I now look with a skeptical eye upon papers that report results using idealized data—for instance, face recognition systems for which all of the faces are evenly lit and neatly centered on a white background. It's not surprising that these systems perform well. They're solving a problem far simpler than true face recognition, and would probably find the performance of their system in a realistic environment quite dismal.

Development of vision applications is facilitated by IVCCLIB

One of the specific goals I set out for myself was to design software for the automatic door that would allow other researchers to use the door as a research tool in the future. I believe IVCCLIB helps to fulfill that goal by serving as an excellent framework for any machine vision research application. Since I completed IVCCLIB a few weeks ago, it has been installed on nearly every machine in my group, and three students are

actively using it in their work. While my door security system was intended to be a proof-of-concept, I consider IVCCLIB to be a complete system.

Facial recognition systems raise ethical questions

In Chapter 1, I described the Mandrake system, a face recognition system connected to 144 CCTV cameras dispersed throughout the city of Needham, Scotland. The system has been credited with helping police in Needham track the movements of known criminals, and is widely supported by the inhabitants of the city. Still, even considering all of its positive applications, it's impossible to ignore the many ways such a system could be misused. It would be fairly easy to use a system like this to track the movements of all the citizens in a small town, and use this information to create every Orwellian nightmare imaginable.

Behind this morbid possibility is a deeper, more philosophical question. Is it a violation of a person's right to privacy to identify them without their permission? Our current laws seem to say "No." There are already laws in place that make it completely legal to photograph or videotape any person in a public place, and it isn't considered it a violation of someone's rights if a photographer publishes a photo with the name of that person in the caption. Why, then, does it raise concerns when the process is automated by a face recognition system? Clearly, part of the reason is that face recognition systems are completely passive—they require no permission from the person being identified and, afterwards, the person remains unaware. I believe that it is the fear of an unknown, invisible, observing force that drives most of the concerns about face recognition systems.

Personally, I believe that just like nuclear science and genetics research, face recognition has many positive and negative applications. As long as the positive

applications remain compelling, research in face recognition should continue in earnest. A cautious eye should follow systems like Mandrake, however—I don't believe that many positive results can come from citywide surveillance.

Gatekeeper will serve as a foundation for further research

Now that the Gatekeeper hardware is in place, and software is written to make it easy to control, the stage is set for some exciting new applications. I would like to see more interactive systems that make more use of the touchscreen. For instance, at night, the Gatekeeper could require that a code be entered at the touchscreen in order to gain entrance to the lab. Or, a lab directory could provide directions to lab visitors, and follow them with additional cameras, giving additional instructions as necessary. It's not difficult to imagine expanding this system into an automated tour of the vision laboratory, in which visitors are tracked from one camera to another while being introduced to ongoing projects, and even lab members!

In the area of vision-based security, I believe that the most interesting applications will not be high-security identification systems, but systems that improve security through more clever means. For example, I believe that a system that simply recorded the faces of every unrecognized person would be valuable to a place like the Artificial Intelligence Laboratory. If equipment were stolen, the police could be provided with pictures of every unrecognized person that came into the lab on that day, which they could compare with a database of people with a criminal record. Alternatively, a system could greet unrecognized people and offer to direct them to the person they are visiting. If the person then did not follow the directions given by the system, their face could be recorded and flagged for later review. A vision system could be taught metrics for "suspicious activity," such as the movement of large items out of the lab, or large groups

of people arriving late at night, and notify a security guard to investigate. All of these systems will be much easier to create now that the Gatekeeper system is in place.

All of the applications I've suggested above, as well as my thesis itself, endeavor to teach computers an essential human skill: the ability to recognize people and interact with them successfully. Computers with these abilities will not only revolutionize automated security, but will be highly useful tools to their users. Progress in this field will enable researchers to turn their attention to more advanced systems and further chip away at the barrier between computers and their users.

Appendix A

Doorman Server Protocol

The doorman server runs on port 9790 of doorman.ai.mit.edu. To use the server, a client makes a standard socket connection to the server, and sends commands in ASCII format, terminated by carriage returns. When a client connects, he is given this prompt:

`GK:`

He should then enter a username for the server, followed by a carriage return. At this point, he is logged onto the server and can send the following commands.

Commands ARE case sensitive.

Command	Description
OPEN	Open the door
CLOS	Close the door
SAY *text*	Using the door speech synthesizer, says *text.*
HTML *url*	Sends the door touchscreen to *url. Url* should be of the form http://www.ai.mit.edu/
CONTROL	Put the door in "live" mode, that is, OPEN and CLOS commands will actually open and close the door.
SIMULATE	Take the door out of "live" mode.

Note: a user should pause between successive "SAY" commands for about a second. Otherwise, one of the "SAY" commands could be lost.

Appendix B

IVCCLIB DOCUMENTATION

This material is from the website I created for IVCCLIB, which is located at http://www.ai.mit.edu/projects/lv/ivcclib.

Contents:

Introduction

IVCCLIB works differently than most video capture environments. An IVCCLIB program doesn't actually get its images from the capture card. Instead, it gets the images from the computer's memory, where they have been stored by a grabber program (section 3). So, before any IVCCLIB program can run, a grabber program has to be started to provide it with images.

Because the grabber program is taking care of moving the images from the capture card into memory, multiple IVCCLIB programs can use the captured images simultaneously. This requires that all IVCCLIB programs be "good citizens", that is, they should never write data into the shared memory location because other programs may be

trying to view that data. I am not preventing IVCCLIB programs from doing this because I believe there may be situations when a researcher might want a program to write into that shared memory section.

A few notes on terminology: Because of their mostly read-only nature, IVCCLIB programs tend to fit the title "Observer." I will use this name to refer to most programs written by users using IVCCLIB. I often use the acronym "IVCC" to stand for Intel Video Capture Card.

This document will walk you through the steps you need to follow in order to set up IVCCLIB and start writing observers. It's only necessary to read all of sections 5,6, and 7 if you plan to use IVCCLIB in Java, C, AND Matlab. Otherwise, I encourage you only to read the section on the language you plan to use.

Installing IVCCLIB's drivers

IVCCLIB requires a particular kernel patch that allows the capture driver to allocate a large section of memory for framegrabbing. This patch is called "bigphysarea", and is available at: http://www.uni-paderborn.de/fachbereich/AG/heiss/linux/bigphysarea.html. Be careful-- patching a kernel isn't easy-- don't hesitate to get help on this!

After the patch is set up successfully, the drivers for your IVCC need to be installed. Note: I didn't write the IVCC drivers-- they are a heavily modified set of drivers that have been passed around from researcher to researcher, and I believe started as Matrox Meteor drivers years ago. I merely made several modifications to the driver to support IVCC's multiple observer feature.

The drivers for IVCCLIB are on the software page, and must be installed with the command "insmod." Insmod must be run by root, and only installs the driver in the

current boot-session of Linux. In order to make sure your driver is loaded everytime the system boots, I suggest you copy it into /etc/rc.d and modify your rc.local file to load the driver at system startup. The following steps should be made as root:

First, move the driver file into the correct directory.

```
cp bt848.o /etc/rc.d/
```

Next, add the following to the end of /etc/rc.d/rc.local

```
/sbin/rmmod bt848
/sbin/insmod -f bt848
```

You then need to make the device files in /dev. The number of device files you should make depends on how many IVCC's you have in your system. The following commands make the necessary devices for 4 IVCC's (yes, it's possible to have 4!) Again, you need to be root to execute these commands.

```
mknod /dev/bt848 c 82 0
mknod /dev/meteor c 82 0
mknod /dev/bt848-0 c 82 0
mknod /dev/bt848-1 c 82 1
mknod /dev/bt848-2 c 82 2
mknod /dev/bt848-3 c 82 3
mknod /dev/bt848-4 c 82 4
chmod a+rw /dev/bt848*
chmod a+rw /dev/meteor

mknod /dev/mmetfgrab0 c 82 0
mknod /dev/mmetfgrab1 c 82 1
mknod /dev/mmetfgrab2 c 82 2
mknod /dev/mmetfgrab3 c 82 3
mknod /dev/mmetfgrab4 c 82 4

chmod a+rw /dev/mmet*
```

Congratulations! After a restart, your system should be ready to start capturing video.

Starting the IVCCLIB grabber program

As explained in the introduction, IVCCLIB observers don't actually get their images from the capture card. A special grabber program puts images from the capture card into a shared memory location, and the IVCC driver notifies the observers when a

new frame has been placed there to allow synchronization. Before you can write an IVCCLIB program, you have to sucessfully start the grabber program. It's also part of the downloads on the software page.

Once you've downloaded it, I suggest you place it with the card driver in /etc/rc.d/ and modify rc.local to start it every time the system starts-- you can always locate it using ps and kill it if you need to restart it.

The arguments for grabber are as follows, and will be displayed if grabber is run without any arguments.

```
grabber <# of IVCC's to be managed> <Source # of first
IVCC> <Source # of second IVCC> ... <Brightness> <Contrast>
```

Source numbers should be either 0, 1, or 2, corresponding to RCA input, SVIDEO, or INTEL-CAMERA input, respectively. Brightness should be between # and # and contrast between # and #, typically. Brightness and contrast are set to the same value for all IVCC's.

Executing this command:

```
grabber 2 1 0 50 50 &
```

Should result in an output that looks like this:

```
Thread 1 started ...
Digitizer /dev/bt848-0 is online, capturing from source 1.
Thread 2 started ...
Digitizer /dev/bt848-1 is online, capturing from source 0.
```

Again, I suggest you put the grabber startup command in your rc.local file similar to the way you placed the insmod command in the rc.local file above.

You can only load one grabber per IVCC-- there's really no reason to want more than one. If you need to kill the grabber process, find its pid by typing something like:

```
ps auxww | grep grabber
```

Then kill each copy of the grabber manually with the "kill" command.

Basic structure of an IVCCLIB program

After you successfully install the IVCCLIB drivers and start the grabber process, it's time to try writing an IVCCLIB observer. You have the choice of writing your observer in C, Java, or Matlab, but whatever you select, you program will tend to take on the same structure.

All of the real work in an IVCCLIB program gets done in its capture callback. The callback is a function that is called every time a new frame is captured (30 frames per second), unless the callback has not completed its last execution. In this case, the callback is not called, but will be called the next time a frame is delivered and the callback is idle.

So, when you write an IVCCLIB observer, you spend 99% of your time writing its callback, and a tiny amount of time writing the code that informs the IVCC that your callback is interested in captured frames. This was the goal of IVCCLIB-- to let you spend your time writing processing code instead of wasting your time with the details of getting captured frames into your program.

Deciding what language to use for your observer is really a matter of personal preference. I am a big fan of the IVCCLIB Matlab environment-- I used it for nearly all of my masters thesis. If you can believe it, IVCCLIB allows 30 frames per second capture into Matlab through the mex gateway, and Matlab is great for vision research if you have the image processing toolbox. Plus, the mex gateway lets you slowly move your Matlab code out into C if more performance is desired. IVCCLIB in C is great if you're looking for high performance code, but certainly takes more work to get something working than Matlab. IVCCLIB in Java is a new arrival, and doesn't offer high

performance at all. However, considering that there aren't many capture libraries in existence for Java, it's a step in the right direction!

Using IVCCLIB in Matlab

I believe that Matlab is the best environment for working in IVCCLIB. Matlab offers a large number of functions that will make it possible for you to accomplish complex tasks with a single command, especially if you pick up a few toolboxes as well. You can capture 30 frames per second of video in Matlab (no kidding!), so it's efficient enough to get you started on a real system. Here's how to get started:

Follow the instructions above to set up your machine with the IVCCLIB drivers and the grabber program. Then, download the IVCCLIB Matlab files. Here is a list of the files and what they are useful for:

showframe.m	This is a sample callback that simply displays the video as it is passed into Matlab, as quickly as possible. You should make a copy of this file and start editing it. Most users will never have to edit anything but that file.
attachgrabber.m	This is the command that starts the IVCCLIB Matlab system. You shouldn't need to read or edit this file unless you want to do something really different.
m_observer.c	This is the C file that handles the passing of frames of video into a callback, like showframe.m above. You may want to edit this file if you're after high performance code-- you can do work on the image before you pass it into the Matlab callback-- you can even edit this file so that it edits the image, passes it into a Matlab callback, then edits it some more in C, then passes it into another Matlab callback, etc..
m_ivccsetup.c m_ivccsetup.h	This file contains the commands that m_observer.c uses to talk to the video capture card, and handles the method by which m_observer.c is informed when a new frame arrives.
m_observer.mexlx	This is a complied file, the result of typing "mex m_observer.c m_ivccsetup.c". If you don't edit m_observer.c, you shouldn't need to recompile this file.
grabber.h	This file contains important constants-- it's actually part of the grabber files—make sure you keep the values in this grabber.h consistent with the values in the other grabber.h
ioctl_meteor.h	Really low level constants and such for the capture card. It's highly unlikely that you'll need to work on this file.

After you've untarred these files (type tar -xf <filename>), you're ready to get started. Make sure your grabber is running, and start Matlab in the same directory as the above files.

You should make sure everything is working correctly before you start writing code. Type this into your Matlab window:

```
attachgrabber(0,'showframe');
```

And, surprise, you'll get a huge, ugly error message about segmentation faults and all sorts of nastiness. This is the *last* remaining skeleton in the IVCCLIB closet-- Matlab doesn't like the way I set up my memory, and throws this error ONLY on the first time you run attachgrabber after starting Matlab. You don't need to worry about this error, just run the command again. You'll see some messages like this:

```
>> attachgrabber(0,'showframe')
Observer online, attached to digitizer /dev/bt848-0.
Grabber #0 supplying images to callback showframe.
```

A figure should appear that starts showing you whatever the camera is looking at. Matlab is only capable of displaying these images at about 2 frames per second, but don't worry-- you *can* capture video at 30 frames per second. A few things might go wrong at this point: Press control-c when you want to stop the capture process. Control-c is the way you'll stop your system every time. You'll see this message:

```
Observer break detected.. calling observershutdown().
```

And then you'll get your Matlab prompt back. At least, that's the way it should all work. Here are solutions to some possible problems:

1. The messages appear as above, and a figure pops up, but you don't see any video in it. This means you either don't have a camera properly connected to your capture

card, or you've set your grabber to the wrong source number (see the section on the grabber above.) The good news is that your callback IS receiving frames-- they just aren't the frames you want to be looking at.

2. The messages appear as above, but no figure appears. This means you probably don't have a grabber running on your system. See the section above on the grabber to get your grabber running. If you can see a grabber on your system by typing "ps auxww | grep grabber," try killing all of the grabber processes, starting a new grabber, and trying again. This problem can also happen occasionally if Matlab has a segmentation fault while IVCCLIB is running. In order to correct this problem, you unfortunately have to quit Matlab and start it again before you'll be able to capture video again.

Now that you have the ability to display video in (semi) real-time, you'll likely want to start on your own observer. The way to proceed is to make a copy of showframe.m into another file, say "mycb.m" and start working in that file. Make sure you start by working entirely within the try and catch statements. If you cause an error outside these statements, Matlab will crash in such a way that it doesn't give IVCCLIB to stop the observer process, and you will have to quit Matlab and restart it before you will be able to capture video successfully again.

Your callback can call other Matlab functions, and it can leave data for you to look at after you IVCCLIB exists by creating global variables. To do this, you must declare the variable global BOTH in your callback AND at the Matlab prompt. For instance, while working on face recognition, I put the last face detected in a global variable so that when I stopped the system, I could look at the last detected face.

If you start feeling like your callback is running too slowly, you can move on to the next level of Matlab IVCCLIB programming. You can begin to move time consuming computations into m_observer.c from your callback. The function gotframe in m_observer.c is where each frame of video goes just before it is passed into your callback. You can edit m_observer.c so that it works on the image before it is passed into your callback. In fact, you can pass an image back and forth from C to Matlab, accomplishing time consuming calculations in C, and making use of Matlab functions as needed. Eventually, you could elect to move all of your code from Matlab into m_observer.c, and stop using the mex gateway altogether. This is part of the reason I believe that IVCCLIB in Matlab is the way to go-- you get all of the nice features of Matlab as long as you want them, and can work towards an entirely C program in the end. Continue reading with the next section if you are interested in working towards an IVCCLIB program in C.

Using IVCCLIB in C

Again, I believe that all new users of IVCCLIB should start by using IVCCLIB in Matlab, but if you're a veteran user or don't have Matlab, IVCCLIB in C does offer high performance and an intuitive way to access real-time video.

To get started, make sure your drivers and installed and your grabber is running (read the sections at the beginning of this document.) Then download the IVCCLIB C files from the downloads page. After unzipping and untarring the package, you'll have these files:

writer.c	A very simple C observer. This file writes captured images to disk in sequence. It should serve as a good framework for new observers.
c_ivccsetup.c c_ivccsetup.h	This file contains the commands that writer.c uses to talk to the video capture card, and handles the method by which writer.c is informed when a new frame arrives.

writer	The above 3 files are compiled to produce this executable
Makefile	Make instructions for writer
grabber.h	This file contains important constants—it's actually part of the grabber files--make sure you keep the values in this grabber.h consistent with the values in the other grabber.h
ioctl_meteor.h	Really low level constants and such for the capture card. It's highly unlikely that you'll need to work on this file.

Try running writer (you may need to re-make it). Let it run for a few seconds, then press control-c. You should see a sequence of files that look like #image#, where the first # is the grabber number and the second # is the number of the frame, i.e., 0,1,2..

If you don't see these files, make sure you have a grabber running (type ps auxww | grep grabber). If you can see grabber processes working, you could try killing them all and restarting your grabber.

Once you have writer working, you should make a copy of writer.c and start writing your own observer. The function "gotframe" in writer.c is where you should make all of your changes. Some important notes:

1. You'll notice that in writer.c, the data for the captured frame is immediately copied to a separate location in memory. This is absoultely necessary if what you plan to do with the image will take longer than 1/30th of a second, because the capture card will overwrite that location of memory every 1/30th of a second. If your code executes very quickly, it's possible that you could finish your execution before the memory is overwritten, but copying to a separate location in memory is, in general, a good idea. Future versions of IVCCLIB might implement a ring-buffer of captured frames, allowing an observer as long as a second before the memory will be overwritten.

2. Images are passed to the callback in a somewhat peculiar format. It looks like this: BGRABGRABGRABGRA... That is, blue, green, red, alpha, for each pixel. Writer.c unshuffles this into RGBRGBRGB before writing the images to disk.

3. The NUM_ROWS and NUM_COLS constants in grabber.h are set to 480 and 640, respectively, by default. If you want to capture at a different resolution, you will want to change these values. However, you need to make sure the values in this grabber.h are the same as the values in the grabber.h that is compiled into your grabber program.

Appendix C

Image Morphology Basics

This appendix provides a brief overview of binary image morphology, concentrating on the Erode, Dilate, Open, and Close operations, which are used extensively in the algorithmic face detector in chapter 2. Image morphology is a basic image processing tool that is useful for a number of reasons. First, it can be used to clean out "noise" in images, as I used it in my face detector. Second, it can be used to "grow" or "shrink" the white regions in a binary image. Finally, image morphology is useful for automatically combining many small white regions into a few larger white regions.

Erosion and Dilation

The two simplest binary image morphological operations are erode and dilate. Dilation, generally speaking, causes the white regions in a binary image to "grow." It accomplishes this by moving a small image, called an *operator*, over every pixel in the

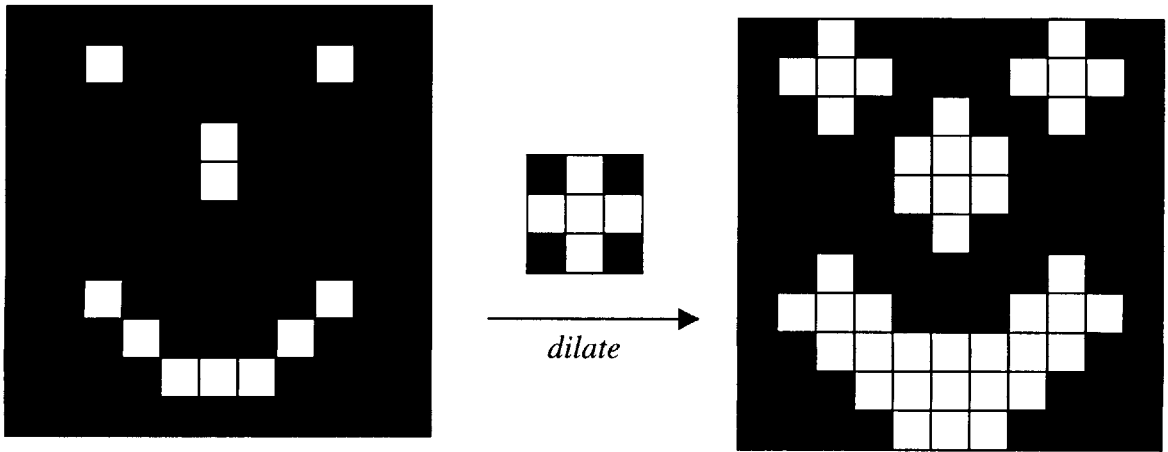


Figure C-1: Dilation with a cross-shaped operator

image. If the pixel below the operator is white, every pixel in the input image that is under a white pixel in the operator is turned white. An intuitive way to think of this process is to imagine moving a white rubber stamp over the black and white input image, pressing it into the image whenever the pixel that is directly below the center of the stamp is white. Figure C-1 shows the results of the dilation operation on a simple image using a cross-shaped operator. Note that an operator can be any shape—in fact, operators of different shapes will cause very different dilation results.

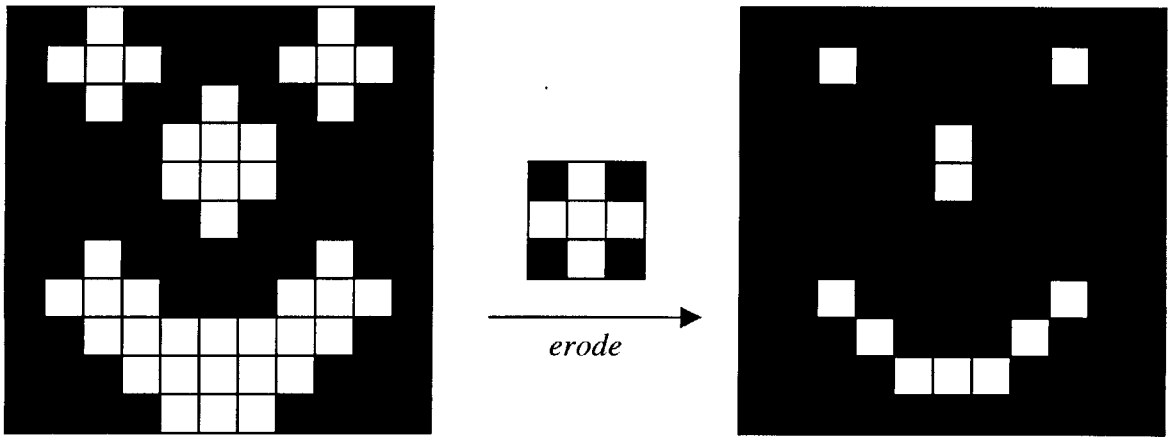


Figure C-2: Erosion with a cross-shaped operator

Erosion is the opposite of dilation. Instead of growing the white regions in an image, erosion shrinks, or even eliminates these regions. In terms of the rubber stamp analogy above, the same operator is moved over the image, this time with black "ink." When the pixel below the operator is black, all pixels under white pixels in the operator are painted black. Figure C-2 shows the results of erosion with a cross-shaped operator. Note that, in this case, erosion precisely undoes the effects of dilation. This isn't always the case, however, as can been seen in C-3. Here, a dilation operation followed by an erosion operation using the same cross shaped operator has eliminated the hole in the center of the circle.

When dilation is followed by erosion, it is called a "close" operation. In general, the close operation fills in holes and combines small white regions together without greatly affecting the contours of the white regions. Erosion followed by dilation is the "open" operation, and generally separates large white regions into smaller regions, again without changing the contours of contiguous regions noticeably.

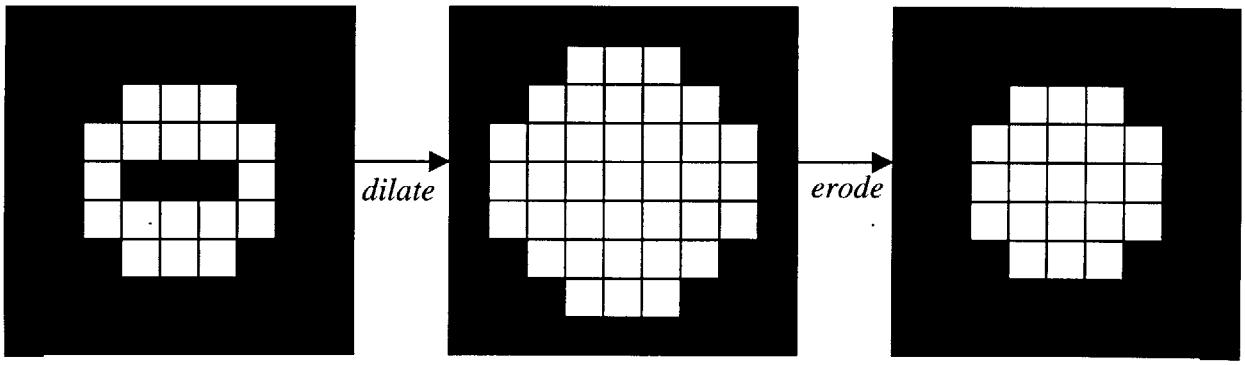


Figure C-3: Dilation followed by erosion forms a "close" operation

Bibliography

1. Drury, H.A., *Meta-Analyses Of Functional Organization Using The Visible Man Surface-Based Cerebral Atlas.* BrainMap: Spatial Normalization And Registration, 1996.
2. Carlson, N.R., *Physiology of Behavior, 5th edition.* 1994, Needham Heights, MA: Paramount Publishing. 88-91.
3. Moylan, M.J., *Biometrics is Big*, in *The Dallas Morning News.* 1997, Pioneer Press: Dallas.
4. Visionics Corporation, *Visionics FaceIt is First Face Recognition Software to be used in a CCTV Control Room Application*, . 1998, Visionics Corporation: Jersey City, NJ.
5. Rogers, W., *Viisage Technology, Inc. to Spin Off New Facial Recognition Company*, in *The Biometric Digest.* 1998.
6. Sung, K.-K. and T. Poggio, *Example-based Learning for View-based Human Face Detection*, . 1994, MIT AI Laboratory: Cambridge, MA.
7. Rowley, H.A., S. Baluja, and T. Kanade, *Neural Network-Based Face Detection.* IEEE Transactions on Pattern Analysis and Machine Intelligence, 1998.
8. Bishop, C.M., *Neural Networks for Pattern Recognition.* 1995, New York: Oxford University Press, Inc.
9. Turk, M. and A. Pentland, *Eigenfaces for Recognition.* Journal of Cognitive Neuroscience, 1991. **3**(1): p. 71-86.
10. Turk, M.A. and A.P. Pentland, *Face Recognition Using Eigenfaces.* 1991.
11. Pentland, A., B. Moghaddam, and T. Starner. *View-Based and Modular Eigenspaces for Face Recognition.* in *IEEE Conf. on Computer Vision & Pattern Recognition.* 1994. Seattle, WA.
12. Belhumeur, P.N., J.P. Hespanha, and D.J. Kriegman, *Eigenfaces vs. Fisherfaces: Recognition Using Class Specific Linear Projection.* IEEE Transactions on Pattern Analysis and Machine Intelligence, 1997. **19**(7): p. 711-720.
13. Duda, R.O. and P.E. Hart, *Pattern Classification and Scene Analysis.* 1973, New York: John Wiley & Sons, Inc.
14. Olivetti Research Laboratory, *Olivetti Research Laboratory Face Database*, .
15. Learning and Vision, *Variable Viewpoint Reality*, . 1999, MIT Artificial Intelligence Laboratory.
16. Stauffer, C., *A Forest of Sensors*, . 1999, MIT Artificial Intelligence Laboratory.