2007

University of North Carolina Wilmington

Master of Science in

Computer Science and Information Systems

Proceedings

https://csbapp.uncw.edu/mscsis

NEUROCOGNITIVE INSPIRED HIERARCHICAL FACE RECOGNITION SYSTEM

RYAN WILKINS

A Capstone Project submitted to the
University North Carolina Wilmington
Master's of Science in Computer Science and Information Systems Degree

Department of Computer

University of North Carolina Wilmington

2007

Approved by

Advisory Committee

_____ _____

_____
Chair

TABLE OF CONTENTS

ABSTRACT

The race-specific face recognition system introduced in this paper increases identification rates relative to the Eigenface baseline established by Mathew Turk. The race-specific system performs processes similar to those that humans do when identifying a face. The neurocognitive phenomenon identified as own-race bias is the basis of this system which first classifies the face by race and then identifies it against a stored database.

Evaluation phase of the race-specific system compares the baseline recognition system against the race-specific by examining the methods which improve classification rates and identification rates. The comparison will evaluate the total system using a standard rank N identification rates. The MORPH database is used as the face corpus where particular images were sampled based on criteria. The MORPH database was developed at UNCW for analysis of the effects of age-progression.

ACKNOWLEDGEMENTS

My thanks go to UNCW and the Department of Computer Science whose courses and clubs have provided me with a good foundation to a successful future. I am especially grateful to the professors that have taught classes like Artificial Intelligence, Pattern Recognition, and Scientific Computing whose content intrigued my thoughts and induced my enthusiasm to learn.

The friends that I have attained through the Computer Science program and many evenings we worked late nights in the lab, thank you. Finally, I would like to thank Dr. Ricanek, without his undivided time, guided strength, and social aptitude I would not have been able to finish this work.

Special thanks to my parents and my sister who helped me both financially and encouragingly. Also, special thanks to my grandmother Eleanor Forbes for her guidance and perseverance.

I would like to thank my committee for there time and effort to keep me on track while developing this thesis. Finally, I would like to thank my committee for their guidance, equipment, financial support, and assistance throughout my studies.

DEDICATION

I would like to dedicate this thesis to my family whose continued support, encouragement and dedication to be loving parents have meant more to me than they will ever know.

LIST OF TABLES

LIST OF FIGURES

1.0 INTRODUCTION

1.1 Background

Over the past decade, the global community has seen an increase in face recognition research and its applications. The actors involved in FR research consist not only as computer scientists, engineers, and mathematicians, but also neuroscientists and psychologists. Because humans have an innate ability to perform rapid, robust face recognition, neuroscientists have focused on determining the cognitive processes that are involved in face recognition in order to better understand the mysteries of the brain, as well as to develop better, neurocognitive inspired models of the human face recognition engine. Computer scientists, engineers, and mathematicians incorporate the findings of neuro-science in computer algorithms that can be used for automated recognition. This process of modeling human neuro-processes in order to exploit natural human capabilities through computers has occurred on many levels, e.g. artificial neural networks, expert systems, etc.

Biometric research is the study of methods for uniquely recognizing humans based upon one or more physical characteristics or behavioral traits. Physical biometrics includes:

- face recognition
- iris and retinal scans
- finger print and finger geometry
- palm print and geometry
- gait (physical pattern associated with walking/jogging/running)

While behavioral-biometrics are traits such as:

- handwriting
- speech
- typing patterns

The biometric chosen for this project is face recognition due to its public acceptance and its legal grounds. Face recognition plays a critical role in biometrics systems and could become the biometric of choice when visual security is maintained. Face images can be found in many legacy databases which may be the only available data source for security systems. The design of FR algorithms that effectively handle environmental influences like lighting, low-resolution, occlusion and complex backgrounds are in high demand. Face recognition as the accepted biometric is growing due to the complex security systems currently needed to provide safety in crowded, densely populated areas. People in general feel comfortable displaying their face in public, which means FR systems do not inhibit the privacy of others.

1.2 Statement of the Problem

A general problem statement of face recognition can be formulated as follows: Given an image of a single face or scene of faces from photo or video identify or verify, depending on the context of the problem, one or more persons relative to the scene using a stored database of known faces

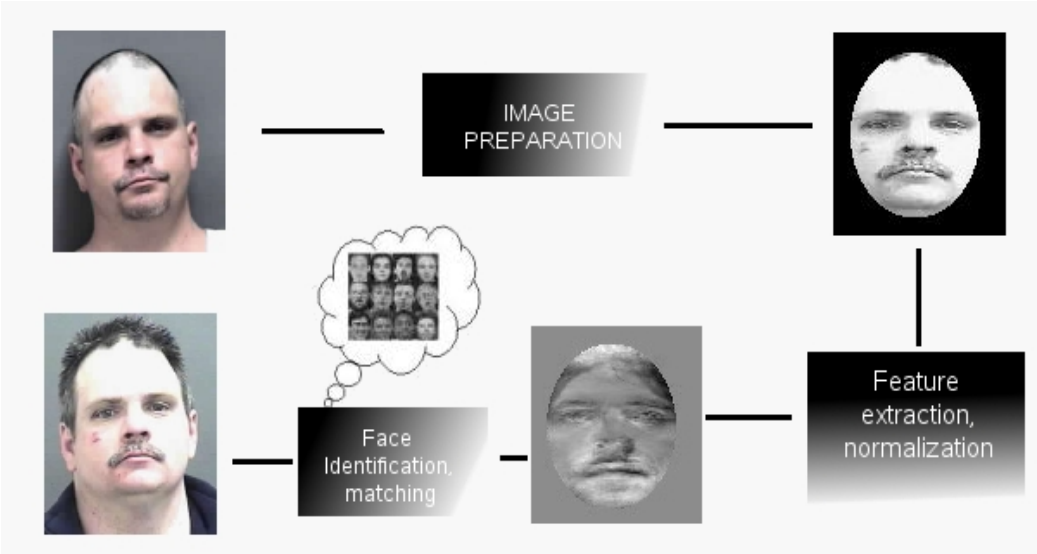

Figure1. General processes involved in face recognition systems.

1.3 Overview

In addition to public acceptance of face biometrics, there are well established legal grounds for ubiquitous use of both public and clandestine video recording equipment. Some examples of the ubiquitous use of video monitoring and recording in public use are grocery store, banks, department stores, fast food locations and other retailers; while clandestine video occurs at ATM machines, police squad cars, and the likes. The United States Supreme Court has determined that government action constitutes a search where it invades a person's reasonable expectation of privacy. But the Court has found that a person does not have a reasonable expectation of privacy in those physical characteristics that are constantly exposed to the public, such as one's facial characteristics, voice or handwriting (*United States v. Dionisio*, 410 U.S. 1, 14 (1973).)

There are many reasons why the research for this project and other human identification techniques are viable today. One major reason is government sponsored

research funds for human identification, which FR is on of the most promising techniques has increased due to the instability of world peace. Currently, airlines, border patrols and other entry points to countries have increased surveillance and security measures. The actions of those who intend to do harm and terrorize have instilled a fear among people to such a point that some personal privacy has been relinquished by the general public. Since September 11, 2001, a focus on security at all levels of government has grown to unprecedented heights influencing the need for human identification research, FR research in particular. At government facilities, such as Federal Bureau of Investigation (F.B.I) and Central Intelligence Agency (C.I.A.), various FR systems are used to identify and verify wanted persons that may either be in custody at a local level or found in a photo or video. The United States armed forces have also began using FR technology overseas to identify high value targets. The threat of terror has changed the world's view on security's processes and procedures in which people face everyday. Social experiences like the extreme example mentioned above influence the way people perceive others. Many psychologists and sociologists hypothesize that it is these social experiences, positive or negative, that build a person's perception of a face. One's personal perception of another does influence a person's actions and behaviors, however the question is asked whether or not these perceptions influence the ability to identify a face.

Psychologists believe that facial features – nose, mouth, eyes, face shape and their spatial relationships – work as the code or key in which a person uses to recognize a face (Ivanov, Heisele, Serre, 2004). Many times one can subset these face features into certain groups, for example a long nose versus a wide nose, or a sharp pointed nose. The

minute details or differences of these features work as the decision rule when a person recognizes a face. The combination of the different facial feature subsets and personal experiences is the means to human face recognition. However, the process that humans take to identify a face is much more complex than only those two processes. Since facial features have subsets, then also the face that displays the features may be sub-grouped. Facial feature subsets are a product of evolution and environment.

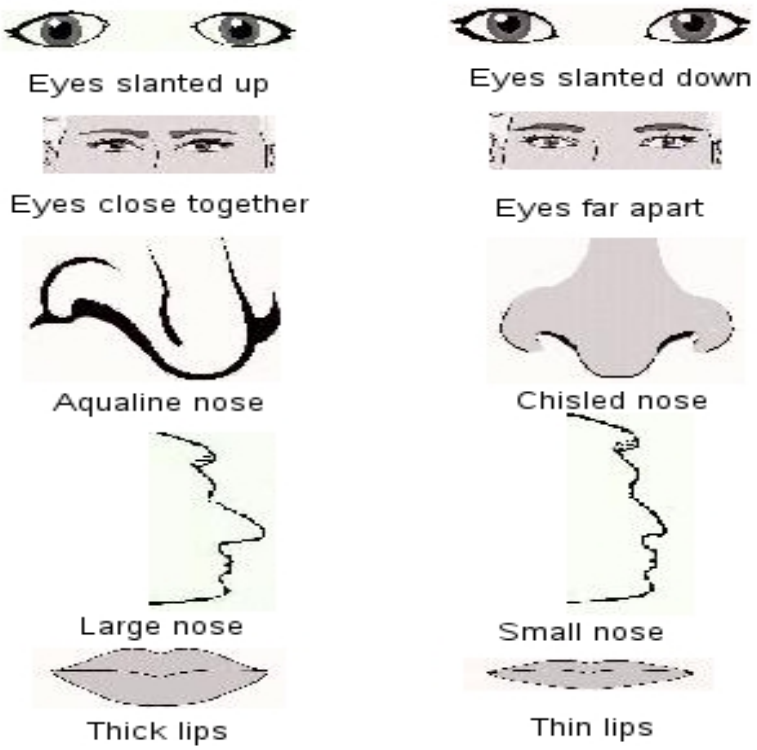

Figure 2. Examples of facial feature subsets which can categorize people with similar features

For about 2.4 million years, people have endured environmental and sociological changes. The changes in the climate have evolved the way people live and look. Before there were acceptable methods to travel far distances, human groups lived separate from each other for a long period of time. People from a certain geographical region had

physical attributes that aided them specifically for that region. Humans' eyes, nose, cheeks and mouth also evolved. These physical changes were passed down the gene line as people reproduced. The physical changes segregated the world population and people began grouping themselves into classes. The classes formed based on the geographical regions where people lived, their beliefs, their physical differences, and their language.

As the world matured and technological innovations made the world a smaller planet – by reducing the physical and monetary cost of travel. People were seeing humans with physical characteristics that were different from their own, for the first time. These differences in physicality and language would drive humans to instinctively segregate people into groups based upon a person's physical characteristics. Even though the races combined, today, in the 21st century, humans still carry these labels with them. Currently, the politically correct way to describe another's class would be to use the word ethnicity. Ethnicity became the basis for social categories that are rooted in socially perceived differences in national origin, language, and/or religion. Notice that this definition does not mention any physicality traits which describe someone's ethnicity. However, there are general physical characteristics which are inherent to a person's ethnicity.

The physical characteristics are generally applicable to a person's ethnicity; however, there are people within a certain ethnic origin that do not carry the common traits. Physical characteristics such as facial features and skin pigmentation are the common exemplars in which people use to classify a face as belonging to a particular ethnic/racial group. Over the past three decades, psychologists have researched human face recognition and how performance is influenced when recognizing a face dissimilar

of their own. The research has found that a person's face recognition performance is influenced by social experiences and unfamiliarity of dissimilar facial features. Generally, a person's face recognition performance increases when recognizing a face similar to their own.

In this paper, a technique to increase the efficacy of a FR system is introduced based on a model of the human neurocognitive system. This work models a psycho-cognitive phenomenon identified as own-race bias1. The own-race bias reflects differences in racial experience such that people have more exposure and practice recognizing faces from their own race group relative to faces of other races (Tanaka, Keiefer, & Bukack, 2004). Section 2 of this paper gives detail and insight into literature that supports this work and gives detail on human face recognition. In Section 3, methods such as the PCA algorithm and the Eigenface approach are examined. Section 4 details a description of the software and evaluates results from classification and identification tests. A conclusion and discussion of future work is found in Section 5. The work in (Tanaka et. al., 2004) has established that people acquire the ability to differentiate faces based on experience of face-to-face contact. The features and configurations, spatial relationships of features, are the main factors used in the human recognition process (Tanaka et. al., 2004). If a human has more experience within a

1 Since the term, *race*, is used frequently and continuously throughout this paper and project, a definition is essential. Although there are many definitions of this controversial term, the work and research presented here defines race by the fact that "...the most widely used human racial categories are based on visible traits (especially skin color, facial features and hair texture), ..."[24]. In face recognition, images are the only quantifiable measure thus any background information like ethnicity goes unused. This, alone, is why the definition of race presented here is limited only to the exterior face features.

class of faces (racial class), then that person's ability to extract and analyze class-specific features of a face will increase.

2.0 REVIEW AND ANALYSIS

2.1 Own-Race Bias

A synonym in context for own-race bias is the *cross-race effect* which demonstrates that other-race faces are more difficult to accurately recognize than same-race faces (Herrera, McQiston, MacLin, & Malpass, 2000). The hypothesis this research uses to explain the phenomenon of own-race bias and cross-race effect is known as *differential experience* (Herrera et. al., 2000). In general, differential experience, which begins at birth, consists of the learning/discrimination of visual stimuli as it is repeatedly encountered. These stimuli continue to gain effect as a child grows older and continues to experience social contact through school, sports, religion, media, etc. Cross, Cross, and Daly (1971) reported that Caucasians living in integrated communities exhibited greater recognition ability for African faces than did those living in segregated communities. Feinman and Entwistle (1976) examined adults who claimed to have close friends of another race against adults without, and found the ones with close, other-race; friends exhibited less of a cross-race bias than those without (Herrera et. al, 2000).

Platz and Hosch (1988) studied African-American and Caucasian-American convenience store clerks in El Paso, TX. The study examined the store clerks' ability to recognize Hispanics. Although Hispanics are heavily populated within this town, the results of the experiment differed from claims of Cross, Cross, and Daly. The clerks had a low degree of recognition for the Hispanic patrons although the face-to-face contact was frequent. This counterpoint is weighted lightly due to the fact that the clerks'

formative racial experience was not obtained – whether the clerks grew up in a racially segregated or integrated community will influence cross-race recognition.

General research shows that adults who grew up in integrated communities had a better ability to recognize other-race faces. Additionally, there is evidence from the research exposure to other races after the formative years, 18 years old and older, yield the same benefits in identification of other-race faces. Hence, the formative years of life play the most critical role in a person's ability to identify other races as an adult.

2.2 Schema Hypothesis

Schema hypothesis states that an age-related memory schema develops over time for own-race faces which increases FR accuracy (Herrera et. al., 2000). At young ages human brains are experiencing the most growth; neural connections are formed, tuned and re-tuned and it is this stage of life, when recognition ability is most influenced. Hence, the more visual contact children have with other-races forces the brain to collect the information, face features and configurations, needed to facilitate recognition at later ages. Chance, Turner and Goldstein (1982) strengthened schema hypothesis in an experiment where Caucasian-American participants ranging 6 to 20 years old were used. Grades one and two recognized Caucasian-American and Asian faces equally well, but for older participants Caucasian FR rates were much higher than those of Asian (Pezdek, Blandon-Gitlin, Moore, 2003). Feinman and Entwisle (1976), along with their previously mentioned study, tested Caucasian-American and African-American children in grades one, two, three, and six. The test was on the ability to recognize both own and cross-race faces (African-American and Caucasian-American). Although FR rates grew with age, own-race effect was still evident: Caucasian-American children were more accurate in

recognizing Caucasian-American face, as with African-American children would more accurately recognize African-American faces.

Recent research by Daniel T. Levin, Ph.D at Kent State University has supported cross-race effect suggesting that the information people "see" when looking at a face of another race is information that allows them to classify the person as Caucasian or African American, but not information that allows them to identify the face. The research presented in (Levin et. al., 2000) is based on experiments designed to determine the type of information people retain when looking at cross-race faces. In his first experiment Dr. Levin examined how frequent participants locate face from other races in a visual search task. The participants consisted of 25 people who were mostly Caucasian and a few Asian. The participants first viewed a set of 32 faces made up 16 African-American and 16 Caucasian which were dispersed at random. In the next step the participants were shown a set of photos from a yearbook and were asked to identify the ones located in the previous set. As expected, the face memory test using yearbook photos, participants were better at identifying Caucasian faces than African-American ones. The experiment supports his hypothesis that people look for discriminating features first when recognizing a face. In other words, the face information people focus on when looking at a face of another racial group is information that is optimal for race classification rather than identification.

This extensive literature review supports the belief that people recognize faces from their own race more holistically than other-race faces. The application of this neurocognitive model, applied to FR algorithms, increases recognition rates by amplifying sensitivity to facial features that are racially homogeneous. In this FR system,

a face is first classified by race, and then recognized by a racially tuned FR algorithm.

3.0 METHODS

3.1 MORPH Data Corpus

The MORPH data corpus is a longitudinal face database developed for researchers investigating all facets of adult age-progression. Only a few of the many face and gesture data corpora that are publicly available for researchers focus on duplicate face images at various adult ages. The MORPH database records meta-data on the face images within the database: subject's ethnicity, height, weight, gender and age are critical to this research project. The images and meta-data are currently, obtained from public records. The entire MORPH database is divided into two sets named Album 1 and Album 2. Figure 3 is a sample of three subjects from the database. The sample demonstrates the general appearance changes that are typical with age-progression: filling out of the face due to weight increase, formation of wrinkles and lines, and sagging due to loss of skin elasticity and muscle tone.

Figure 3. Photographs of two age progressed individuals from Album 1 of the MORPH data corpus

Album 1 contains scans of legacy photographs which were taken between October 26, 1962 and April 7, 1998. The photographs were scanned using a flatbed consumer

grade scanner in full 48-bit color and a capture resolution of 300 dpi. The digital scans were cropped to an image size of 400 x 500, so that all faces are uniformly sized. Image quality was enhanced by methods such as median filtering and contrast stretching via histogram equalization. As of this writing, Album 1 contains over 1,600 images of 632 individuals. There are 1,253 images of faces with African-American descent and 433 images of faces with Caucasian descent. Album 1 is made up of mainly men's faces, only having 285 images of women with 1405 images of men. Figure 4 is an example ranking for each three categories of image quality which are shown.

Figure 4. Three photographs from Album 1 that have poor, fair and good image quality, respectively

Album 2 contains images photographed by a standard consumer grade digital camera between February 23, 2003 and March 15, 2006. The images were stored and compressed using Joint Picture (JPEG) compression. The face images were sized to a uniform 200 x 240 pixels. The images are not enhanced by any filtering or contrast stretching techniques. Album 2 contains 15,204 images of 4,039 individuals where 3440 are males and 599 are females. The population of males has 12,984 images and the female population has 2,220 images. The ethnicity (Table 1a.) and age (Table 1b.) breakdown for Album 2 are presented below. Figure 5 is a sample of Album 2 that shows age progressed faces for both Caucasian and African American sets.

Figure 5. A sample of age progressed faces taken from Album 2.

Table 1a. Counts on the age categories of individuals in Album 2

Statistics by Decade of Age					
	<18	18-29	30-39	40-49	50+
All	0	33	6335	6696	2140
Male	0	29	5371	5679	1905
Female	0	4	964	1017	235

Table 1b. Counts on the race categories of individuals in Album 2

Statistics by Ethnicity						
	African-American	Caucasian-American	Asian	Hispanic	Other	Total
All	11948	3200	7	34	15	15204
Male	10283	2650	5	34	12	12984
Female	1665	550	2	0	3	2220

3.2 Image Samples Used

The populations of images used for this project were sampled from Album 2 due to higher image quality and the number of possible samples. The images used were chosen with the following criteria:

- male population only
- age range between 20-35

- lacking facial hair or little facial hair relative to other images in the database
- neutral facial expression, faces were not smiling or scowling
- as vertical as possible pose with similar view frustum
- Caucasian or African American descent

The criteria listed above were applied to all images used in the project whether as training or testing data. The faces were chosen based on a limitation of scope that was discussed early in the project. A sample of only male images was used because the count of male faces available to sample were much greater than female faces. If a combination of female and male images were used, then the results would have been skewed because female facial features generally have major differences than male. In addition, the goal of this project was to evaluate the use of racial classifiers to increase the efficacy in a FR system. If a dual gender sample was used, then gender classifiers would arise. Further work should be done using this FR system with female faces as input training and testing data. Based on the results of this project, similar to better performance should be seen using female faces as input because facial hair would not be a factor. The faces sampled ranged in age, 20-35, due to little facial change found in this period of life.

However, some pitfalls of these criteria were found; the population of Caucasian faces that followed the above criteria was much smaller than that of the African American population. The upper limit bounded the Caucasian sample size to 200 images of 100 people where each person had two faces photographed at different times. The African American population were sampled the same thus having 200 images of 100 different people. Each image had meta-data which was used to validate against the criteria listed above. Table 2 shows the meta-data available for the MORPH database

notice that age is derived from date of birth and date of acquisition.

Table 2. The meta-data available for an individual in the MORPH database

Field	Field Definition
Idnum	The identification number for the individual
picture num	The number for the picture for the individual
DOB	date of birth
DOA	date of acquisition (date photo taken)
Weight	weight of the individual
Height	height of the individual
Race	Caucasian, Hispanic, Asian, or African Amercan
Gender	male or female
Age	derived from DOB and DOA

3.3 PCA Algorithm

Principal Component Analysis (PCA) is an information-preserving statistical method used for dimensionality reduction and feature extraction. PCA is primarily used to determine an orthogonal space with minimal dimension for representing a data set. In other words, it is a linear transformation from a high dimensional (N-dimensions) space to feature space where the feature space has N'-dimensions and N' << N. PCA can show linear interdependence of a data set where sample variances are found along orthogonal axes.

Principal Components Analysis is as follows:

Suppose $x_1, x_2, \ldots x_m$ are N x 1 vectors

Step 1: $\bar{x} = \frac{1}{M}\sum x_i$ where i = 0,1,2, ... M (1)

Step 2: subtract the mean: $\Phi_i = x_i - \bar{x}$ (2)

Step 3: form the matrix $A = [\Phi_1 \Phi_2 ... \Phi_M]$ (N x M matrix), then compute:

$$C = \frac{1}{M} \sum_{i=1}^{M} \Phi_n \Phi_n^T = AA^T \quad (3)$$

Step 4: compute the eigen values of $C : \lambda_1 > \lambda_2 > ... > \lambda_N$

Step 5: compute the eigen vectors of $C: u_1, u_2, \ldots, u_N$

The covariance matrix C is symmetric thus $u_1, u_2, \ldots, u_N$, form basis vectors for the subspace which means any vector x, can be written as a linear combination of the eigenvectors:

$$x - \bar{x} = b_1 u_1 + b_2 u_2 + ... + b_N u_N = \sum_{i=1}^{N} b_i u_i \quad (4)$$

Step 6: Reduce the dimensionality by keeping only the terms that correspond to the highest energy within the set of eigenvectors. In other words, keep only the eigenvectors corresponding to N' largest eigenvalues where $N' << N$.

3.4 Eigenface Approach

The Eigenface approach is a PCA based face recognition algorithm, where eigenfaces are the eigenvectors and represent the variation of a face training set. Matthew Turk and Alex Pentland developed the eigenface approach which is considered one of the first successful face recognition technology. The algorithm begins with a large set of images digitized under similar lighting conditions with eye and mouth positions in line. The image pixels are then converted to gray scale and a covariance matrix is created from the training set. Eigenfaces are extracted from a covariance matrix made up of a difference distribution between a set of high-dimensional vectors (face images) and the mean vector (average face image). A large set of eigenfaces are calculated from the covariance

matrix.

Eigenfaces represent similar face characteristics in a face set. Those characteristics or facial features are eyes, nose, mouth, jaw line, skin pigmentation plus there relative location to each other. The principal components are chosen from this set, only selecting the eigenvectors (eigenfaces) with the largest eigenvalues. The eigenvalues represent the magnitude of variation that its respective eigenvector describes in a data set. Reducing the dimension of a face dataset reduces the number of characteristics needed to describe the face set. Although image data, for the most part, is noisy with high variation, the Eigenface approach utilizes the fact that a face is a pattern or a set of patterns. Generally, all faces have similarly located features like eyes, nose, and mouth. The Eigenface approach measures the similarity between the patterns of facial features. The following figures assist to visualize the Eigenface approach.

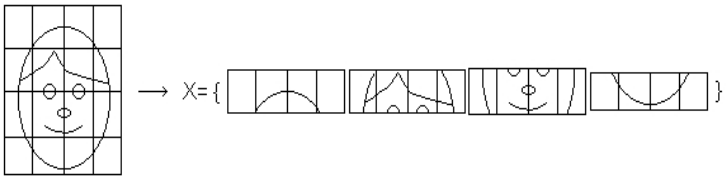

Figure 6a. Construct an image vector by concatenating rows of pixels

$$\Sigma_X = \mathbf{X}^*\mathbf{X}^T = \begin{bmatrix} \sigma_{11}^X & \sigma_{12}^X & \cdots & \sigma_{1,w^*h}^X \\ \sigma_{21}^X & \sigma_{22}^X & \cdots & \sigma_{22}^X \\ \cdots & \cdots & \cdots & \cdots \\ \sigma_{w^*h,1}^X & \sigma_{w^*h,2}^X & \cdots & \sigma_{w^*h,w^*h}^X \end{bmatrix}$$

Figure 6b. The covariance for a set of faces where φ_{ij} represent the covariance between pixel i and the pixel j.

$$\Sigma_Y = \mathbf{Y}*\mathbf{Y}^T = \begin{bmatrix} \sigma_{11}^Y & 0 & \dots & 0 \\ 0 & \sigma_{22}^Y & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & \sigma_{w*h,w*h}^Y \end{bmatrix}$$

Figure 6c. Construction of the face space where **Y** is the matrix containing vectors y_i or the principal components which describe the face x_i in face space

The facial characteristics present in an eigenface are seen when the eigenface is displayed to a screen (Figure 7). The characteristics are noticeable by light and dark shapely formations on the painted face. Many patches are recognizable facial features such as nose, mouth, and hairline, and some are less simple to identify. Basically, eigenfaces represent a standard set of faces where any face could be considered a combination of the standard set. The Eigenface approach has advantages over other face recognition algorithms; it returns a solution relatively fast and it does not require a large number of images summed together to measure similarity between faces.

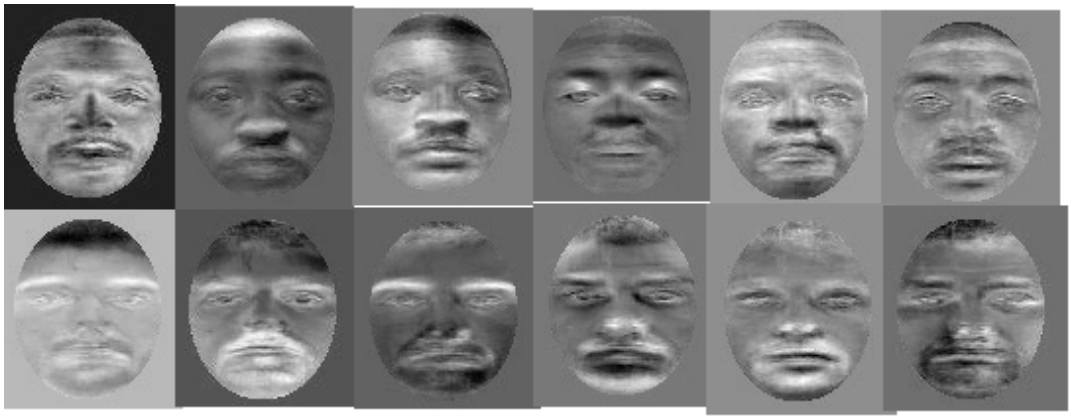

Figure 7. The first row is African American eigenfaces and the second row is Caucasian. The eigenfaces are ranked left to right where highest eigenvalue corresponds the left most eigenface.

A feature of the eigenface approach is that an original image can be reconstructed from the training set by combining eigenfaces. If all of the eigenfaces were used from

the training set, an image can be reconstructed exactly. In most cases, only a smaller subset of the total eigenfaces is used, thus image reconstructions become approximations. However, since eigenfaces basically represent face characteristics, an argument can be made to say that any face can be reconstructed by combining face features at the right proportion. Eigenfaces represent face characteristics found in a face set which means a single, original face may or may not have the same eigenface feature present. However, if the eigenface feature is more apparent in a face, then that eigenface should have a higher share of the "sum" of eigenfaces. Therefore, an original image can be reconstructed by a weighted sum of eigenfaces. The weight or coefficient is the degree in which an eigenface is present in the original image (Equation 4).

The exact method to relate the above concepts, definitions, and features with face recognition may not be apparent. The main clue is the fact that faces can be reconstructed from eigenfaces and a set of weights. However, the opposite is also true; face weights can be extracted from eigenfaces and an unknown face. The coefficients of a face describe the difference between a face and respective eigenfaces. A set of eigenfaces creates a transform from image space to face space where the eigenfaces represent basis vectors. Projecting an unknown face into face space via the eigenface transform extracts face coefficients which are used to measure the similarity between faces. Two general problems are solvable using the weights of an unknown image:

Determine whether or not the unknown image is of a face. This case is determined by finding the difference between the unknown image weights and the known face's weights. If the difference is found to be overall greater for the unknown image, then it is probably not an image.

Faces that are similar have similar weights when projected into the same face space, thus have similar face features. If the face space was graphed, then clusters of weights would be seen. A single set of weights represent the location of an image in face space. The closer a set of weights are to another corresponds to the similarity of two images.

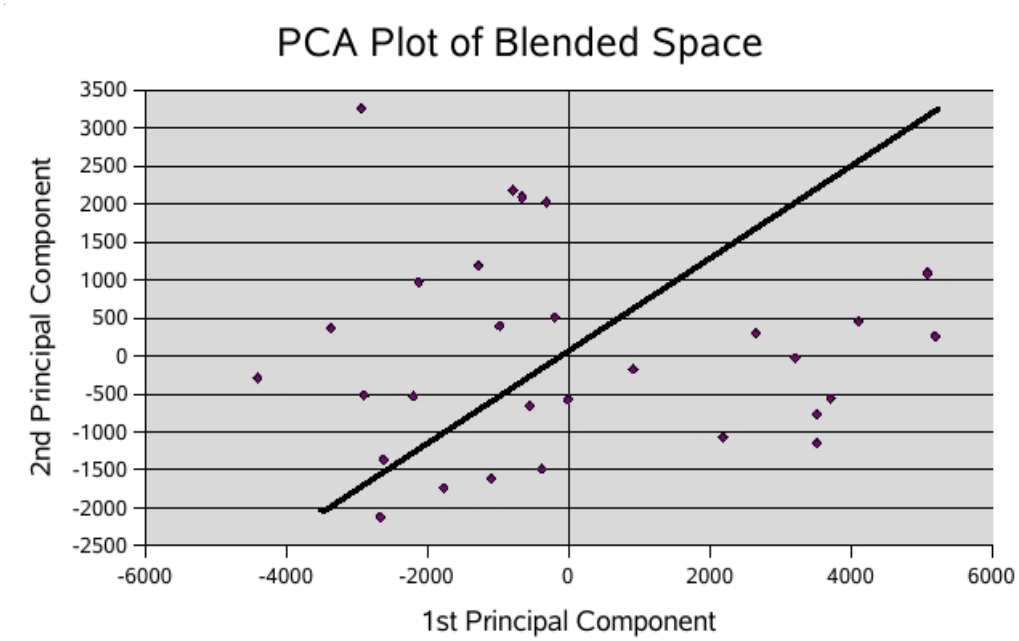

Figure 8. Plot of the 1st set of Eigen coefficients against the 2nd set of Eigen coefficients and a regression line that divides the space.

4.0 PROJECT DESCRIPTION AND RESULTS

The following sections describe the processes taken to prepare face images, build respective face spaces, and create a decision rule for both classification and identification. The project is written in C++ using the Open Source Computer Vision Library (OpenCV). OpenCV is intended to aid commercial uses of computer vision in in human-computer interface, robotics, monitoring, biometrics and security by providing a free and open infrastructure where the distributed efforts of the vision community can be consolidated and performance optimized (Open Source Computer Vision Library). The

project and unit testing are executed on a standard commercial grade laptop computer running a Linux based operating system.

Early in the project a decision was made whether to write the components of the PCA algorithm or not and use existing libraries. Since PCA is a well known algorithm in the computer vision community and programming implementations are easily found on the web, the decision to use the OpenCV library was based on two reasons. First, OpenCV can handle a high number of input images which is bounded by the I/O buffer size. Second, OpenCV has functions that are optimized for computer vision processes, mainly face recognition.

The project contains five classes and one driver class that calls the functions needed to read in the input data, build the face spaces, run classification and identification tests, and display results. Each class is specialized for a particular set of processes within the face recognition system. The six classes are as follows:

FCRS.cpp – the driver class (Face Classification Recognition System)

FaceReader.cpp – this class reads in input data from specified files and builds training/testing data

FaceDatabase.cpp – this class grabs the input data from *FaceReader,* allocates the images and acts as central storage to hold all images

FaceSpace.cpp – this class is instantiated three times to build the three decoupled face spaces (African American, Caucasian and Blended)

FaceClassifier.cpp – this class contains the decision rules for race classification of testing images

FaceIdentifier.cpp – this class makes appropriate function calls to the respective

face spaces and receives return values to evaluate identification metrics

A combined count of the six classes totals to about 4,500 lines of code with an average of 40 functions per class which does not include any external applications written for image preparation or statistics.

4.1 Image Preparation

Since all of the images used in this project are sampled from the MORPH database, then image preparation is imperative due to the uncontrolled image factors like irregular lighting and articulated pose. An external application outside of the main FR system was written specifically to handle the image preparation process. The application was written in Java™ for sake of time and familiarity of libraries. However, Java the application named *Image Preprocessor* is not trivial. Image processing algorithms were written for the following image preparation processes, but did not take advantage of any existing Java Advance Imaging, since the developer was knowledgeable of image processing algorithms.

1. the images are rotated so that all faces are aligned horizontally by the eyes (see Figure 9b.)
2. the image pixels are converted to gray scale
3. the faces are sized to 100 x 110 pixels ensuring all eye points are located at the same x, y coordinates
4. an elliptical mask is used to remove extraneous sources of variation e.g. hair, background etc.

The PCA algorithm requires image preparation because it is sensitive to variation of pixel intensity that may be caused by background differences, hair styles, image size,

pose articulation etc. Many images in the MORPH database have angle pose articulation where faces had to be rotated to create a full frontal pose. In general, image preparation attempts to conform uncontrolled images and remove the image variation added by face pose, camera location and lighting. Rotating the face, scaling it down, and aligning eye points removed the majority of variation extrinsic to a face. An elliptical mask removes outer features such as ears, hair, and background which influence face variation. Even though most images were chosen based on a relative lack of facial hair and view frustum, the ellipse had a static size and did not remove all hair in all cases, especially where facial hair hung down the forehead, i.e. bangs. The Eigenface approach is unable to differentiate between hair (not a face feature) and a face feature, so hair gets misinterpreted as a face characteristic. More importantly, hair becomes a principal component and describes the similarity of a data set. The faces in the MORPH database had frequent hair features that were not avoidable without using a face hair removal algorithm and a dynamic mask. Most of the men in the database had hair features like goatees, mustaches, and bangs. No attempt was made to remove facial hair, so these features were seen when eigenfaces were painted to the screen.

Figure 9a. Example of extreme angled pose

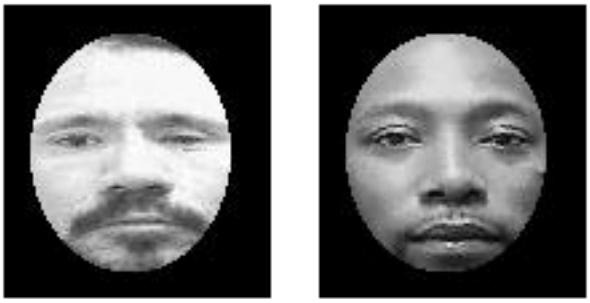

Figure 9b. Preprocessed faces of Figure 9a

4.2 Training Space

The race specific training sets of 30 images are chosen at random for both Caucasian and African-American face sets. The 30 blended training set faces are sampled using 15 Caucasian training images and 15 African-American training set. An average training face is derived from the set of input training faces. The average training face is passed on to create the covariance matrix and eigenfaces. Sixteen eigenfaces with highest eigenvalues are chosen to describe the variation of the training set in face space. The coefficients of the sixteen eigenfaces are stored and later used for race classification. The sixteen eigenfaces were chosen because prior work found in (Yambor, Draper, & Beveridge, 2004) which states that the lower ≈40% of the eigenfaces can be removed without adversely effecting identification rates.

4.3 Known and Test Space

The known set of thirty face images are chosen at random, excluding training faces, for Caucasian, African-American and blended face spaces. Each person has two face images in the population set where each photograph was taken on different dates. The younger face image is used as the known face and the older face image used as the test face, a.k.a. probe, because this would follow real world application where generally a

known face image predates a test image. In the blended face space, the known set is sampled from fifteen known Caucasian faces and fifteen known African-American faces. The blended also holds to the real world model where the test set are the older images of the corresponding known set.

After the Caucasian, African-American and blended face spaces are trained, then classification tests are performed. Two classification metrics, Euclidean distance (L2) with threshold and nearest neighbors using Mahalanobis distance, are evaluated against each other Both methods begin with projecting probe (unknown) faces into face space, then an evaluation metric classifies the image. The following describes two classification metrics:

4.4 Euclidean Metric with Threshold

The Euclidean distance with threshold was the first classification metric applied to the classification system. The method begins by calculating the Euclidean distance measures using the extracted coefficients between the training faces and average face of the Caucasian and African-American face spaces.

$$L2 = \sum_{i=1}^{N} \left(t_i - \bar{x}_i\right)^2 \text{where}, N = 16 \qquad (5)$$

and t_i and $\bar{x}_i$ are the ith coefficient of training and average face, respectively

An average Euclidean distance (Equation 6) and standard deviation (Equation 7) measures are calculated and stored for Caucasian and African-American face spaces.

$$\bar{L}2 = \frac{1}{M}\sum_{i=1}^{M} L2_i \text{ where M = number of training distances} \qquad (6)$$

$$\sigma = \sqrt{\frac{1}{M}\sum_{i=1}^{M}\left(L2_i - \bar{L}2\right)^2} \qquad (7)$$

An unknown face is then projected into the two face spaces and its Euclidean similarity measure is computed from average face. Two approaches using the Euclidean distance were implemented. In the first, if a projected unknown face had a measure that was less than the sum of the average Euclidean measure and standard deviation for a respective face space, then the face was classified as that race.

$$L2' < \overline{L}2 + \sigma \text{ where } L2' \text{ is the probe distance from the average face}$$

In the second, if a projected unknown face had a distance that fell between a range, plus one standard deviation and minus one standard deviation away from the mean, then that face was classified as the corresponding race.

$$\overline{L}2 - \sigma < L2' < \overline{L}2 + \sigma$$

However, a face could be classified under the Euclidean metric as one of the following:

A face could fall into only one classification either Caucasian or African-American face

A face can be cross-classified as both Caucasian and African-American face

A face can be classified as unknown and not pass either classification metric

The average results the two tests, African-American and Caucasian classification are shown in Tables 3a. and 3b., respectively.

Table 3a. Results of 50 African American classification tests using the Euclidean distance measure

Classification metric for African American Tests	**Classified correctly**	**False classification**	**Cross classification**	**Classified unknown**
L2 w/ single threshold	62.0%	16.0%	11.3%	10.7%
L2 w/ std. deviation range	60.0%	10.7%	12.7%	16.7%

Table 3b. Results of 50 Caucasian 50 classification tests

using the Euclidean distance measure

Classification metric for Caucasian Tests	**Classified correctly**	**False classification**	**Cross classification**	**Classified unknown**
L2 w/ single threshold	74.0%	9.3%	10.0%	6.7%
L2 w/ std. dev. range	80.0%	3.3%	10.0%	6.7%

The results displayed above were averaged over **50 classification tests** using randomly sampled training and testing data at each test run. Running concurrent with the classification tests were identification tests. During the tests, a probe face would first pass through the classification system and then if classified correctly, the probe would pass through the identification system. Each test ran ten iterations where 60 probe faces were used which came from the 30 faces of both, African-American and Caucasian test sets. The data shows that classification rates are better using the L2 (Euclidean) technique for a Caucasian probe. Notice that in both test groups, African-American and Caucasian, false classification rates were reduced when using L2 with standard deviation range. Paradoxically, the African-American test group showed lower correct classification rates when using L2 with standard deviation, but higher cross-classification rates and unknown rates. Using the L2 with standard deviation metric in the Caucasian test group increased correct classification rates at the same scale in which false classification rates decreased. Remarkably, the cross-classification and unknown rates stayed the same.

4.5 Nearest Neighbors using Mahalanobis Distance

After evaluating the lower than desired classification rates using L2 with threshold and the high rates of cross classification, a method was chosen to alleviate cross

classification and the unknown factor. The nearest neighbors voting scheme uses the Mahalanobis distance (Equation 8) as the metric instead of L2. The Mahalanobis distance is a useful way to determine the race similarity of an unknown face to the known sample set of faces. Mahalanobis distance differs from Euclidean in that it takes into account the correlations of the data set and scale is invariant, meaning that high magnitude values in a particular dimension will not influence the measure.

$$d(\vec{x},\vec{y}) = \sqrt{(\vec{x}-\vec{y})^T \Sigma^{-1}(\vec{x}-\vec{y})} \quad (8)$$

where $\vec{x},\vec{y}$ are two vectors of coefficient data and Σ is the covariance matrix

Calculating the Mahalanobis distance metric begins by projecting an unknown face into the blended face space. The Mahalanobis distance measure is calculated as the difference between probe face and all known faces. These distance are ranked in ascending order. Four separate voting schemes were evaluated using this Mahalanobis metric and all schemes alleviate cross classification, only allowing face to be classified as either Caucasian or African-American but not both.

The nearest neighbor is chosen and its corresponding race classifies the unknown face Three nearest neighbors are chosen each having one vote and the race with the higher amount of votes classifies the unknown image. Five nearest neighbors are chosen each having one vote and the race with more votes classifies the probe. Five nearest neighbors are chosen each have a vote relative to their rank. The nearest neighbor has 5 votes, the next closest has 4, the next has 3 and the other two follow the same pattern. The race with more votes classifies the unknown face. The results of the four tests are shown in Table 4 (all classification rates are based on the number of correctly classified out of the test set):

Table 4. Classification rates using Mahalanobis distance and the four different voting

schemes

Classification rate	**Nearest neighbor classifies**	**Nearest three neighbors with single vote**	**Nearest five neighbors with single vote**	**Nearest five neighbors with weighted vote**
Caucasian test face set	93.3%	94.4%	95.5%	95.5%
African-American test face set	90.0%	90.0%	92.2%	90.0%

The classification rates using the nearest neighbors voting scheme increased sufficiently in comparison to the Euclidean metrics. One reason is that the Mahalanobis distance is based on correlations of variables by which different patterns can be identified. In the case of FR, the variables are the face features and the patterns are the specific arrangement of features on a face. Another reason is that Mahalanobis distance is scale invariant, so large coefficient values do not influence the distance measure as with Euclidean. However, the classification rate comparison of African-American versus Caucasian groups shows that, again, the Caucasian test set has a higher classification performance. Since this metric looks at nearest neighbors, then an observation is made about the blended feature space. The Caucasian population generally, is populated more densely than that of the African-American subgroup which implies higher classification rates. Overall, the nearest neighbor voting metric using Mahalanobis distance performs better as a classifier of race than that of L2.

4.6 Identification

Identification tests are evaluated in three face spaces, Caucasian, African-American and blended. In the Caucasian and African-American face spaces, a probe is first

classified then projected into race specific face space for identification. However, if the face is falsely classified then it cannot be identified and the system has failed. The main goal of this system is to classify correctly to alleviate any system failure. The identification metric uses Euclidean distance where a probe is projected into either Caucasian or African-American face space and distances are ranked in ascending order. A face is identified when its rank falls in the top 10% of a test set of N faces. Under the current configurations of the system, this means that a probe has to be ranked either 1, 2 or 3 (10% 30 test faces) to considered identified. Identification rates are calculated by the number of faces identified out of a test set. The identification rates are compared to those of the blended face space which acts as the baseline for comparison against the racially tuned identification system. Table 5 shows the results of all identification systems. The average rank is calculated by dividing the sum of all ranked values for a specific face space over the number of ranked images.

Table 5. Identification rates for the race-specific and general face identification systems

Identification tests	**Identification rates**	**Average rank**
African-American test set	53.3%	7.1
Caucasian test set	55.6%	6.2
Blended test set	23.7%	10.1

Merely glancing at Table 5, one might think that the identification rates are extremely low but, when compared to the baseline technique of the general FR system they have increased greatly. The identification rates for African-American and Caucasian face spaces double that of the Blended face space. Since the population of images sampled for the system, were from the MORPH database then these uncontrolled images have a great affect on both classification and identification rates. Many times in the face images,

hot spots are found. A hot spot is a term related to extreme light intensity on a certain region of the face. The hot spots can be misinterpreted by PCA for face features just like facial hair. If a known image projected into face space, has a hot spot and the corresponding probe face does not, then that image may not be identified or even classified correctly. Controlling the variability in a face scene is difficult once a photograph has already been taken.

The identification data in Table 5 was created by running identification tests only, using a random set of training and testing faces for all iterations. During the test runs, classification was not performed thus no classification error is involved with Table 4.4's statistics. Following the same procedure as the classification tests, 50 identification tests were ran where an average of the fifty rates are shown. At a comprehensive look of the identification system, an observation can be made that modeling the own-race effect in software does increase identification rates when compared to the baseline.

5.0 CONCLUSION AND FUTURE WORK

Although the project explored the PCA algorithm and its capabilities to perform racial classification and face identification, there are many other FR algorithms that could be implemented and tested. Linear Discriminate Analysis (LDA), Fisher Discriminate Analysis (FDA), and other feature based FR algorithms could apply to the context of this work. Other metrics for instance negative cosine angle could be applied to the classification or identification system to test its influence on the performance of the system. In addition, a combination of metrics could be applied to evaluate any increase in classification or identification rates. The use of another face data corpus should be used as the population set. Data corpora like the FERET database were looked at for its

use in the current specifications of the project however; the database lacked the meta-data vital to race classification. Finally, the project should be tested against commercial FR software using the same sample population from the MORPH database; this should show a performance deficit in the commercial software due to the uncontrolled.

In this project a proposed method to increase the efficacy of a FR system using the neurocognitive model known as own-race bias has been presented. Principal Component Analysis has been shown to be a very simple yet powerful tool for this purpose. The PCA algorithm reduces the dimensionality of image space to feature space, which helps to model the hypotheses of many psychologists which state that human face recognition is done using only a few exemplars. To further support the own-race effect, a face classifier is built in which provides a relatively high classification rate. While there is not a single measure which is best for all case, in general the Mahalanobis distance metric using nearest neighbors outperforms Euclidean. While it was critical to have the meta-data associated with the MORPH data corpus, the project face sample provides a real world challenge. In all, the project was a success.

BIBLIOGRAPHY

[1] S. Chen, B.C. Lovell, T. Shan. "Combining Generative and Discriminative Learning for Face Recognition." *IEEE Proceedings of the Digital Imaging Computing: Techniques andApplications.* University of Queensland. Brisbane, Australia © 2005

[2] M. I. Gobbini, J. V. Haxby. "Neural systems for recognition of familiar faces." *Elsevier Ltd. Neuropsychologia*. Princeton University. New Jersey, US © 2006

[3] S. Ghutta, H. Wescler. "Network Ensembles for Facial Analysis Tasks." *IEEE Proceedings of the International Joint Conference on Neural Networks*. Briarcliff Manor, New York. George Mason University, Fairfax, VA. © 2000

[4] R. Gottumukkal, V. K. Asari. "System Level Design of Real Time Face Recognition Architecture Based on Composite PCA." *GLSVLSI '03 ACM.* Old Dominion University. Washington, DC. ©2003

[5] V. Herrera, D. E.McQuiston, O. H. MacLin, R. S. Malpass. "Examining the Cross Race Effect Using Racially Ambiguous Faces."2000. University of Texas, El Paso.<http://eyewitness.utep.edu/Documents/Herrera%20WPA%202000.pdf>

[6] Y. Ivanov, B. Heisele, T. Serre. "Using Component Features for Face Recognition." *Proceedings of the Sixth International Conference on Automatic Face and Gesture Recognition*. Honda Research Institute. Boson, MA. © 2004

[7] A. Jain, J. Huang. "Integrating Independent components and Linear Discriminant Analysis for Gender Classification." *Proceedings of the Sixth IEEE International Conference on Automatic Face and Gesture Recognition*. Indiana University-Purdue University-Indianapolis, IN © 2004

[8] S. Lawrence, C. L. Giles, A. C. Tsoi. "Convolutional Neural Networks For Face Recognition." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. Princeton University, NJ. University of Queensland, St. Lucia, Australia. © 1996

[9] C. Liu, H. Weschler. "Learning the Face Space – Representational and Recognition." *Proceeding sof the IEEE International Conference on Pattern Recognition*. University of Missouri – St. Louis, MO. George Mason University – Fairfax, VA. © 2000

[10] C. Liu, H. Weschler. "Probabilistic Reasoning Models for Face Recognition." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. George Mason University –Fairfax, VA. © 1998

[11] X. Liu, T. Chen, J. Rittscher. "Optimal Pose for Face Recognition." *Proceedings of*

the IEEE Computer Society Conference on Computer Vision and Pattern Recognition. General Electric Global Research Center-Scenectady, NY. Carnegie Mellon University-Pittsburg, PA. © 2006

[12] J. Lu, X. Yuan, T. Yahagi. "A method of race recognition based on fuzzy clustering and parallel neural networks." *Elsevier B.V.-Signal Processing.* Chiba University, Japan © 2005

[13] A. J. O'Toole, S. Edelman. "Face distinctivess in recognition across viewpoint: An analysis of the statistical structure of face spaces." *Proceedings of the 2nd IEEE International Conference on Automatic Face and Gesture Recognition.* University of Texas at Dallas-Richardson, TX. The Weizmann Institute of Science-Rehovot, Israel. © 1996

[14] S. Ranganath, K. Arun. "Face Recognition Using Transform Features and Neural Networks." *Elsevier Science Ltd.-Pattern Recognition.* Vol. 30. No. 10. pp. 1615-1622. National University of Singapore, Singapore. Great Britain © 1997

[15] S. Romdhani. "Face Recognition Using Principal Components Analysis." *Masters Thesis.* University of Glasgow, United Kingdom. 1997

[16] S. Sangrigoli, C. Pallier, A.-M. Argenti, V.A.G. Ventureyra, S. de Schonen. "Reversibility of the other-race effect in face recognition during childhood." *Manuscript in press in Psychological Science*. Boulogne-Billancourt, France. Orsay, France. Paris France. © 2004

[17] Z. Su, S. Li, H. Zhang. "Extraction of Feature Subspaces for Content-Based Retrieval Using Relevance Feedback." *In Proceedings of the 9th ACM Multimedia*. Tsinghua University-Beijing, China. Microsoft Research-Beijing, China. © 2001

[18] J. W. Tanaka, M. Keiefer, C. M. Bukack. "A holistic account of the own-race effect in face recognition: evidence from a cross-cultural study." *Elsevier B.V.-Cognition*. University of Victoria-British Colombia, Canada. University of Ulm-Ulm, Germany. Vanderbilt University- Nashville, TN © 2004

[19] X. Wang, X. Tang. "A Unified Framework for Subspace Face Recognition." *IEEE Transactions onPattern Analysis and Machine Intelligence.* Vol. 26. No. 9. Chinese University of Hong Kong © 2004

[20] H. Weschler. "Face Processing and Recognition Using Learning and Evolution." *Proceedings of the IEEE International Conference on Image Analysis and Processing*. pp. 254-257. George Mason-Fairfax, VA. © 1999

[21] J. Yang, D. Zhang. A. F. Frangi, J-y Yang. "Two Dimensional PCA: A New Approach to Appearance-Based Face Representation and Recognition." *IEEE Transactions on Pattern Analysis and Machine Intelligence.* Vol. 26. No. 1. © 2004

[22] D. Zhang, Z-H. Zhou. “PCA: Two-directional two-dimensional PCA for efficient face representation and recognition.” *Elsevier B.V.-Neurocomputing.* Nanjing University-Nanjing, China. © 2005

[23] W. Zhao, R. Chellappa, P.J. Phillips, A. Rosenfeld. “Face Recognition: A Literature Survey.” *ACM Computing Surveys.* Vol. 35. No. 4. pp. 399-458. Sarnoff Corporation. University of Maryland. National Institute of Standards and Technology. © 2003

[24] *Wikipedia: The Free Encyclopedia*. Online Encyclopedia. <http://en.wikipedia.org/wiki/Race>

[25] K. Pezdek, I. Blandon-Gitlin, C. Moore. “Children's Face Recognition Memory: More Evidence for the Cross-Race Effect” *Journal of Applied Psychology.* Vol. 88. No. 4. pp. 760-763.Claremont Graduate University. © 2003

[26] J. F. Cross, J. Cross, J. Daly. “Sex, Race, Age, and beauty as factors in recognition of faces.” *Perception & Psychological.* 10 (6), 393-396.

[27] S. Feinman, D. R. Entwisle. “Children's ability to recognize other children's faces.” *Child Development*. 47, 506-510.

[28] S. J. Platz, H. M. Hosch. “Cross-racial/ethnic eyewitness identification: A field study.” *Journal of Applied Social Psychology*. 18 (11), 972-984

[29] J. E. Chance, A. L. Turner, A. G. Goldstein. “Development of differential recognition for own- and other-race faces.” *The Journal of Psychology*. 112, 29-37

[30] D. T. Levin, M. R. Banaji. “Distortions in the Perceived Lightness of Faces: The Role of Race Categories” *Journal of Experimental Psychology: General*. 2006, 135(4), 501-51 2.

[31] D.T. Levin,” Race as a visual feature: Using visual search and perceptual discrimination tasks to understand face categories and the cross-race recognition deficit” *Journal of Experimental Psychology: General*. 2000, 129(4), 559-574.

[32] Open Source Computer Vision Library <http://www.intel.com/technology/computing/opencv/>

[33] A. Kembhavi. “Face Recognition” *Pattern Recognition ENEE-633*. 2005

[34] W. S. Yambor, B. A. Draper, J. R. Beveridge. “Analyzing PCA-based Face Recognition Algorithms: Eigenvector Selection and Distance Measures” *World Scientific Press Singapore*.2004

APPENDIX A

SOFTWARE SOURCE CODE

```
/*************************************************************************
 * System: Face Classification Recognition System.
 *
 * Software for Thesis:
 *          NEUROCOGNITIVE INSPIRED HIERARCHICAL FACE RECOGNITION SYSTEM
 *
 * File name: FCRS.h
 *
 * Input file name(s):  highgui.h
 *                                  cv.h
 *                                  stdio.h
 *                                  fstream.h
 *                                  cstring.h
 *                                  string.h
 *                                  stdlib.h
 *                                  time.h
 *                                  unistd.h
 *                                  math.h
 *
 * Output file name(s):
 *
 * Author: Ryan Wilkins
 * Date Last Modified: 07/01/07
 * Purpose:
 *
 *************************************************************************/
#ifndef FCRS_
#define FCRS_

#include <highgui.h>
#include <cv.h>
#include <iostream>
#include <stdio.h>
#include <fstream>
#include <FaceSpace.h>
#include <FaceDatabase.h>
#include <FaceReader.h>
#include <FaceIdentifier.h>
#include <FaceClassifier.h>
#include <cstring>
#include <string>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <math.h>

#define NUM_CA_TESTERS 30
#define NUM_AA_TESTERS 30
#define NUM_BLENDED_TESTERS 30

#define NUM_BLENDED_TRAINERS 30
#define NUM_AA_TRAINERS 30
#define NUM_CA_TRAINERS 30

#define NUM_BLENDED_KNOWN 30
#define NUM_AA_KNOWN 30
#define NUM_CA_KNOWN 30
```

```
#define NUM_EIGENS 16

#define CA 0
#define AA 1
#define BOTH 3
#define UNKNOWN 4

class FCRS
{


public:

      FCRS();
      virtual ~FCRS();

      int showSimpleFace(::IplImage* f);
      int showSimpleFace(::IplImage* f, char* title);
      int showEigenFaces();
      int showAvgAAFace();
      int showAvgCAFace();
      int showAvgBlendedFace();
      int showAAProjections();
      int showCAProjections();
      int showBlendedProjections();

      void initFaceReader();
      void initDatabase();
      void initAAFaceSpace();
      void initCAFaceSpace();
      void initFaceClassifier();

      void createTestingSet();
      void runAAIdentificationSet();
      void runCAIdentificationSet();
      void initCATestSet();
      void initAATestSet();
      void initBlendedFaceSpace();
      void ClassifyTestSetInRange();
      void ClassifyTestSet();
      void PrintTestCoeffs();
      void PrintTestSets();
      int ClassifyImage(::IplImage *probe );
      int ClassifyImageWithRange(::IplImage* probe);
      void Identify( ::IplImage* probe , FaceClassifier* fc );
      void RankImageInAA( ::IplImage* probe );
      void RankImageInCA( ::IplImage* probe );
      void Sort( float* fd, int* faces, int size );
      void Swap( int x, int y, float* a );
      void Swap( int x, int y, int* a );
      void initTestSetIndices();
      void initBlendedTestSetIndices();
      void runBlendedIdentification();
      void runAAClassificationIdentificationSet();
      void runCAClassificationIdentificationSet();
      void runAAComponentClassification();
```

```
      void runCAComponentClassification();
      void runCAClassificationTests();
      void runAAClassificationTests();
      void SetupTest();

      FILE* outFile;
      ::IplImage* test_set[NUM_AA_TESTERS + NUM_CA_TESTERS];
      int test_set_indices[NUM_AA_TESTERS];

};


#endif /*FCRS_*/

/*************************************************************************
 * System: Face Classification Recognition System.
 *
 * Software for Thesis:
 *          NEUROCOGNITIVE INSPIRED HIERARCHICAL FACE RECOGNITION SYSTEM
 *
 * File name: FCRS.cpp
 *
 * Input file name(s):  FCRS.h
 *                                  FaceReader.h
 *                                  FaceDatabase.h
 *                                  FaceSpace.h
 *
 * Output file name(s): main_out
 *
 * Author: Ryan Wilkins
 * Date Last Modified: 07/01/07
 * Purpose: The driver class of the face recognition system
 *
 ************************************************************************/


#include "FCRS.h"
#include "FaceReader.h"
#include "FaceDatabase.h"
#include "FaceSpace.h"

/*  #include <gtk/gtk.h>  */

/**
 *    FCRS.cpp (Face Classification Recognition System) is the driver
class the spawns
 *  all actions and methods which run identification and classifcation
tests.  This class
 *  is the root of the software and centralized location to handle all
over classes.
 *
 */


using namespace std;

      FaceReader fr;
```

```
	FaceSpace aa_fs;
	FaceSpace ca_fs;
	FaceSpace bl_fs;
	FaceDatabase fdb;

	/**
	 * This is the constructor that instantiates the class.
	 *
	 */

FCRS::FCRS()
{
	if((outFile = fopen("output/main_out", "w")) == NULL)
		printf(" error creating file\n");

	fprintf( outFile, "\n*** Welcome to my Face Project ***\n");
}

	/**
	 * This is the deconstructor which handles memory clean up and is
called
	 * when the program exits.
	 *
	 */

FCRS::~FCRS()
{

}

	/**
	 * The method, showSimpleFace(), displays a IplImage passed as a
parameter
	 * and gives a title to the window that displays the image.
	 *
	 * @param IplImage* f -- image to display
	 * @param char* title -- title that is set on the window
	 */

int FCRS::showSimpleFace(::IplImage* f, char* title){

	::cvNamedWindow( title, 1);

	::cvShowImage( title, f );

	::cvWaitKey(0);

	::cvReleaseImage( &f );

	return 0;

}

	/**
	 * The method, showSimpleFace(), displays a IplImage passed as a
parameter
	 *
```

```
	 * @param IplImage* f -- image to display
	 */


int FCRS::showSimpleFace(::IplImage* f){

	::cvNamedWindow( "Image Viewer", 1);

	::cvShowImage( "Image Viewer", f );

	::cvWaitKey(0);

	::cvReleaseImage( &f );

	return 0;
}

	/**
	 * The method, showEigenFaces, displays all of the eigenfaces that
have already
	 * been calculated. Displays the African American, Caucasian and
Blended
	 * eigenfaces
	 */

int FCRS::showEigenFaces(){
	int i;

	char* title = "AA Eigen";
	for(i = 0; i < NUM_EIGENS; i++)
		showSimpleFace(aa_fs.eigen_faces[i], title);

	title = "CA Eigen";
	for(i = 0; i < NUM_EIGENS; i++)
		showSimpleFace(ca_fs.eigen_faces[i], title);

	title = "BL Eigen";
	for(i = 0; i < NUM_EIGENS; i++)
		showSimpleFace(bl_fs.eigen_faces[i], title);

	return 0;
}

	/**
	 * The method, showAvgAAFace, displays the average African
American
	 * face of the face space.
	 */

int FCRS::showAvgAAFace(){
	showSimpleFace( aa_fs.aa_avg_face );
	return 0;
}

	/**
	 * The method, showAvgCAFace, displays the average Caucasian face
	 * of its corresponding face space.
```

```
       */

int FCRS::showAvgCAFace(){
      showSimpleFace( ca_fs.ca_avg_face );
      return 0;
}

      /**
       * The method, showAvgBlendedFace, displays the average Blended
face
       * for the Blended face space
       */

int FCRS::showAvgBlendedFace(){
      showSimpleFace( bl_fs.blended_avg_face );
      return 0;
}

      /**
       * The method, showAAProjections, displays the African American
faces
       * projected into its face space
       */

int FCRS::showAAProjections(){
      int i;
      for(i = 0; i < NUM_AA_TRAINERS; i++)
            showSimpleFace(aa_fs.aa_proj[i]);

      return 0;
}

      /**
       * The method, showCAProjections, displays the Caucasian faces
that
       * have been projected into its face space.
       */

int FCRS::showCAProjections(){
      int i;
      for(i = 0; i < NUM_CA_TRAINERS; i++)
            showSimpleFace(ca_fs.ca_proj[i]);

      return 0;
}

      /**
       * The method, showBlendedProjections, displays all of the
projected faces
       * in the blended face space.
       */

int FCRS::showBlendedProjections(){
      int i;
      for(i = 0; i < NUM_BLENDED_TRAINERS; i++)
            showSimpleFace(bl_fs.blended_proj[i]);
```

```
	return 0;
}

	/**
	 * The method, createTestingSet, pulls images out of the
FaceDatabase and allocates
	 * them to a test set of faces.  The set of test faces are merely
an array of IplImages
	 */

void FCRS::createTestingSet()
{
	int i, j;

	fprintf( outFile, "\n\tUnranked Distances of identification by
classification\n" );

	for( i = 0; i < NUM_AA_TESTERS; i++ ){

		test_set[i] = ::cvCloneImage( fdb.all_faces[i*2+1] );


	}

	i = 0;
	for( j = NUM_AA_TESTERS; j < NUM_AA_TESTERS+NUM_CA_TESTERS; j++ ){
		test_set[j] = ::cvCloneImage( fdb.all_faces[(i*2 +
fdb.aa_faces)+1] );
		i++;
	}
}

	/**
	 * The method, inSet, receives an integer array, the end point of
the array and the
	 * integer in question. It returns whether or not the integer
passed in (r) is in
	 * the integer array or not.
	 *
	 * @param int[] set -- array of integers
	 * @param int r -- the integer in question
	 * @param int end --  the end point of the search
	 * @return bool -- the boolean true or false.
	 */

bool inSet( int set[], int end, int r ){

	bool inset = false;
	int i;

	for( i = 0; i < end; i++ ){
		if( set[i] ==     r ){
			inset = true;
		}
	}
	return inset;
}
```

```
	/**
	 * The method, initTestSetIndices, initializes a test set of
indices which
	 * are generated at random from 1 to the number of number of faces
in the
	 * FaceDatabase.
	 */

void FCRS::initTestSetIndices()
{
	int i, m;
	int r;
	m = NUM_AA_TESTERS + NUM_CA_TESTERS;

	::srand( time(NULL) + getpid() );

	for( i = 0; i < 30; i++ ){
		r = (int)((((double)::rand() / (double)RAND_MAX) * m)+0.5);

		if( !inSet( test_set_indices, i, r )){
			test_set_indices[i] = r;
//			fprintf( outFile, "%i \n",test_set_indices[i] );
		}else{
			i--;
		}

	}

}

	/**
	 * The method, initBlendedTestSetIndices, initializes a set of
indices by grabbing
	 * an equal amount of indices for both the Caucasian and African
American space.
	 */

void FCRS::initBlendedTestSetIndices(){

	int i, j;

	for( i = 0; i < 60; i++ ){
		if( i < 15 ){
			test_set_indices[j] = i;
			j++;
		}else if( i > 29 && i < 45 ){
			test_set_indices[j] = i;
			j++;
		}
	}

}

	/**
	 * The method, getRaceCode, returns the race as a char* by the
given integer representation
```

```
       * of that format.
       *
       * @param int i -- the integer representation of the character
code
       * @return char* -- the character representation of the integer
passed as a parameter
       */

char* getRaceCode( int i ){
      char* rc;
      switch( i ){

      case CA:
            rc = "CA";
      break;

      case AA:
            rc = "AA";
      break;

      case BOTH:
            rc = "BOTH";
      break;

      case UNKNOWN:
            rc = "UNK";
      break;

      }

      return rc;
}

      /**
       * The method, runCAIdentificationSet, spawns all of the processes
needed to run
       * a Caucasian identification test.  It first projects known
images into the Caucasian
       * space, it then evaluates the distances the probe is away from
the face.
       * Then an output is written to the log which displays the rank of
the probe image.
       */

void FCRS::runCAIdentificationSet(){

      int i, j, rank;
      char* race;

      FaceIdentifier fi( &aa_fs, &ca_fs, &bl_fs, &fdb );
      fi.ProjectKnownImages();
      fi.PrintKnownCoeffs();

      fprintf( outFile, "\nRunning CA Identification Test\n" );

      fprintf( outFile, "Training CA,\t" );
```

```
	for( i = 0; i < NUM_CA_TRAINERS; i++ ){
		if( i == NUM_CA_TRAINERS - 1)
			fprintf( outFile, "%s, \n", fr.ca_train_fn[i].c_str()
);
		else
			fprintf( outFile, "%s, ", fr.ca_train_fn[i].c_str() );
	}

	fprintf( outFile, "Target CA,\t" );

	for( i = 0; i < NUM_CA_KNOWN; i++ ){
		fprintf( outFile, "%s, ", fr.ca_known_fn[i].c_str() );

	}

	fprintf( outFile, "Rank, ");

	::IplImage* probe = NULL;

	std::string file_name;
	for( i = 0; i < NUM_CA_KNOWN; i++ ){

		file_name = fr.ca_test_fn[i];

//		showSimpleFace( ::cvCloneImage( test_set[index] ) );

		probe = ::cvCloneImage( fdb.ca_test_set[i] );

		fi.RankImageInCA( probe );

		rank = fi.GetImageRankCA(i);

		race = "CA";

		fprintf( outFile ,"\n%s,%s,", file_name.c_str(), race );

		for( j = 0; j < NUM_CA_TRAINERS; j++ ){
			fprintf( outFile, "%3.0f,", fi.unrank_dist_ca[j] );

		}

		rank++;
		fprintf( outFile ,"%i, ", rank );
	}



}

	/**
	 * The method, runAAIdentificationSet, spawns all of the processes
needed to run
	 * a African American identification test.  It first projects
known images into the AA
	 * space, it then evaluates the distances the probe is away from
the face.
```

```
       * Then an output is written to the log which displays the rank of
the probe image.
       */

void FCRS::runAAIdentificationSet(){

      int i, j, rank;
      char* race;

      FaceIdentifier fi( &aa_fs, &ca_fs, &bl_fs, &fdb );
      fi.ProjectKnownImages();
      fi.PrintKnownCoeffs();

      fprintf( outFile, "\nRunning AA Identification Test\n" );

      fprintf( outFile, "Training AA,\t" );

      for( i = 0; i < NUM_AA_TRAINERS; i++ ){
            if( i == NUM_AA_TRAINERS - 1)
                  fprintf( outFile, "%s, \n", fr.aa_train_fn[i].c_str()
);
            else
                  fprintf( outFile, "%s, ", fr.aa_train_fn[i].c_str() );
      }

      fprintf( outFile, "Target AA,\t" );

      for( i = 0; i < NUM_AA_KNOWN; i++ ){
            fprintf( outFile, "%s, ", fr.aa_known_fn[i].c_str() );

      }

      fprintf( outFile, "Rank" );

      ::IplImage* probe = NULL;

      std::string file_name;
      for( i = 0; i < NUM_AA_KNOWN; i++ ){

            file_name = fr.aa_test_fn[i];

//          showSimpleFace( ::cvCloneImage( test_set[index] ) );

            probe = ::cvCloneImage( fdb.aa_test_set[i] );

            fi.RankImageInAA( probe );
            rank = fi.GetImageRankAA( i );

            race = "AA";

            fprintf( outFile ,"\n%s,%s,", file_name.c_str(), race );

            for( j = 0; j < NUM_AA_TRAINERS; j++ ){
                  fprintf( outFile, "%3.0f,", fi.unrank_dist_aa[j] );

            }
```

```
            rank++;
            fprintf( outFile, "%i, ", rank );
      }



}

      /**
       * The method, runAAClassificationIdentificationSet, spawns all
the processes to classify
       * a probe and then identify it.  It first projects know images
into the AA face space.
       * Secondly, it evaluates the distances of a probe to the known
images and by a standard
       * metric classifies the face.  Once classification takes place
then the face can be
       * identified against the face space in which it was classified.
       */

void FCRS::runAAClassificationIdentificationSet()
{

      int i, j, m, race_class, index, rank;
      char* race;
      char* c_race;

      fprintf( outFile, "\n\tRunning Identification Tests\n" );

      FaceClassifier fc( &aa_fs , &ca_fs, &bl_fs, &fdb );
      FaceIdentifier fi( &aa_fs, &ca_fs, &bl_fs, &fdb );
      fi.ProjectKnownImages();
      fi.PrintKnownCoeffs();

      fprintf( outFile, "\n\t\tRunning Classification Identification
Test\n" );

      fprintf( outFile, "Training AA,\t" );

      for( i = 0; i < NUM_AA_TRAINERS; i++ ){
            if( i == NUM_AA_TRAINERS - 1)
                  fprintf( outFile, "%s, \n", fr.aa_train_fn[i].c_str()
);
            else
                  fprintf( outFile, "%s, ", fr.aa_train_fn[i].c_str() );
      }

      fprintf( outFile, "Target AA,\t" );
      for( i = 0; i < 30; i++ )
      {
            fprintf( outFile, "%s, ", fr.aa_known_fn[i].c_str());
      }

      fprintf( outFile, "Rank, " );
//    fprintf( outFile ,"\nTarget CA, \t" );
//    for( i = 0; i < 30; i++ )
//    {
```

```
//          fprintf( outFile, "%s, ", fr.ca_known_fn[i].c_str() );
//    }
//    fprintf( outFile, "\n\tRace,\tClass,");
      std::string file_name;
      ::IplImage* probe = NULL;
      for( i = 0; i < NUM_AA_TESTERS; i++ ){

            file_name = fr.aa_test_fn[i];

//          showSimpleFace( ::cvCloneImage( test_set[index] ) );

            probe = ::cvCloneImage( fdb.aa_test_set[i] );

            race_class = fi.Identify( probe , &fc );

            if( race_class == AA )
                  rank = fi.GetImageRankAA(i);
            else
                  rank = -1;

//          race_class = fc.ClassifyImage( test_set[index] );

            c_race = getRaceCode( race_class );
//
            race = "AA";

            fprintf( outFile ,"\n%s,%s,%s,", file_name.c_str(), race,
c_race );


            for( j = 0; j < NUM_AA_KNOWN; j++ ){
                  fprintf( outFile, "%3.0f,", fi.unrank_dist_aa[j] );

            }

            rank++;

            if( race_class == AA )
                  fprintf( outFile, "%i, ", rank );

      }



}

      /**
       * The method, runCAClassificationIdentificationSet, spawns all
the processes to classify
       * a probe and then identify it.  It first projects know images
into the CA face space.
       * Secondly, it evaluates the distances of a probe to the known
images and by a standard
       * metric classifies the face.  Once classification takes place
then the face can be
       * identified against the face space in which it was classified.
       */
```

```
void FCRS::runCAClassificationIdentificationSet()
{

	int i, j, m, race_class, index, rank;
	char* race;
	char* c_race;

	fprintf( outFile, "\n\tRunning Identification Tests\n" );

	FaceClassifier fc( &aa_fs , &ca_fs, &bl_fs, &fdb );
	FaceIdentifier fi( &aa_fs, &ca_fs, &bl_fs, &fdb );
	fi.ProjectKnownImages();
	fi.PrintKnownCoeffs();

	fprintf( outFile, "\n\t\tRunning Classification Identification
Test\n" );

//	fprintf( outFile, "Target AA,\t" );
//	for( i = 0; i < 30; i++ )
//	{
//		fprintf( outFile, "%s, ", fr.aa_known_fn[i].c_str());
//	}
//
	fprintf( outFile, "Training CA,\t" );

	for( i = 0; i < NUM_CA_TRAINERS; i++ ){
		if( i == NUM_CA_TRAINERS - 1)
			fprintf( outFile, "%s, \n", fr.ca_train_fn[i].c_str()
);
		else
			fprintf( outFile, "%s, ", fr.ca_train_fn[i].c_str() );
	}

	fprintf( outFile ,"\nTarget CA, \t" );
	for( i = 0; i < 30; i++ )
	{
		fprintf( outFile, "%s, ", fr.ca_known_fn[i].c_str() );
	}
	fprintf( outFile, "Rank, " );

//	fprintf( outFile, "\n\tRace,\tClass,");
	::IplImage* probe = NULL;
	std::string file_name;
	for( i = 0; i < NUM_CA_TESTERS; i++ ){

		file_name = fr.ca_test_fn[i];

//		showSimpleFace( ::cvCloneImage( test_set[index] ) );

		probe = cvCloneImage( fdb.ca_test_set[i] );

		race_class = fi.Identify( probe , &fc );

		if( race_class == CA )
			rank = fi.GetImageRankCA(i);
		else
```

```
                  rank = -1;

//          race_class = fc.ClassifyImage( test_set[index] );

            race = "CA";

            c_race = getRaceCode( race_class );
//
            fprintf( outFile ,"\n%s,%s,%s,", file_name.c_str(), race,
c_race );


            for( j = 0; j < NUM_CA_KNOWN; j++ ){
                  fprintf( outFile, "%3.0f,", fi.unrank_dist_ca[j] );

            }

            rank++;

            if( race_class == CA )
                  fprintf( outFile, "%i, ", rank );

      }



}

      /**
       * The method, runAAComponentClassification, handles all the
methods to which
       * classification is achieved by only examining the 1st, 2nd and
3rd principle
       * components of the African American space.
       */

void FCRS::runAAComponentClassification()
{
      int i, j, n, rc_lambda1, rc_lambda2, rc_lambda3, rc_lambda12,
rc_lambda13, rc_lambda23, index, rank;
      char* race;
      char* c_race;
      int one_component[1];
      int two_component[2];

      FaceClassifier fc( &aa_fs , &ca_fs, &bl_fs, &fdb );

      fprintf( outFile, "\n\t\tRunning Component Classification Test\n"
);

      fprintf( outFile, "Training AA,\t" );

      for( i = 0; i < NUM_AA_TRAINERS; i++ ){
            if( i == NUM_AA_TRAINERS - 1)
                  fprintf( outFile, "%s, \n", fr.aa_train_fn[i].c_str()
);
            else
```

```
                fprintf( outFile, "%s, ", fr.aa_train_fn[i].c_str() );
        }

        fprintf( outFile, "Unknown AA,\t" );

        fprintf( outFile,
"Race,Lambda1,Lambda2,Lambda3,Lambda12,Lambda13,Lambda23");

        std::string file_name;
        ::IplImage* probe = NULL;

        for( i = 0; i < NUM_AA_TESTERS; i++ ){

                file_name = fr.aa_test_fn[i];

                probe = ::cvCloneImage( fdb.aa_test_set[i] );

                one_component[0] = 0;
                n = 1;

                rc_lambda1 = fc.ClassifyComponentMahalanobis( probe,
one_component, n);

                one_component[0] = 1;
                n = 1;

                rc_lambda2 = fc.ClassifyComponentMahalanobis( probe,
one_component, n );

                one_component[0] = 2;
                n = 1;

                rc_lambda3 = fc.ClassifyComponentMahalanobis( probe,
one_component, n );

                two_component[0] = 0;
                two_component[1] = 1;
                n = 2;

                rc_lambda12 = fc.ClassifyComponentMahalanobis(probe,
two_component, n);

                two_component[0] = 0;
                two_component[1] = 2;
                n = 2;

                rc_lambda13 = fc.ClassifyComponentMahalanobis( probe,
two_component, n );

                two_component[0] = 1;
                two_component[1] = 2;
                n = 2;

                rc_lambda23 = fc.ClassifyComponentMahalanobis( probe,
two_component, n );

                race = "AA";
```

```
            fprintf( outFile ,"\n%s,%s,%s,%s,%s,%s,%s,%s,",
file_name.c_str(),
                                    race, getRaceCode(rc_lambda1),
getRaceCode(rc_lambda2), getRaceCode(rc_lambda3),

      getRaceCode(rc_lambda12), getRaceCode(rc_lambda13),
getRaceCode(rc_lambda23) );

      }

}

      /**
       * The method, runCAComponentClassification, handles all the
methods to which
       * classification is achieved by only examining the 1st, 2nd and
3rd principle
       * components of the Caucasian space.
       */

void FCRS::runCAComponentClassification()
{
      int i, j, n, rc_lambda1, rc_lambda2, rc_lambda3, rc_lambda12,
rc_lambda13, rc_lambda23, index, rank;
      char* race;
      char* c_race;
      int one_component[1];
      int two_component[2];

      FaceClassifier fc( &aa_fs , &ca_fs, &bl_fs, &fdb );

      fprintf( outFile, "\n\t\tRunning Component Classification Test\n"
);

      fprintf( outFile, "Training CA,\t" );

      for( i = 0; i < NUM_CA_TRAINERS; i++ ){
            if( i == NUM_CA_TRAINERS - 1)
                  fprintf( outFile, "%s, \n", fr.ca_train_fn[i].c_str()
);
            else
                  fprintf( outFile, "%s, ", fr.ca_train_fn[i].c_str() );
      }

      fprintf( outFile, "Unknown CA,\t" );

      fprintf( outFile,
"Race,Lambda1,Lambda2,Lambda3,Lambda12,Lambda13,Lambda23");

      std::string file_name;
      ::IplImage* probe = NULL;

      for( i = 0; i < NUM_CA_TESTERS; i++ ){

            file_name = fr.ca_test_fn[i];
```

```
            probe = ::cvCloneImage( fdb.ca_test_set[i] );

            one_component[0] = 0;
            n = 1;

            rc_lambda1 = fc.ClassifyComponentMahalanobis( probe,
one_component, n);

            one_component[0] = 1;
            n = 1;

            rc_lambda2 = fc.ClassifyComponentMahalanobis( probe,
one_component, n );

            one_component[0] = 2;
            n = 1;

            rc_lambda3 = fc.ClassifyComponentMahalanobis( probe,
one_component, n );

            two_component[0] = 0;
            two_component[1] = 1;
            n = 2;

            rc_lambda12 = fc.ClassifyComponentMahalanobis(probe,
two_component, n);

            two_component[0] = 0;
            two_component[1] = 2;
            n = 2;

            rc_lambda13 = fc.ClassifyComponentMahalanobis( probe,
two_component, n );

            two_component[0] = 1;
            two_component[1] = 2;
            n = 2;

            rc_lambda23 = fc.ClassifyComponentMahalanobis( probe,
two_component, n );

            race = "CA";

            fprintf( outFile ,"\n%s,%s,%s,%s,%s,%s,%s,%s,",
file_name.c_str(),
                                    race, getRaceCode(rc_lambda1),
getRaceCode(rc_lambda2), getRaceCode(rc_lambda3),

      getRaceCode(rc_lambda12), getRaceCode(rc_lambda13),
getRaceCode(rc_lambda23) );

      }

}

      /**
```

```
       * The method, runAAClassificationTests, spawns all of the process
to run the
       * Mahalanobis classification tests for the African American
population of
       * faces.
       */

void FCRS::runAAClassificationTests()
{
      int i, rc_mindist, rc_3vote, rc_3weight, rc_5vote, rc_5weight;
      char* race;

      FaceClassifier fc( &aa_fs , &ca_fs, &bl_fs, &fdb );

      fprintf( outFile, "\n\t\tRunning AA All Classification Tests\n" );

      fprintf( outFile, "Training AA,\t" );

      for( i = 0; i < NUM_AA_TRAINERS; i++ ){
            if( i == NUM_AA_TRAINERS - 1)
                  fprintf( outFile, "%s, \n", fr.aa_train_fn[i].c_str()
);
            else
                  fprintf( outFile, "%s, ", fr.aa_train_fn[i].c_str() );
      }

      fprintf( outFile, "Unknown AA,\t" );

      fprintf( outFile, "Race,Full,Full w/ 3 vote,Full w/ 5 vote, Full
w/ 5 weight");

      std::string file_name;
      ::IplImage* probe = NULL;

      for( i = 0; i < NUM_AA_TESTERS; i++ ){

            file_name = fr.aa_test_fn[i];

            probe = ::cvCloneImage( fdb.aa_test_set[i] );

            race = "AA";

            rc_mindist = fc.ClassifyImageMahalanobis( probe );

            rc_3vote = fc.ClassifyImageMahalanobisVote( probe, 3, false
);

            rc_5vote = fc.ClassifyImageMahalanobisVote( probe, 5, false
);

            rc_5weight = fc.ClassifyImageMahalanobisVote( probe, 5, true
);

            fprintf( outFile ,"\n%s,%s,%s,%s,%s,%s,", file_name.c_str(),
                              race, getRaceCode(rc_mindist),
getRaceCode(rc_3vote),
```

```
                                          getRaceCode(rc_5vote),
getRaceCode(rc_5weight) );


      }
}

      /**
       * The method, runCAClassificationTests, spawns all of the process
to run the
       * Mahalanobis classification tests for the Caucasian population
of
       * faces.
       */

void FCRS::runCAClassificationTests()
{
      int i, rc_mindist, rc_3vote, rc_3weight, rc_5vote, rc_5weight;
      char* race;

      FaceClassifier fc( &aa_fs , &ca_fs, &bl_fs, &fdb );

      fprintf( outFile, "\n\t\tRunning CA All Classification Tests\n" );

      fprintf( outFile, "Training CA,\t" );

      for( i = 0; i < NUM_AA_TRAINERS; i++ ){
            if( i == NUM_AA_TRAINERS - 1)
                  fprintf( outFile, "%s, \n", fr.ca_train_fn[i].c_str()
);
            else
                  fprintf( outFile, "%s, ", fr.ca_train_fn[i].c_str() );
      }

      fprintf( outFile, "Unknown CA,\t" );

      fprintf( outFile, "Race,Full,Full w/ 3 vote,Full w/ 5 vote, Full
w/ 5 weight");

      std::string file_name;
      ::IplImage* probe = NULL;

      for( i = 0; i < NUM_CA_TESTERS; i++ ){

            file_name = fr.ca_test_fn[i];

            probe = ::cvCloneImage( fdb.ca_test_set[i] );

            race = "CA";

            rc_mindist = fc.ClassifyImageMahalanobis( probe );

            rc_3vote = fc.ClassifyImageMahalanobisVote( probe, 3, false
);

            rc_5vote = fc.ClassifyImageMahalanobisVote( probe, 5, false
);
```

```
            rc_5weight = fc.ClassifyImageMahalanobisVote( probe, 5, true
);

            fprintf( outFile ,"\n%s,%s,%s,%s,%s,%s,", file_name.c_str(),
                              race, getRaceCode(rc_mindist),
getRaceCode(rc_3vote),
                                          getRaceCode(rc_5vote),
getRaceCode(rc_5weight) );


      }
}

      /**
       * The method, runBlendedIdentification, handles the process which
are called
       * to set up the blended test set, grab probes out of the
FaceDatabase,
       * and rank each probe in the Blended space.
       */

void FCRS::runBlendedIdentification(){
      int i, j, index, rank;

      std::string file_name;
      char* race;
                  fprintf( outFile, "\n\t\tRunning Blended
Identification Test\n" );

      initBlendedTestSetIndices();
      createTestingSet();
      FaceIdentifier fi( &aa_fs, &ca_fs, &bl_fs, &fdb );

      fprintf( outFile, "Target AA(1-15) Target CA(16-30),\t" );
      for( i = 0; i < NUM_BLENDED_KNOWN; i++ )
      {
            if( i % 2 == 1)
                  fprintf( outFile, "%s, ", fr.bl_known_fn[i].c_str());
      }

      fprintf( outFile, "Rank, " );

      ::IplImage* probe = NULL;

      for( i = 0; i < NUM_BLENDED_TESTERS; i++ ){
            file_name = fr.bl_test_fn[i].c_str();

            if( i < NUM_BLENDED_TESTERS/2 )
                  race = "AA";
            else
                  race = "CA";

            probe = ::cvCloneImage( fdb.bl_test_set[i] );

            fi.IdentifyBlended( probe );
```

```
            rank = fi.GetImageRankBL( i );

            fprintf( outFile ,"\n%s,%s,", file_name.c_str(), race );

            for( j = 0; j < NUM_AA_TRAINERS; j++ ){
                        fprintf( outFile, "%3.0f,", fi.unrank_dist_bl[j]
);
            }

            rank++;

            fprintf( outFile, "%i ,", rank );
      }


}

      /**
       * The method, SetupTest, calls methods to which the Training,
Known, and Testing
       * sets are created.  It generates each set for all faces spaces,
African American,
       * Caucasian and blended.
       */


void FCRS::SetupTest()
{
      fr.CreateAATrainingSet();
      fr.CreateAAKnownSet();
      fr.CreateAATestingSet();

      fr.CreateCATrainingSet();
      fr.CreateCAKnownSet();
      fr.CreateCATestingSet();

      fr.CreateBLTrainingSet();
      fr.CreateBLKnownSet();
      fr.CreateBLTestingSet();


      fr.ReadAATrainFile();
      fr.ReadAAKnownFile();
      fr.ReadAATestFile();

      fr.ReadCATrainFile();
      fr.ReadCATestFile();
      fr.ReadCAKnownFile();

      fr.ReadBLTrainFile();
      fr.ReadBLTestFile();
      fr.ReadBLKnownFile();

      fr.CreateAAFaceSet();
      fr.CreateCAFaceSet();
      fr.PrintAAFaceSet();
      fr.PrintCAFaceSet();
```

```
	fdb.LoadAAKnownSet(fr);
	fdb.LoadAATestSet(fr);
	fdb.LoadAATrainingSet(fr);

	fdb.LoadCAKnownSet(fr);
	fdb.LoadCATestSet(fr);
	fdb.LoadCATrainingSet(fr);

	fdb.LoadBLKnownSet(fr);
	fdb.LoadBLTestSet(fr);
	fdb.LoadBLTrainingSet(fr);

	initAAFaceSpace();
	initCAFaceSpace();
	initBlendedFaceSpace();


}

	/**
	 * The method, initFaceReader, simply initializes the class
FaceReader so that it
	 * reads in the all the necessary file paths and initializes the
faces
	 * to create the FaceDatabase.
	 */

void FCRS::initFaceReader(){

	fr.ReadAAFile();
	fr.ReadCAFile();

	fr.CreateAATrainingSet();
	fr.CreateAAKnownSet();
	fr.CreateAATestingSet();

	fr.CreateCATrainingSet();
	fr.CreateCAKnownSet();
	fr.CreateCATestingSet();

	fr.CreateBLTrainingSet();
	fr.CreateBLKnownSet();
	fr.CreateBLTestingSet();


	fr.ReadAATrainFile();
	fr.ReadAAKnownFile();
	fr.ReadAATestFile();

	fr.ReadCATrainFile();
	fr.ReadCATestFile();
	fr.ReadCAKnownFile();

	fr.ReadBLTrainFile();
	fr.ReadBLTestFile();
	fr.ReadBLKnownFile();
```

```
	fr.CreateAAFaceSet();
	fr.CreateCAFaceSet();

	fr.PrintAAFaceSet();
	fr.PrintCAFaceSet();


}

	/**
	 * The method, initDatabase, loads of the African American,
Caucasian and
	 * blended images in the FaceDatabase
	 */


void FCRS::initDatabase(){

	fdb.SetTotalFaces(fr);
	fdb.InsertImages(fr);

	fdb.LoadAAKnownSet(fr);
	fdb.LoadAATestSet(fr);
	fdb.LoadAATrainingSet(fr);

	fdb.LoadCAKnownSet(fr);
	fdb.LoadCATestSet(fr);
	fdb.LoadCATrainingSet(fr);

	fdb.LoadBLKnownSet(fr);
	fdb.LoadBLTestSet(fr);
	fdb.LoadBLTrainingSet(fr);

}

	/**
	 * The method, initAAFaceSpace, initiliazes the African American
Face space
	 * This method calls of the methods pertinent which sets the race,
creates the
	 * training set, and eigen faces.
	 */

void FCRS::initAAFaceSpace(){

	aa_fs.SetRace(1);
	aa_fs.CreateTrainingSet(fdb);
	aa_fs.CreateEigenObjects();
//
	aa_fs.PrintEigenVals();

	aa_fs.EigenDecomposite();

	aa_fs.CalcMeanEigenFace();

	aa_fs.PrintCoefficients();
```

```
	aa_fs.PrintAvgFace();

	aa_fs.NormalizeCoeffs();

	aa_fs.PrintNormCoefficients();

	aa_fs.EigenProject();

	aa_fs.CreateEigenFaces();

	aa_fs.CreateAvgFace();

	aa_fs.FindEuclideanDistanceFromAvg();

	aa_fs.CalcKnownAvgDistance();

	aa_fs.FindKnownAvgMaxDistance();

	aa_fs.FindKnownAvgMinDistance();

	// added code here
	aa_fs.CalcMeanDist();

	aa_fs.PrintMeanDistance();

	aa_fs.CalcStdDev();
}

	/**
	 * The method, initCAFaceSpace, initiliazes the Caucasian Face
space
	 * This method calls of the methods pertinent which sets the race,
creates the
	 * training set, and eigen faces.
	 */

void FCRS::initCAFaceSpace(){


	ca_fs.SetRace(0);
	ca_fs.CreateTrainingSet(fdb);
	ca_fs.CreateEigenObjects();

	ca_fs.PrintEigenVals();

	ca_fs.EigenDecomposite();

	ca_fs.CalcMeanEigenFace();

	ca_fs.PrintCoefficients();

	ca_fs.PrintAvgFace();

	ca_fs.NormalizeCoeffs();

	ca_fs.PrintNormCoefficients();
```

```
	ca_fs.EigenProject();


	ca_fs.CreateEigenFaces();

	ca_fs.CreateAvgFace();

	ca_fs.FindEuclideanDistanceFromAvg();

	ca_fs.CalcKnownAvgDistance();

	ca_fs.FindKnownAvgMaxDistance();

	ca_fs.FindKnownAvgMinDistance();

	// added code here
	ca_fs.CalcMeanDist();

	ca_fs.PrintMeanDistance();

	ca_fs.CalcStdDev();
}

	/**
	 * The method, initBlendedFaceSpace, initiliazes the Blended Face
space
	 * This method calls of the methods pertinent which sets the race,
creates the
	 * training set, and eigen faces.
	 */

void FCRS::initBlendedFaceSpace(){

	bl_fs.SetRace(2);
	bl_fs.CreateTrainingSet(fdb);
	bl_fs.CreateEigenObjects();
//
	bl_fs.PrintEigenVals();
//
	bl_fs.EigenDecomposite();

	bl_fs.CalcMeanEigenFace();

	bl_fs.PrintCoefficients();

	bl_fs.PrintAvgFace();

	bl_fs.NormalizeCoeffs();

	bl_fs.PrintNormCoefficients();

	bl_fs.EigenProject();


	bl_fs.CreateEigenFaces();
```

```
	bl_fs.CreateAvgFace();

	bl_fs.FindEuclideanDistanceFromAvg();

	bl_fs.CalcKnownAvgDistance();

	bl_fs.FindKnownAvgMaxDistance();

	bl_fs.FindKnownAvgMinDistance();

	// added code here
	bl_fs.CalcMeanDist();

	bl_fs.PrintMeanDistance();

	bl_fs.CalcStdDev();
}

	/**
	 * The method, RunDemo, runs the demo which was presented in the
defense speech.
	 */

//int FCRS::RunDemo(){
//
//		int i, rc_mindist, rc_3vote, rc_3weight, rc_5vote,
rc_5weight;
//	char* race;
//
//	FaceClassifier fc( &aa_fs , &ca_fs, &bl_fs, &fdb );
//
//	std::string file_name;
//
//	fr.ReadSpecFile();
//
//	for( i = 0; i < 9; i++ ){
//
//		fprintf( outFile, "%s\n", fr.aa_spec_fp[i].c_str() );
//
//	}
//
//	IplImage* probe = NULL;
//	IplImage* temp = NULL;
//
//	for( i = 0; i < 9; i++ ){
//
//		file_name = fr.aa_spec_fp[i];
//
//		temp = ::cvLoadImage( file_name.c_str() );
//		probe = ::cvCreateImage( cvSize (100,110), IPL_DEPTH_8U, 1
);
//		cvCvtColor( temp, probe, CV_BGR2GRAY );
//		::cvReleaseImage( &temp );
//
//		race = "AA";
//
//		rc_mindist = fc.ClassifyImageMahalanobis( probe );
```

```
//
//          rc_3vote = fc.ClassifyImageMahalanobisVote( probe, 3, false
);
//
//          rc_5vote = fc.ClassifyImageMahalanobisVote( probe, 5, false
);
//
//          rc_5weight = fc.ClassifyImageMahalanobisVote( probe, 5, true
);
//
//          ::cvReleaseImage( &probe );
//
//          std::string name = file_name.substr(56);
//
//          fprintf( outFile ,"\n%s,%s,%s,%s,%s,%s,",  name.c_str(),
//                           race, getRaceCode(rc_mindist),
getRaceCode(rc_3vote),
//                                        getRaceCode(rc_5vote),
getRaceCode(rc_5weight) );
//
//    }
//
//
//
//    return 0;
//
//}

      /**
       * The main method is the entry point into the recognition system.
It starts the program
       * and calls the needed methods to setup the RaceReader,
FaceDatabase, and all of the needed
       * FaceSpaces.
       */


int main( int argc, char** argv ){
      int i;
      FCRS recog_engine;
      recog_engine.initFaceReader();
      recog_engine.initDatabase();
      recog_engine.initAAFaceSpace();
      recog_engine.initCAFaceSpace();
      recog_engine.initBlendedFaceSpace();

/******** Demo *************/

//    recog_engine.RunDemo();


//
//    FaceClassifier fc( &aa_fs , &ca_fs, &bl_fs, &fdb );
////
////  fc.initAATestSet();
////  fc.initCATestSet();
//
```

```
////  fc.PrintTestSets();
//    fc.ClassifyTestSet();
//    fc.ClassifyTestSetInRange();
//    fc.PrintTestCoeffs();
//
///   recog_engine.showEigenFaces();
//    recog_engine.showAvgAAFace();
//    recog_engine.showAvgCAFace();
//    recog_engine.showAvgBlendedFace();
//

//    for( i = 0; i < 1; i++ ){
//          fprintf( recog_engine.outFile, "\n\n\t\t*********
IDENTIFICATION TEST %i ******************\n\n", i );
//          recog_engine.SetupTest();
//          recog_engine.runAAIdentificationSet();
//          recog_engine.runCAIdentificationSet();
//
//    }
//
//    for( i = 0; i < 1; i++ ){
//          fprintf( recog_engine.outFile, "\n\n\t\t*********
IDENTIFICATION TEST %i ******************\n\n", i );
//          recog_engine.SetupTest();
//          recog_engine.runAAClassificationIdentificationSet();
//          recog_engine.runCAClassificationIdentificationSet();
//
//    }

//    recog_engine.runAAClassificationIdentificationSet();
//    recog_engine.runCAClassificationIdentificationSet();

//    recog_engine.runAAComponentClassification();
//    recog_engine.runAAClassificationTests();
//
//    recog_engine.runCAComponentClassification();
//    recog_engine.runCAClassificationTests();

//    recog_engine.runIdentificationSet();

//    recog_engine.runBlendedIdentification();
//
//    FaceIdentifier fi( &aa_fs, &ca_fs, &bl_fs, &fdb );
//    fi.ProjectKnownImages();
//    fi.PrintKnownCoeffs();


//    int return_class;
//    int j;
//    ::IplImage* unknown = NULL;
//    ::IplImage* cloned = NULL;
//    ::IplImage* face = NULL;
//
//    for( j = 1; j < NUM_AA_TRAINERS; j+=2 ){
//          unknown = ::cvCloneImage( fdb.all_faces[j] );
//          //recog_engine.showSimpleFace(unknown);
//          cloned = ::cvCloneImage( fdb.all_faces[j] );
```

```
//          return_class = fi.Identify( cloned, &fc );
//          if( return_class == AA ){
////              for( i = 0; i < 10; i++ ){
////                   face = ::cvCloneImage(
aa_fs.aa_trainers[fi.rank_aa[i]] );
////                   recog_engine.showSimpleFace( face  , "Ranked
Image" );
////              }
//                fprintf( recog_engine.outFile , "Face %i classified as
African American\n", j );
//          }else if( return_class == CA ){
////              for( i = 0; i < 10; i++ ){
////                   face = ::cvCloneImage(
ca_fs.ca_trainers[fi.rank_ca[i]] );
////                   recog_engine.showSimpleFace(  face , "Ranked
Image" );
////              }
//                fprintf( recog_engine.outFile , "Face %i classified as
Caucasian American\n", j );
//          }else if( return_class == BOTH ){
//                fprintf( recog_engine.outFile,  "Face classified as
both.\n" );;
//          }else if( return_class == UNKNOWN )
//                fprintf( recog_engine.outFile,  "Face unclassified.\n"
);
////
//    }
      return 0;
}

/*********************************************************************
 * System: Face Classification Recognition System.
 *
 * Software for Thesis:
 *          NEUROCOGNITIVE INSPIRED HIERARCHICAL FACE RECOGNITION SYSTEM
 *
 * File name: FaceSpace.h
 *
 * Input file name(s):  highgui.h
 *                                  cv.h
 *                                  cvaux.h
 *                                  cmath.h
 *                                  cstdlib.h
 *                                  stdio.h
 *                                  fstream.h
 *
 * Output file name(s):
 *
 * Author: Ryan Wilkins
 * Date Last Modified: 07/01/07
 * Purpose:
 *
 *********************************************************************/
#ifndef FACESPACE_H_
#define FACESPACE_H_
```

```
#define CA 0
#define     AA 1
#define BLENDED 2

#include <cvaux.h>
#include <highgui.h>
#include <cv.h>
#include <iostream>
#include <stdio.h>
#include <cmath>
#include <cstdlib>
#include <fstream>
#include <FaceDatabase.h>

#define NUM_AA_TRAINERS 30
#define NUM_AA_TESTERS 30
#define NUM_CA_TRAINERS 30
#define NUM_CA_TESTERS 30
#define NUM_BLENDED_TESTERS 30
#define NUM_BLENDED_TRAINERS 30

#define NUM_EIGENS 16

class FaceSpace
{
public:
      void SetNumCATrainers();
      void SetNumCATesters();
      void SetNumAATrainers();
      void SetNumAATesters();
      void SetNumBlendedTrainers();
      void SetNumBlendedTesters();

      bool InSet( int c, int components[], int n_comp );
      float FindComponentMahalanobisDistance( ::IplImage* probe,
::IplImage* known, int components[], int n_comp );
      float FindcvMahalanobisDistance( ::IplImage* probe, ::IplImage
*known );
      float FindMahalanobisDistance( ::IplImage* probe, ::IplImage
*known );
      float FindDistance( ::IplImage* probe, ::IplImage* known );
      void CalcStdDev();
      void CalcMeanDist();
      void CalcMeanEigenFace();
      void GetImageCoeffs(::IplImage* probe, float coeffs[NUM_EIGENS]);
      double* GetUnknownImageCoeffs(::IplImage* probe);
      double CalcUnknownAvgDistance(IplImage* probe);
      void FindKnownAvgMaxDistance();
      void FindKnownAvgMinDistance();
      void CalcKnownAvgDistance();
      void FindEuclideanDistanceFromAvg();
      void CreateAvgFace();
      float FindMax(::IplImage* img);
      float FindMin(::IplImage* img);
      void CreateEigenFaces();
      void PrintNormCoefficients();
      int NormalizeCoeffs();
```

```
        int EigenProject();
        int EigenDecomposite();
        int CreateEigenObjects();
        void PrintEigenVals();
        void PrintAvgFace();
        void PrintCoefficients();
        void PrintMeanDistance();
        void SetRace(int r);
        void CreateTrainingSet(FaceDatabase db);
        IplImage* GetAverageFace();
        void InitAATrainIndex();
        void InitCATrainIndex();


        FaceSpace();
        virtual ~FaceSpace();

        int race;
        FILE* outFile;

        ::IplImage* aa_trainers[NUM_AA_TRAINERS];
        ::IplImage* aa_test[NUM_AA_TESTERS];
        ::IplImage* ca_trainers[NUM_CA_TRAINERS];
        ::IplImage* ca_test[NUM_CA_TESTERS];
        ::IplImage* blended_test[NUM_BLENDED_TESTERS];
        ::IplImage* blended_trainers[NUM_BLENDED_TRAINERS];

        int nEigens;
        IplImage* aa_proj[NUM_AA_TRAINERS];
        IplImage* ca_proj[NUM_CA_TRAINERS];
        IplImage* blended_proj[NUM_CA_TRAINERS];
        IplImage* eigens[NUM_EIGENS];
//      IplImage* eigens[NUM_TRAINERS-1];
        IplImage* eigen_faces[NUM_EIGENS];
        IplImage* aa_avg;
        IplImage* ca_avg;
        IplImage* blended_avg;

        IplImage* aa_avg_face;
        IplImage* ca_avg_face;
        IplImage* blended_avg_face;


        CvTermCriteria criteria;
        float vals[NUM_EIGENS];
        float aa_coeffs[NUM_AA_TRAINERS][NUM_EIGENS];
        float ca_coeffs[NUM_CA_TRAINERS][NUM_EIGENS];
        float blended_coeffs[NUM_BLENDED_TRAINERS][NUM_EIGENS];


        float aa_norm_coeffs[NUM_AA_TRAINERS][NUM_EIGENS];
        float ca_norm_coeffs[NUM_CA_TRAINERS][NUM_EIGENS];
        float blended_norm_coeffs[NUM_BLENDED_TRAINERS][NUM_EIGENS];

        float aa_mean_eigen[NUM_EIGENS];
        float ca_mean_eigen[NUM_EIGENS];
        float blended_mean_eigen[NUM_EIGENS];
```

```
	//float known_unknown_distance_matrix[NUM_TRAINERS][#unknown];

	int i_max;
	int i_min;

	double avg_distance;
	double max_distance;
	double min_distance;

	double aa_known_avg_dist_matrix[NUM_AA_TRAINERS];
	double ca_known_avg_dist_matrix[NUM_CA_TRAINERS];
	double blended_known_avg_dist_matrix[NUM_BLENDED_TRAINERS];

	float aa_mean_dist[NUM_AA_TRAINERS];
	float ca_mean_dist[NUM_CA_TRAINERS];
	float blended_mean_dist[NUM_BLENDED_TRAINERS];

	float mean_dist;
	float max_mean_dist;
	float min_mean_dist;
	float std_dev;

	int aa_train_index[NUM_AA_TRAINERS];
	int ca_train_index[NUM_CA_TRAINERS];
	int blended_train_index[NUM_BLENDED_TRAINERS];
};

#endif /*FACESPACE_H_*/

/*********************************************************************
 * System: Face Classification Recognition System.
 *
 * Software for Thesis:
 *          NEUROCOGNITIVE INSPIRED HIERARCHICAL FACE RECOGNITION SYSTEM
 *
 * File name: FaceSpace.cpp
 *
 * Input file name(s):  FaceSpace.h
 *
 * Output file name(s): aa_space
 *                                  ca_space
 *                                  bl_space
 *
 * Author: Ryan Wilkins
 * Date Last Modified: 07/01/07
 * Purpose: The class that represents race specific Eigen space(s)
 *
 *********************************************************************/

#include "FaceSpace.h"

/**
 *  Race value of 1 is AA, and 0 is CA
 *  Even indexed images in array all_faces make up the training set
 *  for the face space.
 * /home/rbw9908/workspace
```

```
 */

using namespace std;

	/**
	 * The method, FindcvMahalanobisDistance, uses the OpenCV function
	 * to calculate the Mahalanobis distance given a probe and a
	 * known image.
	 *
	 * @param IplImage* probe -- unknown face
	 * @param IplImage* known -- known face
	 * @return float -- the Mahalanobis distance between the two
images.
	 */

float FaceSpace::FindcvMahalanobisDistance( ::IplImage* probe,
::IplImage* known )
{
	CvMat* cov = ::cvCreateMat( 11000, 11000,  CV_32F );
	CvMat* mat = ::cvCreateMat( 11000, 11000, CV_32F );
	const CvArr *arr[NUM_AA_TRAINERS];

	float mahalanobis;
	int i;

	switch( race ){

		case CA:

		break;

		case AA:
			for ( i = 0; i < NUM_AA_TRAINERS; i++ )
				arr[i] = ::cvCloneImage( aa_trainers[i] );

				::cvCalcCovarMatrix( (const CvArr**)aa_trainers,
NUM_AA_TRAINERS, cov, aa_avg, CV_COVAR_USE_AVG );
		break;

		case BLENDED:

		break;

	}

	::cvInvert( cov, mat, 1 );

	mahalanobis = ::cvMahalanobis( probe, known, mat);

	fprintf( outFile, "\nMahalanobis Distance: %f\n", mahalanobis );

	return mahalanobis;


}

	/**
```

```
	 * The method, InSet, finds whether or not a component, c, is in
	 * a set of components.
	 *
	 * @param int c -- component in question
	 * @param int[] components -- the set of components
	 * @param int n_comp -- the number of components in the set
	 * @return bool -- the boolean, true or false that the component
is in the set
	 */

bool FaceSpace::InSet( int c, int components[], int n_comp )
{
	int i;
	bool inSet = false;

	for( i = 0; i < n_comp; i++ )
	{
		if( components[i] == c )
			inSet = true;
	}

	return inSet;
}

	/**
	 * The method, FindComponentMahalanobisDistance, calculates the
Mahalanobis distance
	 * using a small set of coefficients or dimensions instead of all
of the coefficients.
	 * This method was used in an attempt to classify faces based only
on the 1st, 2nd,
	 * 3rd coefficients of a face space since those have been
statistically found as
	 * the main components of a FaceSpace.
	 *
	 * @param IplImage* probe -- unknown image
	 * @param IplImage* known -- known image
	 * @param int[] components -- the components not to use in the
evaluation
	 * @param int n_comp -- the number of components not to use]
	 */

float FaceSpace::FindComponentMahalanobisDistance( ::IplImage* probe,
::IplImage* known, int components[], int n_comp )
{
	int i;
	float pCoeffs[NUM_EIGENS];
	float kCoeffs[NUM_EIGENS];
	float mahalanobis;

	for( i = 0; i < NUM_EIGENS; i++ )
	{
		switch( race ){
		case AA:
			pCoeffs[i] = ::cvCalcDecompCoeff( probe, eigens[i],
aa_avg );
```

```
                  kCoeffs[i] = ::cvCalcDecompCoeff( known, eigens[i],
aa_avg );
            break;
            case CA:
                  pCoeffs[i] = ::cvCalcDecompCoeff( probe, eigens[i],
ca_avg );
                  kCoeffs[i] = ::cvCalcDecompCoeff( known, eigens[i],
ca_avg );
            break;
            case BLENDED:
                  pCoeffs[i] = ::cvCalcDecompCoeff( probe, eigens[i],
blended_avg );
                  kCoeffs[i] = ::cvCalcDecompCoeff( known, eigens[i],
blended_avg );
            break;
            }
      }

      for( i = 0; i < NUM_EIGENS; i++ ){
            if( !InSet( i, components, n_comp ) )
                  mahalanobis +=
(1.0/::sqrt(vals[i]))*(pCoeffs[i]*kCoeffs[i]);
      }

//    ::cvReleaseImage( &known );

      return -mahalanobis;
}

      /**
       * The method, FindMahalanobisDistance, calculates the Mahalanobis
distance
       * of an unknown face with a known face.
       *
       * @param IplImage* probe -- unknown face
       * @param IplImage* known -- known face
       * @return float -- Mahalonibis distance between the two faces
       */

float FaceSpace::FindMahalanobisDistance( ::IplImage* probe, ::IplImage
*known )
{

      int i;
      float pCoeffs[NUM_EIGENS];
      float kCoeffs[NUM_EIGENS];
      float mahalanobis;

      for( i = 0; i < NUM_EIGENS; i++ )
      {
            switch( race ){
            case AA:
                  pCoeffs[i] = ::cvCalcDecompCoeff( probe, eigens[i],
aa_avg );
                  kCoeffs[i] = ::cvCalcDecompCoeff( known, eigens[i],
aa_avg );
            break;
```

```
            case CA:
                  pCoeffs[i] = ::cvCalcDecompCoeff( probe, eigens[i],
ca_avg );
                  kCoeffs[i] = ::cvCalcDecompCoeff( known, eigens[i],
ca_avg );
            break;
            case BLENDED:
                  pCoeffs[i] = ::cvCalcDecompCoeff( probe, eigens[i],
blended_avg );
                  kCoeffs[i] = ::cvCalcDecompCoeff( known, eigens[i],
blended_avg );
            break;
            }

      }

      for( i = 0; i < NUM_EIGENS; i++ ){
            mahalanobis +=
(1.0/::sqrt(vals[i]))*(pCoeffs[i]*kCoeffs[i]);
      }

//    ::cvReleaseImage( &known );

      return -mahalanobis;

}

      /**
       * The method, FindDistance, calculates the Euclidean distance of
an unknown
       * face from a known face in eigen space.
       *
       * @param IplImage* probe -- the unknown image
       * @param IplImage* known -- the known image
       * @return float -- the Euclidean distance between the two input
images
       */

float FaceSpace::FindDistance( ::IplImage* probe, ::IplImage* known ){

      float pCoeffs[NUM_EIGENS];
      float kCoeffs[NUM_EIGENS];
      int i;
      float dx, sumdx, euclidean;

      fprintf( outFile , "\nFinding distance between probe and known
image\n" );

      for( i = 0; i < NUM_EIGENS; i++ )
      {
            switch( race ){
            case AA:
                  pCoeffs[i] = ::cvCalcDecompCoeff( probe, eigens[i],
aa_avg );
                  kCoeffs[i] = ::cvCalcDecompCoeff( known, eigens[i],
aa_avg );
            break;
```

```
            case CA:
                  pCoeffs[i] = ::cvCalcDecompCoeff( probe, eigens[i],
ca_avg );
                  kCoeffs[i] = ::cvCalcDecompCoeff( known, eigens[i],
ca_avg );
            break;
            case BLENDED:
                  pCoeffs[i] = ::cvCalcDecompCoeff( probe, eigens[i],
blended_avg );
                  kCoeffs[i] = ::cvCalcDecompCoeff( known, eigens[i],
blended_avg );
            break;
            }

      }



      for( i = 0; i < NUM_EIGENS; i++ )
      {
            fprintf( outFile , "pCoeffs[%i]=%3.2f ", i, pCoeffs[i] );
      }
      fprintf( outFile , "\n" );
      for( i = 0; i < NUM_EIGENS; i++ )
      {
            fprintf( outFile , "kCoeffs[%i]=%3.2f ", i, kCoeffs[i] );
      }


      for( i = 0; i < NUM_EIGENS; i++ ){

            dx = pCoeffs[i] - kCoeffs[i];
            sumdx += ::pow( dx , 2 );

      }

      euclidean = ::sqrt( sumdx );


      return euclidean;
}

      /**
       * The method, CalcStdDev, calculates the standard deviation of
       * a face space.  Using an Euclidean distance metric, the standard
       * deviation was used in classification attempts by setting a
       * threshold that an unknown face had to be within to get
       * classified.
       */

void FaceSpace::CalcStdDev(){

      int i;
      float var;

      fprintf( outFile, "\n Std Deviation:\n ");
```

```
	switch(race){

		case AA:
			for(i = 0; i < NUM_AA_TRAINERS; i++){
				var += ::pow(aa_mean_dist[i] - avg_distance, 2);

			}

			std_dev = ::sqrt(var/(float)NUM_AA_TRAINERS);

		break;
		case CA:
			for(i = 0; i < NUM_CA_TRAINERS; i++){
				var   += ::pow(ca_mean_dist[i] - avg_distance,
2);
			}
			std_dev = ::sqrt(var/(float)NUM_CA_TRAINERS);
		break;
		case BLENDED:
			for(i = 0; i < NUM_BLENDED_TRAINERS; i++){
				var   += ::pow(blended_mean_dist[i] -
avg_distance, 2);
			}
			std_dev = ::sqrt(var/(float)NUM_BLENDED_TRAINERS);

		break;

	}

	fprintf( outFile, "%3.2f\n", std_dev);


}

	/**
	 * The method, CalcMeanDist, calculates the distance each
	 * of the eigens from the average eigen face.  This was ussed in
Euclidean
	 * classification attempts to create a bubble or distance
threshold around
	 * a face space.
	 */

void FaceSpace::CalcMeanDist(){

	int i, j, n;

	if(race == AA){
		n = NUM_AA_TRAINERS;
	}else if( race == CA ){
		n = NUM_CA_TRAINERS;
	}else
		n = NUM_BLENDED_TRAINERS;

	for(i = 0; i < n; i++){
		float d1 = 0.0;
		for(j = 0; j < nEigens; j++){
```

```
                  switch(race){
                        case AA:
                              d1 += ::pow(aa_coeffs[i][j] -
aa_mean_eigen[j], 2);
                              break;
                        case CA:
                              d1 += ::pow(ca_coeffs[i][j] -
ca_mean_eigen[j], 2);
                              break;
                         case BLENDED:
                              d1 += ::pow(blended_coeffs[i][j] -
blended_mean_eigen[j], 2);
                              break;
                  }
            }

            switch(race){
                  case AA:
                        aa_mean_dist[i] = ::sqrt(d1);
                        break;
                  case CA:
                        ca_mean_dist[i] = ::sqrt(d1);
                        break;
                  case BLENDED:
                        blended_mean_dist[i] = ::sqrt(d1);
                  break;
            }
      }

}

      /**
       * The method, CalcMeanEigenFace, calculates the average
coefficients
       * of the set of eigen coefficients.  This average eigen
coefficients
       * were used in early attempts of classification by creating the
centroid
       * of the eigen vectors.
       */

void FaceSpace::CalcMeanEigenFace(){
      int i,j,n;

      if(race == AA){
            n = NUM_AA_TRAINERS;
      }else if( race == CA){
            n = NUM_CA_TRAINERS;
      }else
            n = NUM_BLENDED_TRAINERS;

      fprintf(outFile, "\n*** Eigen Coefficients ***\n");;

      for(j = 0; j < nEigens; j++){

            float k = 0;
            for(i = 0; i < n; i++){
```

```
                  switch( race ){
                  case AA:
                        k += aa_coeffs[i][j];
                  break;
                  case CA:
                        k += ca_coeffs[i][j];
                  break;
                  case BLENDED:
                        k += blended_coeffs[i][j];
                  break;
                  }
            }
            switch( race ){
            case AA:
                  aa_mean_eigen[j] = k/(float)n;
            break;
            case CA:
                  ca_mean_eigen[j] = k/(float)n;
            break;
            case BLENDED:
                  blended_mean_eigen[j] = k/(float)n;
            break;
            }
      }


}

      /**
       * The method, GetImageCoeffs, allocates image coefficients using
the
       * float[] passed as a parameter.  The probe is an unknown face
and the
       * coefficients are its location in face space.
       *
       * @param IplImage* probe -- unknown image
       * @param float[] coeffs -- the array in which the coefficients
will be set
       */

void FaceSpace::GetImageCoeffs(::IplImage* probe, float
coeffs[NUM_EIGENS]){
      int i;


      fprintf( outFile, "\nInitializing Image Coeffs\n" );

      for(i = 0; i < nEigens; i++){
            switch( race ){
            case AA:
                  coeffs[i] = ::cvCalcDecompCoeff(probe, eigens[i],
aa_avg);
                  fprintf( outFile , "%3.2f, ", coeffs[i] );
            break;
            case CA:
                  coeffs[i] = ::cvCalcDecompCoeff(probe, eigens[i],
ca_avg);
```

```
                  fprintf( outFile , "%3.2f, ", coeffs[i] );
            break;
            case BLENDED:
                  coeffs[i] = ::cvCalcDecompCoeff(probe, eigens[i],
blended_avg);
                  fprintf( outFile , "%3.2f, ", coeffs[i] );
            break;
            }
      }

}

      /**
       * The method, GetUnknowImageCoeffs, finds the coefficients or
location of a
       * given probe in a corresponding FaceSpace depending on race.
Using the
       * input probe, the set of eigens and the average face the
coefficients
       * are calculated.
       *
       * @param IplImage* probe -- unknown probe image
       * @return double* --
       */

double* FaceSpace::GetUnknownImageCoeffs(::IplImage* probe){
      double aa_coeff_in[nEigens];
      double ca_coeff_in[nEigens];
      double blended_coeff_in[nEigens];
      int i;

      for(i = 0; i < nEigens; i++){
            switch( race ){
            case AA:
                  aa_coeff_in[i] = ::cvCalcDecompCoeff(probe, eigens[i],
aa_avg);
                  fprintf( outFile , "%3.2f,", aa_coeff_in[i] );
            break;
            case CA:
                  ca_coeff_in[i] = ::cvCalcDecompCoeff(probe, eigens[i],
ca_avg);
                  fprintf( outFile , "%3.2f,", ca_coeff_in[i] );
            break;
            case BLENDED:
                  blended_coeff_in[i] = ::cvCalcDecompCoeff(probe,
eigens[i], blended_avg);
                  fprintf( outFile , "%3.2f,", blended_coeff_in[i] );
            break;
            }
      }

      return NULL;

}

      /**
```

```
       * The method, CalcUnknownAvgDistance, calculates the Euclidean
distance
       * of a probe face when projected in a FaceSpace.  This distance
is
       * calculated and then later used in classification.
       *
       * @param IplImage* probe -- input image in which an average
distance is calculated
       * @return double -- the distance that has been calculated from
the probe, eigens,
       *                               average face
       */

double FaceSpace::CalcUnknownAvgDistance(IplImage* probe){
      double aa_coeff_in[nEigens];
      double ca_coeff_in[nEigens];
      double blended_coeff_in[nEigens];
      int i;
      double d_avg;
//    fprintf( outFile , "\n**** Calculating Unknown Image Distance ****
\n ");

      for(i = 0; i < nEigens; i++){
            switch( race ){
            case AA:
                  aa_coeff_in[i] = ::cvCalcDecompCoeff(probe, eigens[i],
aa_avg);
            break;
            case CA:
                  ca_coeff_in[i] = ::cvCalcDecompCoeff(probe, eigens[i],
ca_avg);
            break;
            case BLENDED:
                  blended_coeff_in[i] = ::cvCalcDecompCoeff(probe,
eigens[i], blended_avg);
            break;
            }
      }

      switch( race ){
      case AA:
            for(i = 0; i < nEigens; i++)
                  d_avg += ::pow(aa_coeff_in[i], 2);
      break;
      case CA:
            for(i = 0; i < nEigens; i++)
                  d_avg += ::pow(ca_coeff_in[i], 2);
      break;
      case BLENDED:
            for(i = 0; i < nEigens; i++)
                  d_avg += ::pow(blended_coeff_in[i], 2);
      break;
      }

      d_avg = ::sqrt(d_avg);

      return d_avg;
```

```
}

	/**
	 * The method, FindKnowAvgMaxDistance, finds the maximum average
distance
	 * in a matrix of average distances. The maximum distance has been
used
	 * in early attempts of classification as a high threshold cutoff.
	 */

void FaceSpace::FindKnownAvgMaxDistance(){

	int i, n;
	double max = -1000000.00;
	if(race == AA){
		n = NUM_AA_TRAINERS;
	}else if( race == CA){
		n = NUM_CA_TRAINERS;
	}else
		n = NUM_BLENDED_TRAINERS;

	fprintf( outFile , "\n*** Finding KA max distance ***");

	for(i = 0; i < n; i++){
		switch( race ){
		case AA:
			if(max < aa_known_avg_dist_matrix[i]){
				max = aa_known_avg_dist_matrix[i];
				i_max = i;
			}
		break;
		case CA:
			if(max < ca_known_avg_dist_matrix[i]){
				max = ca_known_avg_dist_matrix[i];
				i_max = i;
			}
		break;
		case BLENDED:
			if(max < blended_known_avg_dist_matrix[i]){
				max = blended_known_avg_dist_matrix[i];
				i_max = i;
			}
		break;
		}
	}

	max_distance = max;

	fprintf( outFile, "\n(%i) max:\t%3.2f", i_max, max_distance );

}

	/**
	 * The method, FindKnownAvgMinDistance, find the minimum distance
	 * of a matrix of average distances.  The minimum distance was
used
	 * as a low threshold in early classification attempts.
```

```
       */

void FaceSpace::FindKnownAvgMinDistance(){

      int i, n;
      double min = 1000000.00;
      if(race == AA){
            n = NUM_AA_TRAINERS;
      }else if( race == CA ){
            n = NUM_CA_TRAINERS;
      }else
            n = NUM_BLENDED_TRAINERS;

      fprintf( outFile , "\n*** Finding KA min distance ***");

      for(i = 0; i < n; i++){
            switch( race ){

            case AA:
                  if(min > aa_known_avg_dist_matrix[i]){
                        min = aa_known_avg_dist_matrix[i];
                        i_min = i;
                  }
            break;
            case CA:
                  if(min > ca_known_avg_dist_matrix[i]){
                        min = ca_known_avg_dist_matrix[i];
                        i_min = i;
                  }
            break;
            case BLENDED:
                  if(min > blended_known_avg_dist_matrix[i]){
                        min = blended_known_avg_dist_matrix[i];
                        i_min = i;
                  }
            break;

            }

      }

      min_distance = min;

      fprintf( outFile, "\n(%i) max:\t%3.2f", i_min, min_distance );

}

      /**
       * The method, CalcKnownAvgDistance, finds the average distance of
known
       * faces in a FaceSpace depending its race.  The average distance
of these
       * know images has been used in early attempts to classify a face.
       */

void FaceSpace::CalcKnownAvgDistance(){
      int i, n;
```

```
	double sum = 0.0;

	fprintf( outFile, "\n*** Calculating Avg Distance ****");

	switch( race ){
	case AA:
		for(i = 0; i < NUM_AA_TRAINERS; i++)
			sum += aa_known_avg_dist_matrix[i];
	break;
	case CA:
		for(i = 0; i < NUM_CA_TRAINERS; i++)
			sum += ca_known_avg_dist_matrix[i];
	break;
	case BLENDED:
		for(i = 0; i < NUM_CA_TRAINERS; i++)
			sum += blended_known_avg_dist_matrix[i];
	break;
	}


	avg_distance = sum/n;

	fprintf( outFile, "\naverage distance:\t%3.2f", avg_distance);

}

	/**
	 * The method, FindEuclideanDistanceFromAvg, finds the average
distance of training images
	 * of a FaceSpace depending on the race.  The average distance is
later used in early
	 * attempts to classify a face to a race.
	 */

void FaceSpace::FindEuclideanDistanceFromAvg(){

	int i, c, n;
	double avg_coeff[nEigens];
	double davg;

	if(race == AA){
		n = NUM_AA_TRAINERS;
	}else if (race == CA){
		n = NUM_CA_TRAINERS;
	}else
		n = NUM_BLENDED_TRAINERS;

	fprintf( outFile , "\n*** Calculating Euclidean Distance ***");

	for(i = 0; i < n; i++){
		for(c = 0; c < nEigens; c++){
			switch( race ){
			case AA:
				avg_coeff[c] = ::cvCalcDecompCoeff(
aa_trainers[i], eigens[c] ,  aa_avg);
			break;
			case CA:
```

```
                        avg_coeff[c] = ::cvCalcDecompCoeff(
ca_trainers[i], eigens[c] ,  ca_avg);
                  break;
                  case BLENDED:
                        avg_coeff[c] = ::cvCalcDecompCoeff(
blended_trainers[i], eigens[c] ,  blended_avg);
                  break;
                  }
            }
            davg = 0;

            for(c = 0; c < nEigens; c++)  {
                  davg += ::pow(avg_coeff[c], 2);
            }

            switch( race ){
            case AA:
                  aa_known_avg_dist_matrix[i] = ::sqrt(davg);
                  fprintf(outFile, "\n(%i)\t%3.2f", i,
aa_known_avg_dist_matrix[i]);
            break;
            case CA:
                  ca_known_avg_dist_matrix[i] = ::sqrt(davg);
                  fprintf(outFile, "\n(%i)\t%3.2f", i,
ca_known_avg_dist_matrix[i]);
            break;
            case BLENDED:
                  blended_known_avg_dist_matrix[i] = ::sqrt(davg);
                  fprintf(outFile, "\n(%i)\t%3.2f", i,
blended_known_avg_dist_matrix[i]);
            break;
            }

      }

}

      /**
       * The method, CreateAvgFace, creates the visible representation
of the
       * average face corresponding to a race. It normalizes the
floating point
       * pixels so that it can viewed using the OpenCV window.
       *
       */

void FaceSpace::CreateAvgFace(){

      int i, j;
      int height, width, step, channels;
      float max, min, abs_min;
      ::u_char* aa_out_data;
      ::u_char* ca_out_data;
      ::u_char* blended_out_data;
      CvScalar s;
      int p;
      float intensity;
```

```
      ca_avg_face = ::cvCreateImage( ::cvSize( 100, 110 ), IPL_DEPTH_8U,
1  );

      switch( race ){

      case AA:
            aa_avg_face = ::cvCreateImage( ::cvSize( 100, 110 ),
IPL_DEPTH_8U, 1 );
            height = aa_avg->height;
            width = aa_avg->width;
            step = aa_avg_face->widthStep;
            channels = aa_avg->nChannels;
            aa_out_data = (::u_char *)aa_avg_face->imageData;

            max = FindMax( aa_avg );
            min = FindMin( aa_avg );
            abs_min = ::abs(min);

            max = max + abs_min;

            fprintf( outFile , "\n**** Creating AA Average Face Max(%f)
Min(%f)********\n", max, min);
            for( i = 0; i < height; i++ ){
                  for( j = 0; j < width; j++ ){
                        s = ::cvGet2D( aa_avg , i , j );
                        intensity = s.val[0];
                        intensity = intensity + abs_min;

                        p = 255 * (intensity / max );

            //          fprintf( outFile, "%f ", intensity );
                        aa_out_data[i*step+j] = p;

                  }
            //    fprintf( outFile, "\n");
            }
      break;
      case CA:
//          fprintf( outFile, "\n*** Creating Average Caucasian Face
****\n" );
//          ca_avg_face = ::cvCreateImage( ::cvSize( 100, 110 ),
IPL_DEPTH_8U, 1  );
            height = ca_avg->height;
            width = ca_avg->width;
            step = ca_avg_face->widthStep;
            channels = ca_avg->nChannels;
            ca_out_data = (::u_char *)ca_avg_face->imageData;
//
            max = FindMax( ca_avg );
            min = FindMin( ca_avg );
            abs_min = ::abs(min);
//
            max = max + abs_min;
//
            fprintf( outFile , "\n**** Creating CA Average Face Max(%f)
Min(%f)********\n", max, min);
            for( i = 0; i < height; i++ ){
```

```
                  for( j = 0; j < width; j++ ){
                        s = ::cvGet2D( ca_avg , i , j );
                        intensity = s.val[0];
                        intensity = intensity + abs_min;

                        p = 255 * (intensity / max );


                        fprintf( outFile, "%f ", intensity );
                        ca_out_data[i*step+j] = p;

                  }
                  fprintf( outFile, "\n");

            }
      break;

      case BLENDED:
            blended_avg_face = ::cvCreateImage( ::cvSize( 100, 110 ),
IPL_DEPTH_8U, 1 );
            height = blended_avg->height;
            width = blended_avg->width;
            step = blended_avg_face->widthStep;
            channels = blended_avg->nChannels;
            blended_out_data = (::u_char *)blended_avg_face->imageData;

            max = FindMax( blended_avg );
            min = FindMin( blended_avg );
            abs_min = ::abs(min);

            max = max + abs_min;

            fprintf( outFile , "\n**** Creating Blended Average Face
Max(%f) Min(%f)********\n", max, min);
            for( i = 0; i < height; i++ ){
                  for( j = 0; j < width; j++ ){
                        s = ::cvGet2D( blended_avg , i , j );
                        intensity = s.val[0];
                        intensity = intensity + abs_min;

                        p = 255 * (intensity / max );

            //          fprintf( outFile, "%f ", intensity );
                        blended_out_data[i*step+j] = p;

                  }
            //    fprintf( outFile, "\n");
            }
      break;

      }



}

      /**
```

```
       * The method, FindMin, finds the minimum pixel intensity of an
input face.
       * The input face in this instance will be the non-normalized
eigen face.
       *
       * @param IplImage* img -- input image
       * @return float -- floating point representation of the min pixel
instensity
       */

float FaceSpace::FindMin(::IplImage* img){
      float min, intensity;
      int i, j;
      int height, width, step, channels;
      CvScalar s;

      height = img->height;
      width = img->width;
      step = img->widthStep;
      channels = img->nChannels;
      min = 1000.0;

      for( i = 0; i < height; i++ ){
            for( j = 0; j < width; j++){
                  s = ::cvGet2D( img, i, j);
                  intensity = s.val[0];
                  if(min > intensity){
                       min = intensity;
                  }
            }
      }

      return min;

}

      /**
       * The method, FindMax, finds the maximum pixel intensity for an
EigenFace.
       * This method is used in creating the visible eigen faces and
helps to normalize
       * the input face.
       *
       * @param IplImage* img -- the image in which a max pixel will be
found
       * @return float -- the floating point number of that max pixel
       */

float FaceSpace::FindMax(::IplImage* img){
      float max, intensity;
      int i, j;
      int height, width, step, channels;
      CvScalar s;

      height = img->height;
      width = img->width;
      step = img->widthStep;
```

```
	channels = img->nChannels;
	max = -1000.0;

	for( i = 0; i < height; i++ ){
		for( j = 0; j < width; j++){
			s = ::cvGet2D( img, i, j);
			intensity = s.val[0];
			if(max < intensity){
				max = intensity;
			}
		}
	}

	return max;

}

	/**
	 * The method, CreateEigenFaces, creates visible representation of
the eigen
	 * faces that allows the recognition to display the images to
learn about
	 * which face features/regions are being exemplified in the PCA
process.  It evaluates
	 * the IplImage* representation of the faces to one that has pixel
intensities between
	 * 0 - 255.
	 *
	 */

void FaceSpace::CreateEigenFaces(){
	int i, j, k;
	int height, width, step, channels;
	int out_height, out_width, out_step, out_channels;

	::u_char* data;
	::u_char* out_data;

	for( k=0; k < nEigens; k++ ){
		eigen_faces[k] = ::cvCreateImage( ::cvGetSize( eigens[k] ) ,
IPL_DEPTH_8U, 1 );
		//    fprintf(outFile, "\n*** Printing Image: %i size ",
eigens[i]->imageSize );

		height = eigens[k]->height;
		width = eigens[k]->width;
		step = eigens[k]->widthStep;
		channels = eigens[k]->nChannels;
		data = (::u_char *)eigens[k]->imageData;

		out_width = eigen_faces[k]->width;
		out_height = eigen_faces[k]->height;
		out_step = eigen_faces[k]->widthStep;
		out_channels = eigen_faces[k]->nChannels;
		out_data = (::u_char *)eigen_faces[k]->imageData;
```

```
            fprintf( outFile, "\n Image (%i)x(%i), widthStep(%i),
channels(%i) ", width, height, step, channels );
            fprintf( outFile, " converted to (%i)x(%i), widthStep(%i),
channels(%i) \n", out_width, out_height, out_step, out_channels );

            float max = FindMax(eigens[k]);
            float min = FindMin(eigens[k]);
            float abs_min = fabs(min);

            max = max + abs_min;


            fprintf( outFile, "Max(%f) and Min(%f) \n", ::fabs(max), min
);


            for(i = 0; i < height; i++){

                  for(j = 0; j < width; j++){
                        CvScalar s;
                        float intensity;
                        int p;
                        s = ::cvGet2D( eigens[k], i, j);

                        intensity = s.val[0];
                        intensity = intensity + abs_min;

                  p = 255*(intensity/max);

                        // fprintf( outFile , "%i ", p );
                        out_data[i*(out_step)+j] = p;

                  }
                  // fprintf( outFile, "\n");
            }
      }
}

      /**
       * The method, NormalizeCoeffs, depending on the race calculates
normalized
       * coeffients of the training coefficients.
       *
       * @return int
       */


int FaceSpace::NormalizeCoeffs(){
      int i, j;

      switch( race ){

      case AA:
            for( j = 0; j < nEigens; j++ ){
                  double max = -100000.00;
                  for( i = 0; i < NUM_AA_TRAINERS; i++){
                        float abs_val = fabs( aa_coeffs[i][j] );
```

```
                        if(abs_val > max)
                              max = abs_val;
                  }

                  for( i = 0; i < NUM_AA_TRAINERS; i++ ){
                        aa_norm_coeffs[i][j] = aa_coeffs[i][j]/max;
                  }
            }
      break;

      case CA:
            for( j = 0; j < nEigens; j++ ){
                  double max = -100000.00;

                  for( i = 0; i < NUM_CA_TRAINERS; i++){
                        float abs_val = fabs( ca_coeffs[i][j] );

                        if(abs_val > max)
                              max = abs_val;
                  }
                  for( i = 0; i < NUM_CA_TRAINERS; i++ ){
                        ca_norm_coeffs[i][j] = ca_coeffs[i][j]/max;
                  }

            }
      break;

      case BLENDED:
            for( j = 0; j < nEigens; j++ ){
                  double max = -100000.00;
                  for( i = 0; i < NUM_BLENDED_TRAINERS; i++){
                        float abs_val = fabs( blended_coeffs[i][j] );

                        if(abs_val > max)
                              max = abs_val;
                  }

                  for( i = 0; i < NUM_BLENDED_TRAINERS; i++ ){
                        blended_norm_coeffs[i][j] =
blended_coeffs[i][j]/max;
                  }

            }
      break;

      }
      return 0;
}

      /**
       * The method, EigenProject, projects the training faces
decomposed
       * in FaceSpace by applying the eigen coefficients to create a
       * fuzzy representation of the original face.
       *
       * @return int
```

```
       */

int FaceSpace::EigenProject(){
      int i;
      fprintf( outFile, "\n*** Eigen Projections ****");
      switch( race ){
      case AA:
            for( i = 0; i < NUM_AA_TRAINERS; i++ )
                  aa_proj[i] = ::cvCreateImage( cvGetSize(
aa_trainers[0] ), IPL_DEPTH_8U, 1);

            for(i = 0; i < NUM_AA_TRAINERS; i++){
                  ::cvEigenProjection( eigens, nEigens, 0, 0,
aa_coeffs[i],  aa_avg, aa_proj[i]);
            }
      break;

      case CA:
            for( i = 0; i < NUM_CA_TRAINERS; i++ )
                  ca_proj[i] = ::cvCreateImage( cvGetSize(
ca_trainers[0] ), IPL_DEPTH_8U, 1);

            for(i = 0; i < NUM_CA_TRAINERS; i++){
                  ::cvEigenProjection( eigens, nEigens, 0, 0,
ca_coeffs[i],  ca_avg, ca_proj[i]);
            }
      break;

      case BLENDED:
            for( i = 0; i < NUM_BLENDED_TRAINERS; i++ )
                  blended_proj[i] = ::cvCreateImage( cvGetSize(
blended_trainers[0] ), IPL_DEPTH_8U, 1);

            for(i = 0; i < NUM_BLENDED_TRAINERS; i++){
                  ::cvEigenProjection( eigens, nEigens, 0, 0,
blended_coeffs[i],  blended_avg, blended_proj[i]);
            }
      break;

      }
      fprintf( outFile, "\n*** End Projection ***");

      return 0;
}

      /**
       * The method, EigenDecomposite, generates a set of coefficients
that
       * represent location of the training faces in FaceSpace.  It uses
the
       * OpenCV method, cvEigenDecomposite, to allocate these
coefficients.
       *
       * @return int
       */

int FaceSpace::EigenDecomposite(){
```

```
	int i;

	switch( race ){

	case AA:
		for(i = 0; i < NUM_AA_TRAINERS; i++){
			::cvEigenDecomposite( aa_trainers[i], nEigens, eigens,
0, 0, aa_avg, aa_coeffs[i]);
		}
	break;

	case CA:
		for(i = 0; i < NUM_CA_TRAINERS; i++){
			::cvEigenDecomposite( ca_trainers[i], nEigens, eigens,
0, 0, ca_avg, ca_coeffs[i]);
		}
	break;

	case BLENDED:
		for(i = 0; i < NUM_BLENDED_TRAINERS; i++){
			::cvEigenDecomposite( blended_trainers[i], nEigens,
eigens, 0, 0, blended_avg, blended_coeffs[i]);
		}
	break;
	}

	return 0;
}

	/**
	 * The method, CreateEigenObjects, uses the OpenCV function that
	 * depending on the number and type of input faces the method
	 * creates EigenObjects that are a set of standard faces for a
	 * corresponding training set.
	 *
	 * @return int
	 */

int FaceSpace::CreateEigenObjects(){

	int i;

	fprintf(outFile, "\n*** Creating Eigen Objects ***");

	switch( race ){

	case AA:
		for(i = 0; i < nEigens; i++)
			eigens[i] = cvCreateImage( cvGetSize(aa_trainers[0]),
IPL_DEPTH_32F, 1 );

		aa_avg = cvCreateImage( cvGetSize(aa_trainers[0]),
IPL_DEPTH_32F, 1);

		criteria.type = CV_TERMCRIT_ITER;// | CV_TERMCRIT_EPS;
		criteria.max_iter = NUM_EIGENS;
		criteria.epsilon = 0.1;
```

```
            ::cvCalcEigenObjects( NUM_AA_TRAINERS, aa_trainers, eigens,
0, 0, 0, &criteria, aa_avg, vals );

            fprintf( outFile, "\n*** Eigen Objects complete ***");

      break;

      case CA:
            for(i = 0; i < nEigens; i++)
                  eigens[i] = cvCreateImage( cvGetSize(ca_trainers[0]),
IPL_DEPTH_32F, 1 );

            ca_avg = cvCreateImage( cvGetSize(ca_trainers[0]),
IPL_DEPTH_32F, 1);

            criteria.type = CV_TERMCRIT_ITER;// | CV_TERMCRIT_EPS;
            criteria.max_iter = NUM_EIGENS;
            criteria.epsilon = 0.1;

            ::cvCalcEigenObjects( NUM_CA_TRAINERS, ca_trainers, eigens,
0, 0, 0, &criteria, ca_avg, vals );

            fprintf( outFile, "\n*** Eigen Objects complete ***");

      break;

      case BLENDED:
            for(i = 0; i < nEigens; i++)
                  eigens[i] = cvCreateImage(
cvGetSize(blended_trainers[0]), IPL_DEPTH_32F, 1 );

            blended_avg = cvCreateImage( cvGetSize(blended_trainers[0]),
IPL_DEPTH_32F, 1);

            criteria.type = CV_TERMCRIT_ITER;// | CV_TERMCRIT_EPS;
            criteria.max_iter = NUM_EIGENS;
            criteria.epsilon = 0.1;

            ::cvCalcEigenObjects( NUM_BLENDED_TRAINERS,
blended_trainers, eigens, 0, 0, 0, &criteria, blended_avg, vals );

            fprintf( outFile, "\n*** Eigen Objects complete ***");

      break;

      }

      return 0;
}

      /**
       * The method, PrintEigenVals, is an output method that simply
       * prints the EigenVals calculated from the EigenFace
coefficients.
       */
```

```
void FaceSpace::PrintEigenVals(){
      int i;

      fprintf( outFile, "\n*** Eigen Values ***");

      for(i = 0; i < nEigens; i++){
            fprintf( outFile, "\n e_val: %i %15.2lf ", i, vals[i]);
      }

}

      /**
       * The method, PrintAvgFace, is an output method that prints
       * the average EigenFace.  The average EigenFace is calculated by
       * a weighted sum of the each of the coefficients or dimensions
       * of the EigenFaces.
       */

void FaceSpace::PrintAvgFace(){
      int i;

      fprintf(outFile, "\nMean Eigen Face :  " );

      for(i = 0; i < nEigens; i++){
            switch( race ){

            case AA:
                  fprintf(outFile, " %f", aa_mean_eigen[i]);
            break;

            case CA:
                  fprintf(outFile, " %f", ca_mean_eigen[i]);
            break;

            case BLENDED:
                  fprintf(outFile, " %f", blended_mean_eigen[i]);
            break;

            }

      }

}

      /**
       * The method, PrintCoefficients, is an output method that prints
       * the Eigen coefficients depending of the race of this space.
       * These coefficients have been calculated at another place within
       * this class.
       */

void FaceSpace::PrintCoefficients(){
      int i,j;

      fprintf(outFile, "\n*** Eigen Coefficients ***\n");;

      switch( race ){
```

```
	case AA:
		for(i = 0; i < NUM_AA_TRAINERS; i++){
			fprintf( outFile , "%i: ", i);
			for(j = 0; j < nEigens; j++){
				fprintf(outFile, "\t%3.2lf", aa_coeffs[i][j]);
			}
			fprintf(outFile,"\n");
		}
	break;

	case CA:
		for(i = 0; i < NUM_CA_TRAINERS; i++){
			fprintf( outFile , "%i: ", i);
			for(j = 0; j < nEigens; j++){
				fprintf( outFile , "\t%1.2lf", ca_coeffs[i][j]);
			}
			fprintf(outFile,"\n");
		}
	break;

	case BLENDED:
		for(i = 0; i < NUM_BLENDED_TRAINERS; i++){
			fprintf( outFile , "%i: ", i);
			for(j = 0; j < nEigens; j++){
				fprintf( outFile , "\t%1.2lf",
blended_coeffs[i][j]);
			}
			fprintf(outFile,"\n");
		}
	break;

	}

}

	/**
	 * The method, PrintNormCoefficients, is an output method that
	 * prints the normalized coefficients of the training faces.
	 * These values have been calculated elsewhere in this class.
	 *
	 */

void FaceSpace::PrintNormCoefficients(){
	int i,j;

	fprintf( outFile, "\n*** Normalized Eigen Coefficients ***\n");

	switch( race ){

	case AA:
		for(i = 0; i < NUM_AA_TRAINERS; i++){
			fprintf( outFile , "%i: ", i);
			for(j = 0; j < nEigens; j++){
				fprintf( outFile , "\t%1.2lf",
aa_norm_coeffs[i][j]);
			}
```

```
                fprintf(outFile,"\n");
            }
      break;

      case CA:
            for(i = 0; i < NUM_CA_TRAINERS; i++){
                fprintf( outFile , "%i: ", i);
                for(j = 0; j < nEigens; j++){
                    fprintf( outFile , "\t%1.2lf",
ca_norm_coeffs[i][j]);
                }
                fprintf(outFile,"\n");
            }
      break;

      case BLENDED:
            for(i = 0; i < NUM_BLENDED_TRAINERS; i++){
                fprintf( outFile , "%i: ", i);
                for(j = 0; j < nEigens; j++){
                    fprintf( outFile , "\t%1.2lf",
blended_norm_coeffs[i][j]);
                }
                fprintf(outFile,"\n");
            }
      break;

      }

}

      /**
       * The method, PrintMeanDistance, is an output method that prints
the
       * average distance per dimension of the face space.  The average
distance
       * has been calculated at another place within this class.
       */

void FaceSpace::PrintMeanDistance(){

      int i;

      fprintf( outFile, "\nPrinting Mean Distance\n");

      switch(race){
            case AA:
                for(i = 0; i < NUM_AA_TRAINERS; i++){
                    fprintf( outFile, "%i: %3.2f\n", i,
aa_mean_dist[i] );
                }
                break;

            case CA:
                for(i = 0; i < NUM_CA_TRAINERS; i++){
                    fprintf( outFile, "%i: %3.2f\n", i,
ca_mean_dist[i] );
                }
```

```
                  break;
            case BLENDED:
                  for(i = 0; i < NUM_BLENDED_TRAINERS; i++){
                        fprintf( outFile, "%i: %3.2f\n", i,
blended_mean_dist[i] );
                  }
            break;

            default:
                  fprintf( outFile, "\nUnknown Race");
      }

}

      /**
       * The method, SetRace, sets a class variable, race,
       * dependent on the parameter passed. Not only does it set
       * the race but this method also opens key output files to
       * evaluate the FaceSpace(s).
       */

void FaceSpace::SetRace(int r){
      race = r;

      switch( race ){

      case AA:
            if((outFile = fopen("output/aa_space", "w")) == NULL)
                  printf(" error creating file\n");
      break;

      case CA:
            if((outFile = fopen("output/ca_space", "w")) == NULL)
                  printf(" error creating file\n");
      break;

      case BLENDED:
            if((outFile = fopen("output/blended_space", "w")) == NULL)
                  printf(" error creating file\n");
      break;

      }

}

      /**
       * The method, CreateTrainingSet, depending on the race, creates a
       * training set of faces for the FaceSpace.  It grabs the training
       * images already stored in the Database and allocates memory
       * for them to be used in this class.
       *
       * @param FaceDatabase db -- the FaceDatabase centralized location
of faces
       */

void FaceSpace::CreateTrainingSet(::FaceDatabase db){
      int i;
```

```
	switch( race ){

	case AA:
		for(i = 0; i < NUM_AA_TRAINERS; i++){
			aa_trainers[i] = ::cvCloneImage( db.aa_training_set[i]
);
		}
	return;
	case CA:
		for(i = 0; i < NUM_CA_TRAINERS; i++){
			ca_trainers[i] = ::cvCloneImage( db.ca_training_set[i]
);
		}
	return;
	case BLENDED:
		for(i = 0; i < NUM_BLENDED_TRAINERS; i++){
			blended_trainers[i] = ::cvCloneImage(
db.bl_training_set[i] );
		}
	return;
	default:
		fprintf( outFile , "Could not create training set for
race.\n" );
	return;
	}

}

	/**
	 * The method, GetAverageFace, returns the average face of the
corresponding
	 * face space. A class variable, race, is set when a FaceSpace is
created.
	 *
	 * @return IplImage* -- average face
	 */

IplImage* FaceSpace::GetAverageFace(){
	switch( race ){

	case AA:
		return aa_avg;
	break;
	case CA:
		return ca_avg;
	break;
	case BLENDED:
		return blended_avg;
	break;
	}
	return NULL;
}


	/**
	 * This the constructor that instantiates the class.  It sets one
	 * class variable nEigens.
```

```
	 *
	 */


FaceSpace::FaceSpace()
{
	nEigens = 30;

}

	/**
	 * This is the destructor, it is called when the program exits.
It
	 * does a number of tasks like cleanup all the images in this
FaceSpace:
	 * Afican American, Caucasian, Eigen, EigenFaces, etc.
	 */

FaceSpace::~FaceSpace()
{
	int i;

	for( i = 0; i < NUM_EIGENS; i++){
		::cvReleaseImage( &eigens[i] );
		::cvReleaseImage( &eigen_faces[i] );
	}

	if(race == AA){
		::cvReleaseImage( &aa_avg );
		::cvReleaseImage( &aa_avg_face );
		::cvReleaseImage( &ca_avg_face );

		for( i = 0; i < NUM_AA_TESTERS; i++){
			::cvReleaseImage( &aa_test[i] );
		}

		for( i = 0; i < NUM_AA_TRAINERS; i++){
			::cvReleaseImage( &aa_proj[i] );
			::cvReleaseImage( &aa_trainers[i] );
		}
	}


	if(race == CA){
		::cvReleaseImage( &ca_avg );
		::cvReleaseImage( &ca_avg_face );

		for( i = 0; i < NUM_CA_TESTERS; i++){
			::cvReleaseImage( &ca_test[i] );
		}

		for( i = 0; i < NUM_CA_TRAINERS; i++){
			::cvReleaseImage( &ca_proj[i] );
			::cvReleaseImage( &ca_trainers[i] );
		}
```

```
      }

}

/*************************************************************************
 * System: Face Classification Recognition System.
 *
 * Software for Thesis:
 *          NEUROCOGNITIVE INSPIRED HIERARCHICAL FACE RECOGNITION SYSTEM
 *
 * File name: FaceReader.h
 *
 * Input file name(s):  highgui.h
 *                                  cv.h
 *                                  cvaux.h
 *                                  cxcore.h
 *                                  cvtypes.h
 *                                  iostream.h
 *                                  stdlib.h
 *                                  stdio.h
 *                                  fstream.h
 *                                  cstring.h
 *
 * Output file name(s):
 *
 * Author: Ryan Wilkins
 * Date Last Modified: 07/01/07
 * Purpose:
 *
 *************************************************************************/
#ifndef FACEREADER_H_
#define FACEREADER_H_

#include <highgui.h>
#include <cv.h>
#include <cvaux.h>
#include <cxcore.h>
#include <cvtypes.h>
#include <cxtypes.h>
#include <iostream>
#include <stdlib.h>
#include <stdio.h>
#include <fstream>
#include <string>


#define NUM_AA_TRAINERS  30
#define NUM_CA_TRAINERS  30
#define NUM_BL_TRAINERS 30
#define NUM_AA_TEST  30
#define NUM_CA_TEST  30
#define NUM_BL_TEST 30
#define NUM_AA_KNOWN  30
#define NUM_CA_KNOWN  30
#define NUM_BL_KNOWN 30

#define NUM_AA_FACES 200
```

```
#define NUM_CA_FACES 200

class FaceReader
{

public:


      FaceReader();
      virtual ~FaceReader();


      int ReadAAFile();
      int ReadCAFile();
      int Read_File();
      int CreateAAFaceSet();
      int CreateCAFaceSet();
      void PrintAAFaceSet();
      void PrintCAFaceSet();
      int GetNumFaces();
      int GetNumAA();
      int GetNumCA();
      int ReadCAKnownFile();
      int ReadAAKnownFile();
      int ReadBLKnownFile();
      int ReadAATestFile();
      int ReadCATestFile();
      int ReadBLTestFile();
      int ReadCATrainFile();
      int ReadAATrainFile();
      int ReadBLTrainFile();

      int CreateAATrainingSet();
      int CreateAAKnownSet();
      int CreateAATestingSet();

      int CreateCATrainingSet();
      int CreateCAKnownSet();
      int CreateCATestingSet();

      int CreateBLTrainingSet();
      int CreateBLKnownSet();
      int CreateBLTestingSet();

      bool InAATrainSet( int i );
      bool InBLTrainSet( int i, char r );
      bool InCATrainSet( int i );

      bool InAAKnownSet( int i );
      bool InBLKnownSet( int i, char r );
      bool InCAKnownSet( int i );

      int AddAATrainIndex(int n, int i);
      int AddAAKnownIndex(int n, int i);

      int AddCATrainIndex(int n, int i);
      int AddCAKnownIndex(int n, int i);
```

```
        int AddBLTrainIndex(int n, int i, char r);
        int AddBLKnownIndex(int n, int i, char r);

        ::IplImage* aa_faces[NUM_AA_FACES];
        ::IplImage* ca_faces[NUM_CA_FACES];
        std::string aa_train_paths[NUM_AA_FACES];
        std::string ca_train_paths[NUM_CA_FACES];

        std::string ca_file_names[NUM_AA_FACES];
        std::string aa_file_names[NUM_AA_FACES];

        std::string ca_known_fn[NUM_CA_KNOWN];
        std::string aa_known_fn[NUM_AA_KNOWN];
        std::string bl_known_fn[NUM_BL_KNOWN];

        std::string ca_test_fn[NUM_CA_TEST];
        std::string aa_test_fn[NUM_AA_TEST];
        std::string bl_test_fn[NUM_BL_TEST];

        std::string aa_train_fn[NUM_AA_TRAINERS];
        std::string ca_train_fn[NUM_CA_TRAINERS];
        std::string bl_train_fn[NUM_BL_TRAINERS];

        std::string ca_known_fp[NUM_CA_KNOWN];
        std::string aa_known_fp[NUM_AA_KNOWN];
        std::string bl_known_fp[NUM_BL_KNOWN];

        std::string ca_test_fp[NUM_CA_TEST];
        std::string aa_test_fp[NUM_AA_TEST];
        std::string bl_test_fp[NUM_BL_TEST];

        std::string aa_train_fp[NUM_AA_TRAINERS];
        std::string ca_train_fp[NUM_CA_TRAINERS];
        std::string bl_train_fp[NUM_BL_TRAINERS];

        int aa_train_indices[NUM_AA_TRAINERS];
        int aa_known_indices[NUM_AA_KNOWN];

        int ca_known_indices[NUM_CA_TRAINERS];
        int ca_train_indices[NUM_CA_KNOWN];

        int bl_known_indices[NUM_BL_KNOWN][2];
        int bl_train_indices[NUM_BL_TRAINERS][2];


        FILE* outFile;
};

#endif /*FACEREADER_H_*/

/*********************************************************************
 * System: Face Classification Recognition System.
 *
 * Software for Thesis:
 *          NEUROCOGNITIVE INSPIRED HIERARCHICAL FACE RECOGNITION SYSTEM
 *
```

```
 * File name: FaceClassifier.cpp
 *
 * Input file name(s):  FaceClassifier.h
 *                                  FCRS.h
 *                                  FaceReader.h
 *                                  FaceDatabase.h
 *                                  FaceSpace.h
 *
 * Output file name(s): classifier_out
 *
 * Author: Ryan Wilkins
 * Date Last Modified: 07/01/07
 * Purpose: The level of the system where a face is classified
 *
 ****************************************************************/
#include "FaceClassifier.h"
#include "FCRS.h"
#include "FaceReader.h"
#include "FaceDatabase.h"
#include "FaceSpace.h"


      FaceSpace* aa;
      FaceSpace* ca;
      FaceSpace* bl;
      FaceDatabase* db;

      /**
       * The constructor of FaceClassifier sets key pointers to the
FaceSpace(s)
       * and the FaceDatabase.  These pointers allow this class to
access the
       * class methods of the parameters.
       *
       * @param FaceSpace* a -- pointer to the African American face
space
       * @param FaceSpace* c -- pointer to the Caucasian face space
       * @param FaceSpace* bl -- pointer to the Blended face space
       * @param FaceDatabase* d -- pointer to the FaceDatabase
       */


FaceClassifier::FaceClassifier(FaceSpace* a, FaceSpace* c, FaceSpace*
b, FaceDatabase* d)
{
      aa = a;
      ca = c;
      db = d;
      bl = b;

      if((outFile = fopen("output/classifier_out", "w")) == NULL)
            printf(" error creating file\n");

      fprintf( outFile, "\n*** Starting my FaceClassifier ***\n");
      fprintf( outFile, "Initializing...\n");
      fprintf( outFile, " AA Space built with %d eigen faces.\n", aa-
>nEigens );
```

```
}

      /**
       * The deconstructor of FaceClassifier is called when the system
exits and
       * cleans up memory allocation if needed.
       */

FaceClassifier::~FaceClassifier()
{

}

      /**
       * The method, initCATestSet, initializes a test of Caucasian
       * indices.
       */

void FaceClassifier::initCATestSet(){
      int i, m;
      fprintf( outFile, "\nInitializing test set for %i \n", db-
>aa_faces );
      m = db->ca_faces-db->num_ca_train;

      for(i = 0; i < NUM_CA_TESTERS; i++){
            ca_test_index[i] = (::rand() % m)+db->num_ca_train+db-
>aa_faces;
      }

      fprintf( outFile, "...Done Initializing\n"     );
}

      /**
       * The method, initAATestSet, initializes a test of African
American
       * indices.
       */

void FaceClassifier::initAATestSet(){
      int i, m;
      fprintf( outFile, "\nInitializing test set for %i \n", db-
>aa_faces);
      m = db->aa_faces-db->num_aa_train;

      for(i = 0; i < NUM_AA_TESTERS; i++){
            aa_test_index[i] = (::rand() % m)+db->num_aa_train;
      }

      fprintf( outFile, "...Done Initializing\n"     );
}

      /**
       * The method, ClassifyComponentMahalanobis, classifies a probe by
only evaluating
       * a set of components which less than the entire set.
```

```
       *
       * @param IplImage* probe -- the unknown image to classify
       * @param int[] components -- the components to use when classify
the probe
       * @param n -- the length of the component set
       * @return int -- a integer that corresponds to a race
       */

int FaceClassifier::ClassifyComponentMahalanobis(::IplImage *probe, int
components[], int n )
{
      float dist[NUM_BL_TRAINERS];
      float min_distance;
      int i, return_val, min_index;

      fprintf( outFile, "\nClassifying Image via Component
Mahalanobis\n" );

      IplImage* train;

      for( i = 0; i < NUM_BL_TRAINERS; i++ ){
            train = ::cvCloneImage( db->bl_training_set[i] );
            dist[i] = bl->FindComponentMahalanobisDistance(probe, train,
components, n);
      }

      min_index = FindMin( dist );
      min_distance = dist[min_index];

      if( min_index % 2 == 0 )
            return_val = AA;
      else
            return_val = CA;

      return return_val;

}

      /**
       * The method, ClassifyImageMahalanobisVote, classifies an input
unknown probe
       * using the Mahalanobis distance and a voting scheme.  The images
close to the probe
       * image in Face space get a vote or a number of votes depending
on the scheme.
       * The race class with the most votes classifies the image.
       *
       * @param IplImage* probe -- the unknown face
       * @param int num_neigbors -- the number of neighbors that get a
vote
       * @param bool weighted -- true or false whether it is a weighted
vote or not
       * @return int -- the integers representation of the classified
race
       */
```

```
int FaceClassifier::ClassifyImageMahalanobisVote(::IplImage *probe, int
num_neighbors, bool weighted){

      float n_dist[num_neighbors];
      float dist[NUM_BL_TRAINERS];
      int neighbors[NUM_BL_TRAINERS];
      int i, ca_votes = 0, aa_votes = 0, return_val;

      fprintf( outFile, "\nClassifying Image via Voting and Mahalanobis
Distance\n" );

      IplImage* train;

      for( i = 0; i < NUM_BL_TRAINERS; i++ ){
            train = ::cvCloneImage( db->bl_training_set[i] );
            dist[i] = bl->FindMahalanobisDistance(probe, train);
            neighbors[i] = i;
      }

      Sort( dist, neighbors, NUM_BL_TRAINERS );

      for( i = 0; i < num_neighbors; i++ ){
            n_dist[i] = dist[i];

            fprintf( outFile, "Vote %i goes to %i\n", i, neighbors[i] );

            if(!weighted){
                  if( neighbors[i] % 2 == 0 )
                        aa_votes++;
                  else
                        ca_votes++;

            }else{
                  if( neighbors[i] % 2 == 0 )
                        aa_votes += num_neighbors - i;
                  else
                        ca_votes += num_neighbors - i;

            }
      }

      if( aa_votes > ca_votes ){
            return_val = AA;
      }else
            return_val = CA;

      fprintf( outFile, "Image Classified as %i with CA votes = %i and
AA votes = %i \n", return_val, ca_votes, aa_votes  );

      return return_val;
}

      /**
       * The method, ClassifyImageMahalanobis, classifies an image using
the
       * Mahalanobis distance.
       *
```

```
       * @param IplImage* probe -- unknown face
       * @return int -- the integer representation of the classified
race
       */

int FaceClassifier::ClassifyImageMahalanobis(::IplImage *probe ){

      float min_distance;
      int i, min_index, return_val;
      float dist[NUM_BL_TRAINERS];


      fprintf( outFile, "\nClassifying Image via Mahalonobis Distance\n"
);


      /**
       *  Changed implementation of this code so that passing a blended
       *  training face and returning the distance into a an array of
distances.
       *  Then choosing the smallest distance in the array of distances
and looking
       *  up the face with the smallest distance.  Knowing the race of a
certain
       *  training face is simple due to the fact that we know what
index it is
       *  in the set.
       */

      IplImage* train;

      for( i = 0; i < NUM_AA_TRAINERS; i++ ){
            train = ::cvCloneImage( db->bl_training_set[i] );
            dist[i] = bl->FindMahalanobisDistance( probe, train );
      }

      min_index = FindMin( dist );
      min_distance = dist[min_index];



      if( min_index % 2 == 0 )
            return_val = AA;
      else
            return_val = CA;

//    cvReleaseImage( &train );

      fprintf( outFile, "Image Classified as %i by %f \n", return_val,
min_distance );

      return return_val;
}

      /**
       * The method, Swap, is a general swap method that takes
       * two indices to swap and an integer array to do the swapping.
```

```
       *
       * @param int x -- the old location
       * @param int y -- the new location
       * @param int[] a -- the array in which the swap takes place
       */

void FaceClassifier::Swap( int x, int y, int a[NUM_BL_TRAINERS] ){
      int temp;
      temp = a[x];
      a[x] = a[y];
      a[y] = temp;
}

      /**
       * The method, Swap, is a general swap method that takes
       * two indices to swap and a pointer to an float array
       * to do the swapping.
       *
       * @param int x -- the old location
       * @param int y -- the new location
       * @param float* a -- pointer to the array in which the swap takes
place
       */


void FaceClassifier::Swap( int x, int y, float* a ){
      float temp;
      temp = a[x];
      a[x] = a[y];
      a[y] = temp;
}

      /**
       * The method, Sort, takes three separate parameters in which two
       * arrays it sorts.  The float array is the distances of the
unknown
       * face from known faces and the int array are the indices or
ranks
       * of those faces.
       *
       * @param float[] fd -- distances of the unknown image from the
knowns
       * @param int[] faces -- are the rank of these images
       * @param int size -- is the size of these arrays
       */

void FaceClassifier::Sort( float fd[NUM_BL_TRAINERS], int
face_index[NUM_BL_TRAINERS], int size ){
      int i, j, min;


      for( i = 0; i < size - 1; i++){
            min = i;
            for( j = i+1; j < size; j++){
                  if( fd[j] < fd[min] )
                        min = j;
            }
```

```
            Swap( i , min , fd);
            Swap( i, min, face_index);
      }
}

      /**
       * The method, FindMin, finds the minimum float within an array of
floats.
       * In this instance, the floating point array is an array of
distances.
       *
       * @param float[] d -- the distance in which a minimum will be
found
       * @return int -- the index of the minimum distance
       */

int FaceClassifier::FindMin( float d[] ){
      int i, min;
      float min_dist = 10000000.0;

      for( i = 0; i < NUM_BL_TRAINERS; i++ ){
            if( min_dist > d[i] ){
                  min_dist = d[i];
                  min = i;
            }

      }
      return min;
}

      /**
       * The method, ClassifyImage, classifies an unknown probe face
using the
       * Euclidean distance method and a single threshold.
       *
       * @param IplImage* probe -- the unknown image
       * @return int -- the integer representation of the classified
race or in this case
       *                        both and unknown can be possible
outcomes of classification
       */

int FaceClassifier::ClassifyImage(::IplImage *probe ){
      double aa_dist, ca_dist;
      char* c;
      float aa_thresh, ca_thresh;
      int return_val;

//    aa_thresh = aa->max_distance + aa->std_dev;
//    ca_thresh = ca_fs.max_distance + ca_fs.std_dev;

      aa_thresh = aa->avg_distance + aa->std_dev;
      ca_thresh = ca->avg_distance + ca->std_dev;

      aa_dist = aa->CalcUnknownAvgDistance(probe);
      ca_dist = ca->CalcUnknownAvgDistance(probe);
```

```
	// 0 is white 1 is african 3 both 4 unknown

	if(aa_dist < aa_thresh && ca_dist < ca_thresh){
		c = "CA";
		return_val = BOTH;
	}else if(aa_dist < aa_thresh){
		c = "A";
		return_val = AA;
	}else if(ca_dist < ca_thresh){
		c = "C";
		return_val = CA;
	}else{
		c = "U";
		return_val = UNKNOWN;
	}
	fprintf( outFile,
"\t%s\t\t%3.2f\t\t%3.2f\t\t%3.2f\t\t%3.2f\t\t\n",c, aa_dist, aa_thresh,
ca_dist, ca_thresh);


	return return_val;
}

	/**
	 * The method, ClassifyImageWithRange, classifies an unknown face
	 * using the Euclidean distance and a dual threshold.  A dual
threshold
	 * or range in which to classify an unknown face.
	 *
	 * @param IplImage* probe -- the unknown face
	 * @return int -- the integer representation of the classified
race or in this case
	 *                          both and unknown can be possible
outcomes of classification
	 */

int FaceClassifier::ClassifyImageWithRange(::IplImage* probe){
	double aa_dist, ca_dist;
	char* c;
	float aa_thresh_low, aa_thresh_high, ca_thresh_low,
ca_thresh_high;
	int return_val;
	aa_thresh_low = aa->avg_distance - aa->std_dev;
	aa_thresh_high = aa->avg_distance + aa->std_dev;

	ca_thresh_low = ca->avg_distance - ca->std_dev;
	ca_thresh_high = ca->avg_distance + ca->std_dev;

	aa_dist = aa->CalcUnknownAvgDistance(probe);
	ca_dist = ca->CalcUnknownAvgDistance(probe);

	/**
	 *    The first condition is if they are in both ranges, the
second condition is that the image is within
	 *  the African Amer. distance and the third would be within
Caucasian distance.
	 *
```

```
       */

      if( (aa_dist > aa_thresh_low && aa_dist < aa_thresh_high) &&
(ca_dist > ca_thresh_low && ca_dist < ca_thresh_high) ){
            c = "CA";
            return_val = BOTH;
      }else if( aa_dist > aa_thresh_low && aa_dist < aa_thresh_high ){
            c = "A";
            return_val = AA;
      }else if( ca_dist > ca_thresh_low && ca_dist < ca_thresh_high ){
            return_val = CA;
            c = "C";
      }else{
            return_val = UNKNOWN;
            c = "*";
      }

      fprintf( outFile, "\t%s\t\t%3.2f\t\t[%3.2f,
%3.2f]\t\t%3.2f\t\t[%3.2f, %3.2f]\t\t",c, aa_dist,
aa_thresh_low,aa_thresh_high, ca_dist, ca_thresh_low,ca_thresh_high);

      return return_val;
}

      /**
       * The method, ClassifyTestSetInRange, is used to classify a set
of test faces
       * so that many faces are tested instead of a single one at a
time.  This
       * method uses early attempts of classification, Euclidean
distance range method.
       */

void FaceClassifier::ClassifyTestSetInRange(){
      int i;
      fprintf( outFile, "\n*** Classifying Images Within Range *** \n");
      fprintf( outFile,
"Probe\tR\tClass\t\tAD\t\t\tA(low,high)\t\t\tCD\t\t\tC(low,high)");
      ::IplImage* probe = NULL;

      for(i = 0; i < NUM_AA_TESTERS; i++){

            fprintf( outFile, "\n%i\t\tA", i);
            probe = ::cvCloneImage( db->aa_test_set[i] );
            ClassifyImageWithRange(probe);

      }

      fprintf( outFile, "\n");

      for(i = 0; i < NUM_CA_TESTERS; i++){

            fprintf( outFile, "\n%i\t\tC", i);
            probe = ::cvCloneImage( db->ca_test_set[i] );
          ClassifyImageWithRange(probe);

      }
```

```
	cvReleaseImage( &probe );
}

	/**
	 * The method, ClassifyTestSet, is used to classify a set of test
faces
	 * so that many faces are tested instead of a single one at a
time.  This
	 * method uses early attempts of classification, Euclidean
distance method.
	 */

void FaceClassifier::ClassifyTestSet(){
	int i;
	fprintf( outFile, "\n*** Classifying Images *** \n");
	fprintf( outFile,
"Probe\tR\tClass\t\tAD\t\t\tAT\t\t\tCD\t\t\tCT");

	::IplImage* probe = NULL;

	for(i = 0; i < NUM_AA_TESTERS; i++){

		fprintf( outFile, "\n%i\t\tA",i);
		probe = ::cvCloneImage( db->aa_test_set[i] );
		ClassifyImage(probe);

	}

	fprintf( outFile, "\n");

	for(i = 0; i < NUM_CA_TESTERS; i++){

		fprintf( outFile, "\n%i\t\tC", i);
		probe = ::cvCloneImage( db->ca_test_set[i] );
		ClassifyImage(probe);


	}

	::cvReleaseImage( &probe );
}

	/**
	 * The method, PrintTestCoeffs, is an output method that prints
the
	 * coefficients of the test set to an output file.
	 */

void FaceClassifier::PrintTestCoeffs(){
	int i;
	fprintf( aa->outFile, "Probe\tR\tA Coeffs\tC Coeffs\n");
	fprintf( ca->outFile, "Probe\tR\tA Coeffs\tC Coeffs\n");

	IplImage* probe = NULL;

	for(i = 0; i < NUM_AA_TESTERS; i++){
```

```
            fprintf( aa->outFile, "\n%i,A,", i);
            fprintf( ca->outFile, "\n%i,A,", i);

            probe = ::cvCloneImage( db->aa_test_set[i] );

            aa->GetUnknownImageCoeffs(probe);
            ca->GetUnknownImageCoeffs(probe);
      }

      for(i = 0; i < NUM_CA_TESTERS; i++){

            fprintf( ca->outFile, "\n%i,C,",i);
            fprintf( aa->outFile, "\n%i,C,",i);

            probe = ::cvCloneImage( db->ca_test_set[i] );

            aa->GetUnknownImageCoeffs(probe);
            ca->GetUnknownImageCoeffs(probe);
      }

      cvReleaseImage( &probe );

}

      /**
       * The method, PrintTestSets, prints the indices of the test faces
to an
       * output file
       */

void FaceClassifier::PrintTestSets(){
      int i;
      fprintf( outFile, "\n CA Test Set {");
      for(i = 0; i < NUM_CA_TESTERS; i++){
            fprintf( outFile, " %d,", ca_test_index[i]);
      }

      fprintf( outFile, "}\n AA Test Set {");
      for(i = 0; i < NUM_AA_TESTERS; i++){
            fprintf( outFile, " %d, ", aa_test_index[i]);
      }
      fprintf( outFile, "}\n" );
}

/*************************************************************************
 * System: Face Classification Recognition System.
 *
 * Software for Thesis:
 *          NEUROCOGNITIVE INSPIRED HIERARCHICAL FACE RECOGNITION SYSTEM
 *
 * File name: FaceClassifier.h
 *
 * Input file name(s):  highgui.h
 *                                  cv.h
 *                                  stdio.h
 *                                  fstream.h
 *                                  cstring.h
```

```
 *
 * Output file name(s):
 *
 * Author: Ryan Wilkins
 * Date Last Modified: 07/01/07
 * Purpose:
 *
 ************************************************************************/
#ifndef FACECLASSIFIER_H_
#define FACECLASSIFIER_H_


#include <highgui.h>
#include <cv.h>
#include <iostream>
#include <stdio.h>
#include <fstream>
#include <FaceSpace.h>
#include <FaceDatabase.h>
#include <FaceReader.h>
#include <cstring>

#define NUM_CA_TESTERS 30
#define NUM_AA_TESTERS 30

#define NUM_AA_TRAINERS 30
#define NUM_CA_TRAINERS 30

#define CA 0
#define AA 1
#define BOTH 3
#define UNKNOWN 4

class FaceClassifier{


      public:

      FaceClassifier(FaceSpace* a, FaceSpace* c, FaceSpace* b,
FaceDatabase* d);
      virtual ~FaceClassifier();

      int FindMin( float d[] );
      void Swap( int x, int y, int a[NUM_BL_TRAINERS] );
      void Swap( int x, int y, float* a );
      int ClassifyComponentMahalanobis(::IplImage *probe, int
components[], int n );
      void Sort( float fd[NUM_BL_TRAINERS], int
face_index[NUM_BL_TRAINERS], int size );
      int ClassifyImageMahalanobisVote(::IplImage *probe, int
num_neighbors, bool weighted);
      int ClassifyImageMahalanobis(::IplImage *probe );
      int ClassifyImage(::IplImage *probe );
      void ClassifyTestSet();
      void ClassifyTestSetInRange();
      int ClassifyImageWithRange(::IplImage* probe);
      void PrintTestCoeffs();
```

```
      void PrintTestSets();
      void initAATestSet();
      void initCATestSet();

      FILE* outFile;
      int   ca_test_index[NUM_CA_TESTERS];
      int aa_test_index[NUM_AA_TESTERS];

      float rank_dist_ca[NUM_AA_TRAINERS];
      float rank_dist_aa[NUM_CA_TRAINERS];

      int rank_aa[NUM_AA_TRAINERS];
      int rank_ca[NUM_CA_TRAINERS];
};


#endif /*FACECLASSIFIER_H_*/

/*************************************************************************
 * System: Face Classification Recognition System.
 *
 * Software for Thesis:
 *          NEUROCOGNITIVE INSPIRED HIERARCHICAL FACE RECOGNITION SYSTEM
 *
 * File name: FaceDatabase.cpp
 *
 * Input file name(s):  FaceDatabase.h
 *
 * Output file name(s): database_out
 *
 * Author: Ryan Wilkins
 * Date Last Modified: 07/01/07
 * Purpose: The class that represents centralized location of all the
 *               faces stored
 *
 ************************************************************************/
#include "FaceDatabase.h"


using namespace std;

      /**
       * The method, InsertImages, inserts all of the images
       * read in by the FaceReader.
       *
       * @param FaceReader fr -- the FaceReader class so that the method
can
       *                                      access needed variables and
methods
       * @return int
       */

int FaceDatabase::InsertImages(FaceReader fr){
      int i;

      fprintf( outFile, "Inserting images %i \n", image_index);
```

```
	for(i = image_index; i < fr.GetNumFaces(); i++){
		if(image_index < fr.GetNumAA()){
			all_faces[i] = ::cvCloneImage(fr.aa_faces[i]);
			image_index++;
		}else{
			all_faces[i] = ::cvCloneImage(fr.ca_faces[i-
fr.GetNumCA()]);
			image_index++;
		}
	}

	;

	fprintf( outFile, "Done inserting images %i \n", image_index );

	return 0;
}

	/**
	 * The method, LoadAATrainingSet, loads the African American
training set images into
	 * the FaceDatabase for quick access.
	 *
	 * @param FaceReader fr -- the FaceReader class so that this
method can
	 *                                      access its methods
	 * @return int
	 */

int FaceDatabase::LoadAATrainingSet(FaceReader fr){

	int i;

	fprintf( outFile, "\nLoading AA Training Set \n");

	for(i = 0; i < NUM_AA_TRAINERS; i++){
		IplImage* temp = ::cvLoadImage(fr.aa_train_fp[i].c_str());
		aa_training_set[i] = ::cvCreateImage( cvSize(100,110),
IPL_DEPTH_8U, 1 );
		cvCvtColor( temp, aa_training_set[i], CV_BGR2GRAY );
		cvReleaseImage( &temp );
	}

	fprintf( outFile, "Done...\n");

	return 0;
}

	/**
	 * The method, LoadBLTrainingSet, loads the Blended training set
images into
	 * the FaceDatabase for quick access.
	 *
	 * @param FaceReader fr -- the FaceReader class so that this
method can
	 *                                      access its methods
	 * @return int
```

```
       */


int FaceDatabase::LoadBLTrainingSet(FaceReader fr){

      int i;

      fprintf( outFile, "\nLoading BL Training Set \n");

      for(i = 0; i < NUM_BL_TRAINERS; i++){
            IplImage* temp = ::cvLoadImage(fr.bl_train_fp[i].c_str());
            bl_training_set[i] = ::cvCreateImage( cvSize (100, 110),
IPL_DEPTH_8U, 1 );
            cvCvtColor( temp, bl_training_set[i], CV_BGR2GRAY );
            cvReleaseImage( &temp );
      }

      fprintf( outFile, "Done...\n");

      return 0;
}

      /**
       * The method, LoadCATrainingSet, loads the Caucasian training set
images into
       * the FaceDatabase for quick access.
       *
       * @param FaceReader fr -- the FaceReader class so that this
method can
       *                                        access its methods
       * @return int
       */

int FaceDatabase::LoadCATrainingSet(FaceReader fr){

      int i;

      fprintf( outFile, "\nLoading CA Training Set \n");

      for(i = 0; i < NUM_CA_TRAINERS; i++){
            IplImage* temp = ::cvLoadImage(fr.ca_train_fp[i].c_str());
            ca_training_set[i] = ::cvCreateImage( cvSize (100,110),
IPL_DEPTH_8U, 1 );
            cvCvtColor( temp, ca_training_set[i], CV_BGR2GRAY );
            cvReleaseImage( &temp );
      }

      fprintf( outFile, "Done...\n");

      return 0;
}

      /**
       * The method, LoadAAKnownSet, loads the African American known
set images into
       * the FaceDatabase for quick access.
       *
```

```
       * @param FaceReader fr -- the FaceReader class so that this
method can
       *                                        access its methods
       * @return int
       */

int FaceDatabase::LoadAAKnownSet(FaceReader fr){
      int i;

      fprintf( outFile, "\nLoading AA Known Set \n");

      for(i = 0; i < NUM_AA_KNOWN; i++){
            IplImage* temp = ::cvLoadImage(fr.aa_known_fp[i].c_str());
            aa_known_set[i] = ::cvCreateImage( cvSize(100,110),
IPL_DEPTH_8U, 1 );
            cvCvtColor( temp, aa_known_set[i], CV_BGR2GRAY );
            cvReleaseImage( &temp );
      }

      fprintf( outFile, "Done...\n");

      return 0;
}

      /**
       * The method, LoadBLKnownSet, loads the Blended known set images
into
       * the FaceDatabase for quick access.
       *
       * @param FaceReader fr -- the FaceReader class so that this
method can
       *                                        access its methods
       * @return int
       */

int FaceDatabase::LoadBLKnownSet(FaceReader fr){
      int i;

      fprintf( outFile, "\nLoading BL Known Set \n");

      for(i = 0; i < NUM_BL_KNOWN; i++){
            IplImage* temp = ::cvLoadImage(fr.bl_known_fp[i].c_str());
            bl_known_set[i] = ::cvCreateImage( cvSize(100,110) ,
IPL_DEPTH_8U, 1 );
            cvCvtColor( temp, bl_known_set[i], CV_BGR2GRAY );
            cvReleaseImage( &temp );
      }

      fprintf( outFile, "Done...\n");

      return 0;
}

      /**
       * The method, LoadCAKnownSet, loads the Caucasian known set
images into
       * the FaceDatabase for quick access.
```

```
	 *
	 * @param FaceReader fr -- the FaceReader class so that this
method can
	 *                                        access its methods
	 * @return int
	 */


int FaceDatabase::LoadCAKnownSet(FaceReader fr){
      int i;

      fprintf( outFile, "\nLoading CA Known Set \n");

      for(i = 0; i < NUM_CA_KNOWN; i++){
            IplImage* temp = ::cvLoadImage(fr.ca_known_fp[i].c_str());
            ca_known_set[i] = ::cvCreateImage( cvSize(100,110),
IPL_DEPTH_8U, 1 );
            cvCvtColor( temp, ca_known_set[i], CV_BGR2GRAY );
            cvReleaseImage( &temp );
      }

      fprintf( outFile, "Done...\n");

      return 0;
}

      /**
       * The method, LoadAATestSet, loads the African American test set
images into
       * the FaceDatabase for quick access.
       *
       * @param FaceReader fr -- the FaceReader class so that this
method can
       *                                        access its methods
       * @return int
       */

int FaceDatabase::LoadAATestSet(FaceReader fr){
      int i;

      fprintf( outFile, "\nLoading AA Test Set \n");

      for(i = 0; i < NUM_AA_TEST; i++){
            IplImage* temp = ::cvLoadImage(fr.aa_test_fp[i].c_str());
            aa_test_set[i] = ::cvCreateImage( cvSize(100,110),
IPL_DEPTH_8U, 1 );
            cvCvtColor( temp, aa_test_set[i], CV_BGR2GRAY );
            cvReleaseImage( &temp );
      }

      fprintf( outFile, "Done...\n");


      return 0;
}

      /**
```

```
	 * The method, LoadBLTestSet, loads the Blended test set images
into
	 * the FaceDatabase for quick access.
	 *
	 * @param FaceReader fr -- the FaceReader class so that this
method can
	 *                                        access its methods
	 * @return int
	 */

int FaceDatabase::LoadBLTestSet(FaceReader fr){
      int i;

      fprintf( outFile, "\nLoading BL Test Set \n");

      for(i = 0; i < NUM_BL_TEST; i++){
            IplImage* temp = ::cvLoadImage(fr.bl_test_fp[i].c_str());
            bl_test_set[i] = ::cvCreateImage( cvSize(100,110),
IPL_DEPTH_8U, 1 );
            cvCvtColor( temp, bl_test_set[i], CV_BGR2GRAY );
            cvReleaseImage( &temp );
      }

      fprintf( outFile, "Done...\n");


      return 0;
}

      /**
	 * The method, LoadCATestSet, loads the Caucasian test set images
into
	 * the FaceDatabase for quick access.
	 *
	 * @param FaceReader fr -- the FaceReader class so that this
method can
	 *                                        access its methods
	 * @return int
	 */

int FaceDatabase::LoadCATestSet(FaceReader fr){

      int i;

      fprintf( outFile, "\nLoading CA Test Set \n");

      for(i = 0; i < NUM_CA_KNOWN; i++){
            IplImage* temp = ::cvLoadImage(fr.ca_test_fp[i].c_str());
            ca_test_set[i] = ::cvCreateImage( cvSize(100,110),
IPL_DEPTH_8U, 1 );
            cvCvtColor( temp, ca_test_set[i], CV_BGR2GRAY );
            cvReleaseImage( &temp );
      }

      fprintf( outFile, "Done...\n");

      return 0;
```

```
}

	/**
	 * The method, InsertImage, inserts a face into the FaceDatabase.
	 *
	 * @param IplImage* image -- the face in which to insert
	 * @return int
	 */

int FaceDatabase::InsertImage(IplImage* image){

	all_faces[image_index] = image;
	image_index++;

	return 0;
}

	/**
	 * The method, getImage, returns an IplImage* based on the
	 * index passed into it.
	 *
	 * @param int i -- index into the array to return an image
	 * @return IplImage* -- the face which is returned
	 */

IplImage* FaceDatabase::getImage(int i){
	if(i > image_index)
		return NULL;

	return all_faces[i];
}

	/**
	 * The method, SetTotalFaces, sets the total number of faces to
	 * be stored in the FaceDatabase.  It gets this number from the
	 * FaceReader since it is this class which knows how many faces
	 * have been read.
	 *
	 * @param FaceReader fr -- the FaceReader class
	 */

void FaceDatabase::SetTotalFaces(::FaceReader fr){
	total_faces = fr.GetNumFaces();
}

	/**
	 * The constructor of FaceDatabase opens the output file
	 * and sets some variables that are used within the
	 * class.
	 */

FaceDatabase::FaceDatabase()
{
	image_index = 0;
	num_aa_train = 30;
	num_aa_test = 30;
	num_ca_train = 30;
```

```
	num_ca_test = 30;

	ca_faces = 200;
	aa_faces = 200;

	if((outFile = fopen("output/database_out", "w")) == NULL)
		printf(" error creating file\n");

}

	/**
	 * The deconstructor of FaceDatabase cleans up memory allocation
	 * taken place in the class.
	 */

FaceDatabase::~FaceDatabase()
{
	int i;

//	for( i = 0; i < total_faces; i++){
//		::cvReleaseImage( &all_faces[i] );
//	}
}

/*********************************************************************
 * System: Face Classification Recognition System.
 *
 * Software for Thesis:
 *		NEUROCOGNITIVE INSPIRED HIERARCHICAL FACE RECOGNITION SYSTEM
 *
 * File name: FaceDatabase.h
 *
 * Input file name(s):  highgui.h
 *                                  cv.h
 *                                  stdio.h
 *                                  fstream.h
 *                                  string.h
 *                                  iostream.h
 *
 * Output file name(s):
 *
 * Author: Ryan Wilkins
 * Date Last Modified: 07/01/07
 * Purpose:
 *
 *********************************************************************/
#ifndef FACEDATABASE_H_
#define FACEDATABASE_H_


#include <highgui.h>
#include <cv.h>
#include <iostream>
#include <stdio.h>
#include <fstream>
#include <string>
#include <FaceReader.h>
```

```
#define NUM_AA_TRAINERS  30
#define NUM_CA_TRAINERS  30
#define NUM_BL_TRAINERS 30

#define NUM_AA_TEST  30
#define NUM_CA_TEST  30
#define NUM_BL_TEST 30

#define NUM_BL_KNOWN 30
#define NUM_AA_KNOWN  30
#define NUM_CA_KNOWN  30

class FaceDatabase
{
public:
      int InsertImages(::FaceReader fr);
      int InsertImage(IplImage* image);
      IplImage* getImage(int i);
      void SetTotalFaces(::FaceReader fr);
      int LoadAATrainingSet(FaceReader fr);
      int LoadCATrainingSet(FaceReader fr);
      int LoadBLTrainingSet(FaceReader fr);
      int LoadAAKnownSet(FaceReader fr);
      int LoadCAKnownSet(FaceReader fr);
      int LoadBLKnownSet(FaceReader fr);
      int LoadAATestSet(FaceReader fr);
      int LoadCATestSet(FaceReader fr);
      int LoadBLTestSet(FaceReader fr);


      FaceDatabase();
      virtual ~FaceDatabase();

      ::IplImage* all_faces[400];

      ::IplImage* aa_training_set[NUM_AA_TRAINERS];
      ::IplImage* ca_training_set[NUM_CA_TRAINERS];
      ::IplImage* bl_training_set[NUM_BL_TRAINERS];

      ::IplImage* aa_known_set[NUM_AA_KNOWN];
      ::IplImage* ca_known_set[NUM_CA_KNOWN];
      ::IplImage* bl_known_set[NUM_BL_KNOWN];

      ::IplImage* aa_test_set[NUM_AA_TEST];
      ::IplImage* ca_test_set[NUM_CA_TEST];
      ::IplImage* bl_test_set[NUM_BL_TEST];

      int image_index;
      int num_aa_train;
      int num_ca_train;
      int num_aa_test;
      int num_ca_test;
      int num_bl_test;
      int num_bl_train;

      int total_faces;
```

```
      int ca_faces;
      int aa_faces;

      FILE* outFile;
};

#endif /*FACEDATABASE_H_*/

/*************************************************************************
 * System: Face Classification Recognition System.
 *
 * Software for Thesis:
 *          NEUROCOGNITIVE INSPIRED HIERARCHICAL FACE RECOGNITION SYSTEM
 *
 * File name: FaceIdentifier.cpp
 *
 * Input file name(s):  FaceIdentifier.h
 *
 * Output file name(s): identifier_out
 *
 * Author: Ryan Wilkins
 * Date Last Modified: 07/01/07
 * Purpose: The level of the system where the face is identified
 *
 *************************************************************************/
#include "FaceIdentifier.h"



      FaceSpace* a_space;
      FaceSpace* c_space;
      FaceSpace* bl_space;
      FaceDatabase* dbase;

      /**
       * The constructor of FaceIdentifier sets key pointers to the
FaceSpace(s)
       * and the FaceDatabase.  These pointers allow this class to
access the
       * class methods of the parameters.
       *
       * @param FaceSpace* a -- pointer to the African American face
space
       * @param FaceSpace* c -- pointer to the Caucasian face space
       * @param FaceSpace* bl -- pointer to the Blended face space
       * @param FaceDatabase* d -- pointer to the FaceDatabase
       */

FaceIdentifier::FaceIdentifier(FaceSpace* a, FaceSpace* c, FaceSpace*
bl, FaceDatabase* d)
{
      a_space = a;
      c_space = c;
      bl_space = bl;
      dbase = d;

      if((outFile = fopen("output/identifier_out", "w")) == NULL)
```

```
            printf(" error creating file\n");

      fprintf( outFile, "\n***** Starting FaceIdentifier******\n" );
      fprintf( outFile, "%i Faces in the Caucasian Space\n", dbase-
>num_ca_train );
      fprintf( outFile, "%i Faces in the African American Space\n",
dbase->num_aa_train );


}

      /**
       * The deconstructor of FaceIdentifier is called when the system
exits and
       * cleans up memory allocation if needed.
       */

FaceIdentifier::~FaceIdentifier()
{

}

      /**
       * The method, Swap, is a general swap method that takes
       * two indices to swap and an integer array to do the swapping.
       *
       * @param int x -- the old location
       * @param int y -- the new location
       * @param int[] a -- the array in which the swap takes place
       */

void FaceIdentifier::Swap( int x, int y, int a[NUM_AA_TESTERS] ){
      int temp;
      temp = a[x];
      a[x] = a[y];
      a[y] = temp;
}

      /**
       * The method, Swap, is a general swap method that takes
       * two indices to swap and a pointer to an float array
       * to do the swapping.
       *
       * @param int x -- the old location
       * @param int y -- the new location
       * @param float* a -- pointer to the array in which the swap takes
place
       */

void FaceIdentifier::Swap( int x, int y, float* a ){
      float temp;
      temp = a[x];
      a[x] = a[y];
      a[y] = temp;
}

      /**
```

```
       * The method, Sort, takes three separate parameters in which two
       * arrays it sorts.  The float array is the distances of the
unknown
       * face from known faces and the int array are the indices or
ranks
       * of those faces.
       *
       * @param float[] fd -- distances of the unknown image from the
knowns
       * @param int[] faces -- are the rank of these images
       * @param int size -- is the size of these arrays
       */

void FaceIdentifier::Sort( float fd[NUM_AA_TESTERS], int
faces[NUM_AA_TESTERS], int size ){

      int i, j, min, first, temp;


      for( i = 0; i < size - 1; i++){
            min = i;
            for( j = i+1; j < size; j++){
                  if( fd[j] < fd[min] )
                        min = j;
            }
            Swap( i , min , fd);
            Swap( i, min, faces);
      }

}

      /**
       * The method, ProjectKnownImages, gets the face coefficients or
       * location in face space of the known images.  These coefficients
are
       * important when identifying a face sense it'll be the
coefficients
       * in which an unknown calculates its distance from.
       */

void FaceIdentifier::ProjectKnownImages(){
      int i;

      ::IplImage* probe = NULL;
      for(i = 0; i < NUM_AA_KNOWN; i++){
            probe = ::cvCloneImage( dbase->aa_known_set[i] );
            a_space->GetImageCoeffs( probe, known_aa_coeffs[i] );
      }

      for(i = 0; i < NUM_AA_KNOWN; i++){
            probe = ::cvCloneImage( dbase->ca_known_set[i] );
            c_space->GetImageCoeffs( probe, known_ca_coeffs[i] );
      }

      for(i = 0; i < NUM_BL_KNOWN; i++){
            probe = ::cvCloneImage( dbase->bl_known_set[i] );
            c_space->GetImageCoeffs( probe, known_bl_coeffs[i] );
```

```
	}

	::cvReleaseImage( &probe );


}

	/**
	 * The method, GetImageRankBL, returns the rank of a Blended
	 * face index.  The rank is important so that we know where the
younger
	 * face is away from the probe or corresponding older face.
	 *
	 * @param int i -- index in which the rank will be found
	 * @param int -- the rank of the index
	 */

int FaceIdentifier::GetImageRankBL( int i ){
	int j, index;
	if( i >= NUM_BL_TESTERS )
		return -1;

	for( j = 0; j < NUM_BL_TESTERS; j++ ){

		if( rank_blended[j] == i )
			index = j;

	}


	return index;
}

	/**
	 * The method, GetImageRankCA, returns the rank of a Caucasian
	 * face index.  The rank is important so that we know where the
younger
	 * face is away from the probe or corresponding older face.
	 *
	 * @param int i -- index in which the rank will be found
	 * @param int -- the rank of the index
	 */


int FaceIdentifier::GetImageRankCA( int i ){
	int j, index;
	if( i >= NUM_CA_TESTERS )
		return -1;

	for( j = 0; j < NUM_CA_TESTERS; j++ ){

		if( rank_ca[j] == i )
			index = j;

	}

	return index;
```

```
}

      /**
       * The method, GetImageRankAA, returns the rank of a African
American
       * face index.  The rank is important so that we know where the
younger
       * face is away from the probe or corresponding older face.
       *
       * @param int i -- index in which the rank will be found
       * @param int -- the rank of the index
       */


int FaceIdentifier::GetImageRankAA( int i ){
      int j, index;
      if( i >= NUM_AA_TESTERS )
            return -1;

      for( j = 0; j < NUM_AA_TESTERS; j++ ){

            if( rank_aa[j] == i )
                  index = j;
      }

      return index;
}

      /**
       * The method, RankImageInAA, ranks a given probe or unknown
       * image against the known images in African American face space.
       *
       * @param IplImage* probe -- the unknown image
       */

void FaceIdentifier::RankImageInAA( ::IplImage* probe )
{
      int i;

//    fprintf( outFile, "\n Ranking image \n" );
      ::IplImage* known = NULL;
      for( i = 0; i < NUM_AA_KNOWN; i++ ){
            known = ::cvCloneImage( dbase->aa_known_set[i]);
            rank_dist_aa[i] = a_space->FindMahalanobisDistance(probe ,
known);
//          rank_dist_aa[i] = a_space->FindDistance( probe , known );
            unrank_dist_aa[i] = rank_dist_aa[i];
            rank_aa[i] = i;
//          fprintf(outFile, "%i: %3.2f \n", rank_aa[i],
rank_dist_aa[i]);
      }

      Sort( rank_dist_aa, rank_aa, NUM_AA_KNOWN );


      fprintf( outFile, "\n Printing ranked results \n" );
```

```
	for( i = 0; i < NUM_AA_TRAINERS; i++ ){
		fprintf(outFile, "%i: %3.2f \n", rank_aa[i],
rank_dist_aa[i]);
	}

	::cvReleaseImage( &known );
	::cvReleaseImage( &probe );
}

	/**
	 * The method, RankImageInCA, ranks a given probe or unknown
	 * image against the known images in Caucasian face space.
	 *
	 * @param IplImage* probe -- the unknown image
	 */

void FaceIdentifier::RankImageInCA( ::IplImage* probe )
{
	int i;

	fprintf( outFile, "\n Ranking image \n" );
	::IplImage* known = NULL;
	for( i = 0; i < NUM_CA_KNOWN; i++ ){
		known = ::cvCloneImage( dbase->ca_known_set[i] );
		rank_dist_ca[i] = c_space->FindMahalanobisDistance( probe,
known );
//		rank_dist_ca[i] = c_space->FindDistance( probe , known );
		unrank_dist_ca[i] = rank_dist_ca[i];
		rank_ca[i] = i;
		fprintf(outFile, "%i: %3.2f \n", rank_ca[i],
rank_dist_ca[i]);
	}

	Sort( rank_dist_ca, rank_ca, NUM_AA_TRAINERS );


	fprintf( outFile, "\n Printing ranked results \n" );

	for( i = 0; i < NUM_AA_TRAINERS; i++ ){
		fprintf(outFile, "%i: %3.2f \n", rank_ca[i],
rank_dist_ca[i]);
	}

	::cvReleaseImage( &known );
	::cvReleaseImage( &probe );
}

	/**
	 * The method, IdentifyBlended, ranks an unknown face probe in
	 * the Blended space.  This is how the baseline method works where
it
	 * attempts to identify an unknown image against a set of Blended
	 * faces in Eigen space.
	 *
	 * @param IplImage* probe -- the unknown image
	 * @return int
```

```
       */

int FaceIdentifier::IdentifyBlended( ::IplImage* probe )
{
      int i;

      fprintf( outFile, "\n Ranking image \n" );
      ::IplImage* known = NULL;
      for( i = 0; i < NUM_BLENDED_TRAINERS; i++ ){
            known = ::cvCloneImage( dbase->bl_known_set[i] );
            rank_dist_bl[i] = bl_space->FindDistance( probe , known );
            unrank_dist_bl[i] = rank_dist_bl[i];
            rank_blended[i] = i;
            fprintf(outFile, "%i: %3.2f \n", rank_blended[i],
rank_dist_bl[i]);
      }

      Sort( rank_dist_ca, rank_ca, NUM_AA_TRAINERS );

      fprintf( outFile, "\n Sorted \n" );
      for( i = 0; i < NUM_BLENDED_TRAINERS; i++ ){
            fprintf(outFile, "%i: %3.2f \n", rank_blended[i],
rank_dist_bl[i]);
      }

      ::cvReleaseImage( &known );
      ::cvReleaseImage( &probe );

      return 0;
}

      /**
       * The method, Identify, classifies an unknown probe face first
and
       * then ranks that unknown image in a corresponding face space.
       * It has a choice of 4 differenct classification methods that
have been
       * tested and evaluated.
       *
       * @param IplImage* probe -- the unknown image
       * @param FaceClassifier* fc -- a pointer to the FaceClassifier so
that methods
       *                                                  can be accessed
       */

int FaceIdentifier::Identify( ::IplImage* probe , FaceClassifier* fc)
{

//    int face_class = fc->ClassifyImageMahalanobisVote(probe, 5,
false);
//    int face_class = fc->ClassifyImageMahalanobis( probe );
//    int face_class = fc->ClassifyImage( probe );
      int face_class = fc->ClassifyImageWithRange( probe );

      switch(face_class){

      case CA:
```

```
            RankImageInCA( probe );
      break;

      case AA:
            RankImageInAA( probe );
      break;

      case BOTH:

            fprintf(outFile, "\n Unable to classify image. Image belongs
to both sets.\n");
      break;

      case UNKNOWN:
            fprintf(outFile, "\n Unable to classify image. Image does
not belong to any sets.\n");
      break;

      }

      return face_class;

}

      /**
       * The method, PrintKnownCoeffs, prints the known coefficients to
an output
       * file which is important for evaluation.
       *
       */

void FaceIdentifier::PrintKnownCoeffs(){
      int i, j;

      fprintf( outFile, "\nKnown AA Coefficients\n" );
      for( i = 0; i < NUM_AA_KNOWN; i++ ){
            for( j = 0; j < NUM_EIGENS; j++ ){
                  fprintf( outFile, "%3.2f, ", known_aa_coeffs[i][j] );

            }
            fprintf( outFile, "\n" );
      }

      fprintf( outFile, "\nKnown CA Coefficients\n" );
      for( i = 0; i < NUM_CA_KNOWN; i++ ){
            for( j = 0; j < NUM_EIGENS; j++ ){
                  fprintf( outFile, "%3.2f, ", known_ca_coeffs[i][j] );

            }
            fprintf( outFile, "\n" );
      }

      fprintf( outFile, "\nKnown BL Coefficients\n" );
      for( i = 0; i < NUM_BL_KNOWN; i++ ){
            for( j = 0; j < NUM_EIGENS; j++ ){
                  fprintf( outFile, "%3.2f, ", known_bl_coeffs[i][j] );
```

```
            }
            fprintf( outFile, "\n" );
        }


}


/*****************************************************************
 * System: Face Classification Recognition System.
 *
 * Software for Thesis:
 *          NEUROCOGNITIVE INSPIRED HIERARCHICAL FACE RECOGNITION SYSTEM
 *
 * File name: FaceIdentifier.h
 *
 * Input file name(s):  highgui.h
 *                                  cv.h
 *                                  stdio.h
 *                                  fstream.h
 *                                  cstring.h
 *                                  string.h
 *
 * Output file name(s):
 *
 * Author: Ryan Wilkins
 * Date Last Modified: 07/01/07
 * Purpose:
 *
 *****************************************************************/
#ifndef FACEIDENTIFIER_H_
#define FACEIDENTIFIER_H_

#include <highgui.h>
#include <cv.h>
#include <iostream>
#include <stdio.h>
#include <fstream>
#include <FaceSpace.h>
#include <FaceDatabase.h>
#include <FaceReader.h>
#include <FaceClassifier.h>
#include <cstring>
#include <string.h>

#define NUM_CA_TESTERS 30
#define NUM_AA_TESTERS 30
#define NUM_BL_TESTERS 30

#define NUM_AA_TRAINERS 30
#define NUM_CA_TRAINERS 30
#define NUM_BLENDED_TRAINERS 30

#define NUM_AA_KNOWN  30
#define NUM_CA_KNOWN  30
#define NUM_BL_KNOWN  30
```

```
#define NUM_EIGENS 16

#define CA 0
#define AA 1
#define BOTH 3
#define UNKNOWN 4

class FaceIdentifier
{

      public:

            FaceIdentifier(FaceSpace* a, FaceSpace* c, FaceSpace* bl_fs,
FaceDatabase* d);
            virtual ~FaceIdentifier();

            float rank_dist_ca[NUM_AA_TRAINERS];
            float rank_dist_aa[NUM_CA_TRAINERS];
            float rank_dist_bl[NUM_BLENDED_TRAINERS];

            float unrank_dist_ca[NUM_AA_TRAINERS];
            float unrank_dist_aa[NUM_CA_TRAINERS];
            float unrank_dist_bl[NUM_BLENDED_TRAINERS];

            int rank_aa[NUM_AA_TRAINERS];
            int rank_ca[NUM_CA_TRAINERS];
            int rank_blended[NUM_BLENDED_TRAINERS];

            float known_aa_coeffs[NUM_AA_TRAINERS][NUM_EIGENS];
            float known_ca_coeffs[NUM_CA_TRAINERS][NUM_EIGENS];
            float known_bl_coeffs[NUM_BL_TRAINERS][NUM_EIGENS];

            FILE* outFile;

            void Swap( int x, int y, int a[NUM_AA_TESTERS] );
            void Swap( int x, int y, float* a );
            void Sort( float fd[NUM_AA_TESTERS], int
faces[NUM_AA_TESTERS], int size );
            void RankImageInAA( ::IplImage* probe );
            void RankImageInCA( ::IplImage* probe );
            int Identify( ::IplImage* probe , FaceClassifier* fc);
            int IdentifyBlended( ::IplImage* probe );
            void ProjectKnownImages();
            void PrintKnownCoeffs();
            int GetImageRankAA( int i );
            int GetImageRankCA( int i );
            int GetImageRankBL( int i );

};

#endif /*FACEIDENTIFIER_H_*/

/*****************************************************************
 * System: Face Classification Recognition System.
 *
 * Software for Thesis:
 *          NEUROCOGNITIVE INSPIRED HIERARCHICAL FACE RECOGNITION SYSTEM
```

```
 *
 * File name: FaceReader.cpp
 *
 * Input file name(s):  FaceReader.h
 *                                  aa_faces.fpl
 *                                  aa_known.fpl
 *                                  aa_test.fpl
 *                                  aa_spec_test.fpl
 *                                  bl_known.fpl
 *                                  bl_test.fpl
 *                                  bl_train.fpl
 *                                  ca_faces.fpl
 *                                  ca_known.fpl
 *                                  ca_test.fpl
 *                                  ca_train.fpl
 *                                  facelist.fpl
 *
 * Output file name(s): reader_out
 *
 * Author: Ryan Wilkins
 * Date Last Modified: 07/01/07
 * Purpose: The class that reads in face path lists and allocates
 *                face images
 *
 ***************************************************************************/

#include "FaceReader.h"

using namespace std;

      int NUM_OF_FACES = 400;
      int FILE_NAME_LENGTH = 75;
      int NUM_OF_AA = 200;
      int NUM_OF_CA = 200;

      /**
       * The method, ReadSpecFile, reads from the spec file that was
used in the
       * presentation demo.  It allocates those specific file paths
intow arrays
       * for later use in the demo.
       *
       * @return int
       */

//int FaceReader::ReadSpecFile()
//{
//
//    string file_name( "facelists/aa_spec_test.fpl" );
//    // image path should be set at 1 more than the number
//    // of characters in a filepath
//    char image_path[100];
//    int index = 0;
//    int count = 0;
//
//    fprintf(outFile, "Reading file. %s \n", file_name.c_str());
//
```

```
//    fstream face_file( file_name.c_str(), ios::in );
//
//    if( face_file.is_open() ){
//
//          fprintf(outFile, "File is open \n");
//
//          while( index < NUM_AA_SPECS ){
//          //    cout << "\ni: " << index;
//
//                face_file.getline( image_path, FILE_NAME_LENGTH);

//
//                if( count == 0 ){
//                      fprintf (outFile, "\n%s Images\n", image_path );
//                      count++;
//                }else{
//                      aa_spec_fp[index] = image_path;
//
////                    aa_spec_fn[index] =
aa_spec_fp[index].substr(55);
//
////                    fprintf( outFile, "%s\n", aa_spec_fp[index] );
//                      index++;
//                }
//
//
//          }
//          fprintf(outFile, "End of file \n");
//    }
//
//
//
//
//    face_file.close();
//
//    return 0;
//
//}


      /**
       * The method, ReadCAFile, reads the master file of Caucasian
faces
       * and allocates file paths and file names into arrays.
       *
       * @return int
       */

int FaceReader::ReadCAFile()
{

      string file_name( "facelists/ca_faces.fpl" );
      // image path should be set at 1 more than the number
      // of characters in a filepath
      char image_path[100];
      int index = 0;
      int count = 0;
```

```
	fprintf(outFile, "Reading file. %s \n", file_name.c_str());

	fstream face_file( file_name.c_str(), ios::in );

	if( face_file.is_open() ){

		fprintf(outFile, "File is open \n");

		while( index < NUM_CA_FACES ){
		//    cout << "\ni: " << index;

			face_file.getline( image_path, FILE_NAME_LENGTH);


			if( count == 0 ){
				fprintf (outFile, "\n%s Images\n", image_path );
				count++;
			}else{
				ca_train_paths[index] = image_path;
				if(ca_train_paths[index].length() < 58 )
					fprintf( outFile, "\nError:: Bad String
Transfer(%s) \n", ca_train_paths[index].c_str() );
				ca_file_names[index] =
ca_train_paths[index].substr(58);
				if(ca_train_paths[index].length() < 58 )
					fprintf( outFile, "\nError:: Bad String
Transfer(%s) \n", ca_train_paths[index].c_str() );

				fprintf( outFile, "%s\n",
ca_train_paths[index].c_str() );
				index++;
			}


		}
		fprintf(outFile, "End of file \n");
	}




	face_file.close();

	return 0;
}

	/**
	 * The method, ReadAAFile, reads the master file of African
American faces
	 * and allocates file paths and file names into arrays.
	 *
	 * @return int
	 */

int FaceReader::ReadAAFile()
{
```

```
	string file_name( "facelists/aa_faces.fpl" );
	// image path should be set at 1 more than the number
	// of characters in a filepath
	char image_path[100];
	int index = 0;
	int count = 0;

	fprintf(outFile, "Reading file. %s \n", file_name.c_str());

	fstream face_file( file_name.c_str(), ios::in );

	if( face_file.is_open() ){

		fprintf(outFile, "File is open \n");

		while( index < NUM_AA_FACES ){
		//    cout << "\ni: " << index;


			face_file.getline( image_path, FILE_NAME_LENGTH);


			if( count == 0 ){
				fprintf (outFile, "\n%s Images", image_path )  ;
				count++;
			}else{
				aa_train_paths[index] = image_path;
				aa_file_names[index] =
aa_train_paths[index].substr(58);

				fprintf( outFile, "\n%s",
aa_train_paths[index].c_str()  );

				index++;
			}


		}
		fprintf(outFile, "End of file \n");
	}




	face_file.close();

	return 0;
}

	/**
	 * The method, Read_File, reads the master file of all the faces
	 * and allocates file paths and file names into arrays.
	 *
	 * @return int
	 */
```

```
int FaceReader::Read_File()
{
	string file_name( "facelists/facelist.fpl" );
	// image path should be set at 1 more than the number
	// of characters in a filepath
	char image_path[100];
	int index = 0;
	int count = 0;

	fprintf(outFile, "Reading file. %s \n", file_name.c_str());

	fstream face_file( file_name.c_str(), ios::in );

	if( face_file.is_open() ){

		fprintf(outFile, "File is open \n");
		int ca_i = 0;
		while( index < NUM_OF_FACES ){
		//    cout << "\ni: " << index;

			face_file.getline( image_path, FILE_NAME_LENGTH);



			if(count >= 1 && index < NUM_OF_AA){
				aa_train_paths[index] = image_path;
				aa_file_names[index] =
aa_train_paths[index].substr(58);
				index++;
				//fprintf(outFile, "AA(%i): %s\n", index-1,
aa_train_paths[index-1].c_str());
			}else if(index >= NUM_OF_AA){
				ca_i = index - NUM_OF_AA;
				ca_train_paths[ca_i] = image_path;

				ca_file_names[ca_i] =
ca_train_paths[ca_i].substr(58);
				//fprintf(outFile, "CA(%i): %s\n", ca_i,
ca_train_paths[ca_i].c_str());
				index++;
			}

			count++;

		}
		fprintf(outFile, "End of file \n");
	}


	face_file.close();

	return (0);
}

	/**
	 * The method, ReadCAKnownFile, reads from the Caucasian
	 * input file of known face file paths.  It then sets the file
```

```
       * paths and file names into arrays.
       *
       * @return int
       */

int FaceReader::ReadCAKnownFile(){
      string file_name( "facelists/ca_known.fpl" );
      string dir(
"/home/rbw9908/Documents/thesis/trainingsets/ts1/masked/ca/" );
      // image path should be set at 1 more than the number
      // of characters in a filepath
      char image_path[100];
      int index = 0;
      int count = 0;

      fprintf(outFile, "Reading file. %s \n", file_name.c_str());

      fstream face_file( file_name.c_str(), ios::in );

      if( face_file.is_open() ){

            fprintf(outFile, "File is open \n");
            while( index < NUM_CA_KNOWN ){

                  face_file.getline( image_path, FILE_NAME_LENGTH);


                  if(count >= 1){
                        ca_known_fp[index] = image_path;
                        if( ca_known_fp[index].length() > 58 ){
                              ca_known_fn[index] =
ca_known_fp[index].substr(58);
                        }else{
                              ca_known_fn[index] = ca_known_fp[index];
                              fprintf( outFile ,"\n***ERROR: File
path(%s) %i chars ****\n" ,ca_known_fp[index].c_str(),
ca_known_fp[index].length());

                              ca_known_fp[index] = dir.append(
ca_known_fp[index].c_str() );

                              fprintf( outFile , "Appended %s
",ca_known_fp[index].c_str() );

                        }
//                      fprintf(outFile, "%i: %s\n", index,
ca_known_fp[index].c_str());
                        index++;
                  }

                  count++;

            }
            fprintf(outFile, "End of file \n");
      }
```

```
	face_file.close();

	return 0;
}

	/**
	 * The method, ReadBLKnownFile, reads from the Blended
	 * input file of known face file paths.  It then sets the file
	 * paths and file names into arrays.
	 *
	 * @return int
	 */

int FaceReader::ReadBLKnownFile(){
	string file_name( "facelists/bl_known.fpl" );
	// image path should be set at 1 more than the number
	// of characters in a filepath
	char image_path[100];
	int index = 0;
	int count = 0;

	fprintf(outFile, "Reading file. %s \n", file_name.c_str());

	fstream face_file( file_name.c_str(), ios::in );

	if( face_file.is_open() ){

		fprintf(outFile, "File is open \n");
		while( index < NUM_BL_KNOWN ){

			face_file.getline( image_path, FILE_NAME_LENGTH);


			if(count >= 1){
				bl_known_fp[index] = image_path;
				if( bl_known_fp[index].length() > 58 )
					bl_known_fn[index] =
bl_known_fp[index].substr(58);
//				fprintf(outFile, "%i: %s\n", index,
bl_known_fp[index].c_str());
				index++;
			}

			count++;

		}
		fprintf(outFile, "End of file \n");
	}


	face_file.close();

	return 0;
}

	/**
	 * The method, ReadAAKnownFile, reads from the African American
```

```
       * input file of known face file paths.  It then sets the file
       * paths and file names into arrays.
       *
       * @return int
       */

int FaceReader::ReadAAKnownFile(){
      string file_name( "facelists/aa_known.fpl" );
      // image path should be set at 1 more than the number
      // of characters in a filepath
      char image_path[100];
      int index = 0;
      int count = 0;

      fprintf(outFile, "Reading file. %s \n", file_name.c_str());

      fstream face_file( file_name.c_str(), ios::in );

      if( face_file.is_open() ){

            fprintf(outFile, "File is open \n");
            while( index < NUM_AA_KNOWN ){

                  face_file.getline( image_path, FILE_NAME_LENGTH);


                  if(count >= 1){
                        aa_known_fp[index] = image_path;
                        if( aa_known_fp[index].length() > 58 )

                              aa_known_fn[index] =
aa_known_fp[index].substr(58);
//                                          fprintf(outFile, "%i:
%s\n", index, aa_known_fp[index].c_str());
                        index++;
                  }

                  count++;

            }
            fprintf(outFile, "End of file \n");
      }


      face_file.close();

      return 0;
}

      /**
       * The method, ReadCATestFile, reads from the Caucasian
       * input file of testing face file paths.  It then sets the file
       * paths and file names into arrays.
       *
       * @return int
       */
```

```
int FaceReader::ReadCATestFile(){
      string file_name( "facelists/ca_test.fpl" );
      string dir(
"/home/rbw9908/Documents/thesis/trainingsets/ts1/masked/ca/" );
      // image path should be set at 1 more than the number
      // of characters in a filepath
      char image_path[100];
      int index = 0;
      int count = 0;

      fprintf(outFile, "Reading file. %s \n", file_name.c_str());

      fstream face_file( file_name.c_str(), ios::in );

      if( face_file.is_open() ){

            fprintf(outFile, "File is open \n");
            while( index < NUM_CA_TEST ){

                  face_file.getline( image_path, FILE_NAME_LENGTH);


                  if(count >= 1){
                        ca_test_fp[index] = image_path;
                        if( ca_test_fp[index].length() > 58 ){
                              ca_test_fn[index] =
ca_test_fp[index].substr(58);
                        } else {
                              ca_test_fn[index] = ca_test_fp[index];
                              fprintf( outFile ,"\n***ERROR: File
path(%s) %i chars ****\n" ,
ca_test_fp[index].c_str(),ca_test_fp[index].length());
                              ca_test_fp[index] = dir.append(
ca_test_fp[index] );

                              fprintf( outFile , "Appended %s ",
ca_test_fp[index].c_str() );
                        }
//                      fprintf(outFile, "%i: %s\n", index,
ca_test_fp[index].c_str());
                        index++;
                  }

                  count++;

            }
            fprintf(outFile, "End of file \n");
      }


      face_file.close();

      return 0;
}

      /**
       * The method, ReadBLTestFile, reads from the Blended
```

```
	 * input file of testing face file paths.  It then sets the file
	 * paths and file names into arrays.
	 *
	 * @return int
	 */

int FaceReader::ReadBLTestFile(){
	string file_name( "facelists/bl_test.fpl" );
	// image path should be set at 1 more than the number
	// of characters in a filepath
	char image_path[100];
	int index = 0;
	int count = 0;

	fprintf(outFile, "Reading file. %s \n", file_name.c_str());

	fstream face_file( file_name.c_str(), ios::in );

	if( face_file.is_open() ){

		fprintf(outFile, "File is open \n");
		while( index < NUM_BL_TEST ){

			face_file.getline( image_path, FILE_NAME_LENGTH);


			if(count >= 1){
				bl_test_fp[index] = image_path;
				if( bl_test_fp[index].length() > 58 );
					bl_test_fn[index] =
bl_test_fp[index].substr(58);
//				fprintf(outFile, "%i: %s\n", index,
bl_test_fp[index].c_str());
				index++;
			}

			count++;

		}
		fprintf(outFile, "End of file \n");
	}


	face_file.close();

	return 0;
}

	/**
	 * The method, ReadAATestFile, reads from the African American
	 * input file of testing file paths.  It then sets the file
	 * paths and file names into arrays.
	 *
	 * @return int
	 */

int FaceReader::ReadAATestFile(){
```

```
	string file_name( "facelists/aa_test.fpl" );
	// image path should be set at 1 more than the number
	// of characters in a filepath
	char image_path[100];
	int index = 0;
	int count = 0;

	fprintf(outFile, "Reading file. %s \n", file_name.c_str());

	fstream face_file( file_name.c_str(), ios::in );

	if( face_file.is_open() ){

		fprintf(outFile, "File is open \n");
		while( index < NUM_AA_TEST ){

			face_file.getline( image_path, FILE_NAME_LENGTH);


			if(count >= 1){
				aa_test_fp[index] = image_path;
				if( aa_test_fp[index].length() > 58 )
					aa_test_fn[index] =
aa_test_fp[index].substr(58);
//				fprintf(outFile, "%i: %s\n", index,
aa_test_fp[index].c_str());
				index++;
			}

			count++;

		}
		fprintf(outFile, "End of file \n");
	}


	face_file.close();

	return 0;
}

	/**
	 * The method, ReadCATrainFile, reads from the Caucasian
	 * input file of training face file paths.  It then sets the file
	 * paths and file names into arrays.
	 *
	 * @return int
	 */

int FaceReader::ReadCATrainFile(){
	string file_name( "facelists/ca_train.fpl" );
	string dir(
"/home/rbw9908/Documents/thesis/trainingsets/ts1/masked/ca/" );
	// image path should be set at 1 more than the number
	// of characters in a filepath
	char image_path[100];
	int index = 0;
```

```
	int count = 0;

	fprintf(outFile, "Reading file. %s \n", file_name.c_str());

	fstream face_file( file_name.c_str(), ios::in );

	if( face_file.is_open() ){

		fprintf(outFile, "File is open \n");
		while( index < NUM_CA_TRAINERS ){

			face_file.getline( image_path, FILE_NAME_LENGTH);


			if(count >= 1){
				ca_train_fp[index] = image_path;
				if( ca_train_fp[index].length() > 58 ){
					ca_train_fn[index] =
ca_train_fp[index].substr(58);
				}else{
					ca_test_fn[index] = ca_test_fp[index];
					fprintf( outFile ,"\n***ERROR: File
path(%s) %i chars ****\n" , ca_train_fp[index].c_str(),
ca_train_fp[index].length());
					ca_train_fp[index] = dir.append(
ca_train_fp[index] );
					fprintf( outFile , "Appended %s ",
ca_train_fp[index].c_str() );

				}
//				fprintf(outFile, "%i: %s\n", index,
ca_train_fp[index].c_str());
				index++;
			}

			count++;

		}
		fprintf(outFile, "End of file \n");
	}

	face_file.close();

	return 0;
}

	/**
	 * The method, ReadBLTrainFile, reads from the Blended
	 * input file of training face file paths.  It then sets the file
	 * paths and file names into arrays.
	 *
	 * @return int
	 */

int FaceReader::ReadBLTrainFile(){
	string file_name( "facelists/bl_train.fpl" );
	// image path should be set at 1 more than the number
```

```
	// of characters in a filepath
	char image_path[100];
	int index = 0;
	int count = 0;

	fprintf(outFile, "Reading file. %s \n", file_name.c_str());

	fstream face_file( file_name.c_str(), ios::in );

	if( face_file.is_open() ){

		fprintf(outFile, "File is open \n");
		while( index < NUM_BL_TRAINERS ){

			face_file.getline( image_path, FILE_NAME_LENGTH);


			if(count >= 1){
				bl_train_fp[index] = image_path;
				if( bl_train_fp[index].length() > 58 )
					bl_train_fn[index] =
bl_train_fp[index].substr(58);
//				fprintf(outFile, "%i: %s\n", index,
bl_train_fp[index].c_str());
				index++;
			}

			count++;

		}
		fprintf(outFile, "End of file \n");
	}

	face_file.close();

	return 0;
}

	/**
	 * The method, ReadAATrainFile, reads from the African American
	 * input file of training face file paths.  It then sets the file
	 * paths and file names into arrays.
	 *
	 * @return int
	 */

int FaceReader::ReadAATrainFile(){
	string file_name( "facelists/aa_train.fpl" );
	// image path should be set at 1 more than the number
	// of characters in a filepath
	char image_path[100];
	int index = 0;
	int count = 0;

	fprintf(outFile, "Reading file. %s \n", file_name.c_str());

	fstream face_file( file_name.c_str(), ios::in );
```

```
	if( face_file.is_open() ){

		fprintf(outFile, "File is open \n");
		while( index < NUM_AA_TRAINERS ){

			face_file.getline( image_path, FILE_NAME_LENGTH);


			if(count >= 1){
				aa_train_fp[index] = image_path;
				if( aa_train_fp[index].length() > 58 )
					aa_train_fn[index] =
aa_train_fp[index].substr(58);
				fprintf(outFile, "%i: %s\n", index,
aa_train_fp[index].c_str());
				index++;
			}

			count++;

		}
		fprintf(outFile, "End of file \n");
	}
	face_file.close();

	return 0;
}

	/**
	 * The method, AddCAKnownIndex, adds a given index at
	 * a location.
	 *
	 * @param int n -- the index to add
	 * @param int i -- the location where to add the index into the
array
	 */

int FaceReader::AddCAKnownIndex(int n, int i){

	if( i >= NUM_CA_KNOWN )
		return -1;

	ca_known_indices[i] = n;

	return 0;

}

	/**
	 * The method, AddCATrainIndex, adds a given index at
	 * a location.
	 *
	 * @param int n -- the index to add
	 * @param int i -- the location where to add the index into the
array
	 */
```

```
int FaceReader::AddCATrainIndex(int n, int i)
{
	if ( i >= NUM_CA_TRAINERS )
		return -1;

	ca_train_indices[i] = n;

	return 0;
}

	/**
	 * The method, InCATrainSet, decides whether or not an index
	 * is in the set of known Caucasian indices.
	 *
	 * @param int n -- index in question
	 * @param bool -- true or false the index is in the set
	 */

bool FaceReader::InCATrainSet( int n )
{
	int i;
	bool inSet = false;

	for( i = 0; i < NUM_CA_TRAINERS; i++){
		if( ca_train_indices[i] == n ){
			inSet = true;
			break;
		}
	}

	return inSet;
}

	/**
	 * The method, InCAKnownSet, decides whether or not an index
	 * is in the set of known Caucasian indices.
	 *
	 * @param int n -- index in question
	 * @param bool -- true or false the index is in the set
	 */

bool FaceReader::InCAKnownSet( int n)
{
	int i;
	bool inSet = false;

	for( i = 0; i < NUM_CA_TRAINERS; i++){
		if( ca_known_indices[i] == n ){
			inSet = true;
			break;
		}
	}

	return inSet;
}
```

```
	/**
	 * The method, AddAAKnownIndex, adds a given index at
	 * a location.
	 *
	 * @param int n -- the index to add
	 * @param int i -- the location where to add the index into the
array
	 */


int FaceReader::AddAAKnownIndex(int n, int i){

	if( i >= NUM_AA_KNOWN )
		return -1;

	aa_known_indices[i] = n;

	return 0;

}

	/**
	 * The method, AddAATrainIndex, adds a given index at
	 * a location.
	 *
	 * @param int n -- the index to add
	 * @param int i -- the location where to add the index into the
array
	 */

int FaceReader::AddAATrainIndex(int n, int i)
{
	if ( i >= NUM_AA_TRAINERS )
		return -1;

	aa_train_indices[i] = n;

	return 0;
}

	/**
	 * The method, InAATrainSet, decides whether or not an index
	 * is in the set of known African American indices.
	 *
	 * @param int n -- index in question
	 * @param bool -- true or false the index is in the set
	 */

bool FaceReader::InAATrainSet( int n )
{
	int i;
	bool inSet = false;

	for( i = 0; i < NUM_AA_TRAINERS; i++){
		if( aa_train_indices[i] == n ){
			inSet = true;
			break;
```

```
            }
      }

      return inSet;
}

      /**
       * The method, InAAKnownSet, decides whether or not an index
       * is in the set of known African American indices.
       *
       * @param int n -- index in question
       * @param bool -- true or false the index is in the set
       */

bool FaceReader::InAAKnownSet( int n )
{
      int i;
      bool inSet = false;

      for( i = 0; i < NUM_AA_TRAINERS; i++){
            if( aa_known_indices[i] == n ){
                  inSet = true;
                  break;
            }
      }

      return inSet;
}

      /**
       * The method, AddBLKnownIndex, adds a given index and race char
       * at a location.
       *
       * @param int n -- index to add to the set
       * @param int i -- location in the array to add the index
       * @param char r -- the race of the face index
       * @return int
       */

int FaceReader::AddBLKnownIndex(int n, int i, char r){

      if( i >= NUM_BL_KNOWN )
            return -1;

      bl_known_indices[i][0] = n;
      bl_known_indices[i][1] = r;

      return 0;

}

      /**
       * The method, AddBLTrainIndex, adds a given index and race char
       * at a location.
       *
       * @param int n -- index to add to the set
       * @param int i -- location in the array to add the index
```

```
       * @param char r -- the race of the face index
       * @return int
       */


int FaceReader::AddBLTrainIndex(int n, int i, char r)
{
      if ( i >= NUM_BL_TRAINERS )
            return -1;

      bl_train_indices[i][0] = n;
      bl_train_indices[i][1] = r;

      return 0;
}

      /**
       * The method, InBLTrainSet, decides whether or not face index
       * is in the set of train face indices given an index and
character
       * that represents a race.
       *
       * @param int n -- index in question
       * @param char r -- the race of the face index in question
       * @return bool -- true or false that the index is in the set
       */

bool FaceReader::InBLTrainSet( int n, char r )
{
      int i;
      bool inSet = false;

      for( i = 0; i < NUM_BL_TRAINERS; i++){
            if( bl_train_indices[i][0] == n && bl_train_indices[i][1] ==
r ){
                  inSet = true;
                  break;
            }
      }

      return inSet;
}

      /**
       * The method, InBLKnownSet, decides whether or not face index
       * is in the set of known face indices given an index and
character
       * that represents a race.
       *
       * @param int n -- index in question
       * @param char r -- the race of the face index in question
       * @return bool -- true or false that the index is in the set
       */

bool FaceReader::InBLKnownSet( int n, char r )
{
      int i;
```

```
	bool inSet = false;

	for( i = 0; i < NUM_BL_TRAINERS; i++){
		if( bl_known_indices[i][0] == n && bl_known_indices[i][1] ==
r ){
			inSet = true;
			break;
		}
	}

	return inSet;
}

	/**
	 * The method, CreateAATrainingSet, chooses faces from the
	 * total set of African American faces and prints their file
	 * paths out to file to be read in later.  The faces in the
	 * African American training set are unique from those in the
known
	 * and testing sets.
	 *
	 * @return int
	 */

int FaceReader::CreateAATrainingSet()
{
	int i, r;

	FILE* aa_train_file;

	if( (aa_train_file = fopen( "facelists/aa_train.fpl", "w" )) ==
NULL )
		printf(" error creating file\n");

	::srand( time(NULL) + getpid() );

	fprintf( aa_train_file, "%i\n", NUM_AA_TRAINERS );

	for( i = 0; i < NUM_AA_TRAINERS; i++ ){

		r = (int)((((double)::rand() / (double)RAND_MAX) *
NUM_OF_AA)+0.5);

		AddAATrainIndex( r , i );

		if( i == NUM_AA_TRAINERS - 1 )
			fprintf( aa_train_file, "%s",
aa_train_paths[r].c_str() );
		else
			fprintf( aa_train_file, "%s\n",
aa_train_paths[r].c_str() );
	}

	fclose( aa_train_file );

	return (0);
}
```

```
      /**
       * The method, CreateAATestingSet, chooses African American faces
       * from the total set of African American faces then prints the
file
       * paths of the faces out to file to be read in later.  It first
makes
       * sure that testing faces are the second image or older image of
a face
       * in the known set.
       *
       * @return int
       */

int FaceReader::CreateAATestingSet()
{
      int i, r;

      FILE* aa_test_file;

      if( (aa_test_file = fopen( "facelists/aa_test.fpl", "w" )) == NULL
)
            printf(" error creating file\n");

      ::srand( time(NULL) + getpid() );

      fprintf( aa_test_file, "%i\n", NUM_AA_TEST );

      for( i = 0; i < NUM_AA_TEST; i++ ){
            if( i == NUM_AA_TEST - 1)
                  fprintf( aa_test_file , "%s",
aa_train_paths[aa_known_indices[i]+1].c_str() );
            else
                  fprintf( aa_test_file , "%s\n",
aa_train_paths[aa_known_indices[i]+1].c_str() );

      }

      fclose( aa_test_file );

      return (0);
}

      /**
       * The method, CreateAAKnownSet, chooses African American faces
       * from the total set of AA faces and prints the file
       * paths of these faces out to a file to be read later.  The known
       * set of faces are unique from the training set.
       *
       * @return int
       */

int FaceReader::CreateAAKnownSet()
{
      int i, r;

      FILE* aa_known_file;
```

```
      if( (aa_known_file = fopen( "facelists/aa_known.fpl", "w" )) ==
NULL )
            printf(" error creating file\n");

      ::srand( time(NULL) + getpid() );

      fprintf( aa_known_file, "%i\n", NUM_AA_KNOWN );

      for( i = 0; i < NUM_AA_KNOWN; i++ ){

            r = (int)((((double)::rand() / (double)RAND_MAX) *
NUM_OF_AA)+0.5);

            if( r % 2 == 0 ){ // a younger face

                  if( InAATrainSet(r) || InAATrainSet(r+1) ||
InAAKnownSet(r) || InAAKnownSet(r+1)){   // in the training set
                        i--;
                  }else{
                        if( i == NUM_AA_KNOWN - 1)
                              fprintf( aa_known_file, "%s",
aa_train_paths[r].c_str() );
                        else
                              fprintf( aa_known_file, "%s\n",
aa_train_paths[r].c_str() );

                        AddAAKnownIndex( r, i );
                  }

            }else{
                  if( InAATrainSet(r) || InAATrainSet(r-1) ||
InAAKnownSet(r) || InAAKnownSet(r-1) ){
                        i--;
                  }else{
                        if( i == NUM_AA_KNOWN - 1)
                              fprintf( aa_known_file, "%s",
aa_train_paths[r-1].c_str() );
                        else
                              fprintf( aa_known_file, "%s\n",
aa_train_paths[r-1].c_str() );

                        AddAAKnownIndex(r-1 , i);

                  }
            }


      }


      fclose( aa_known_file );

      return (0);
}

      /**
```

```
       * The method, CreateCATrainingSet, chooses faces from the
       * total set of Caucasian faces and prints their file
       * paths out to file to be read in later.  The faces in the
       * Caucasian training set are unique from those in the known
       * and testing sets.
       *
       * @return int
       */

int FaceReader::CreateCATrainingSet()
{
      int i, r;

      FILE* ca_train_file;

      if( (ca_train_file = fopen( "facelists/ca_train.fpl", "w" )) ==
NULL )
            printf(" error creating file\n");

      ::srand( time(NULL) + getpid() );

      fprintf( ca_train_file, "%i\n", NUM_CA_TRAINERS );

      for( i = 0; i < NUM_CA_TRAINERS; i++ ){

            r = (int)((((double)::rand() / (double)RAND_MAX) *
NUM_OF_CA)+0.5);

            AddCATrainIndex( r , i );

            if( i == NUM_CA_TRAINERS - 1 )
                  fprintf( ca_train_file, "%s",
ca_train_paths[r].c_str() );
            else
                  fprintf( ca_train_file, "%s\n",
ca_train_paths[r].c_str() );
      }

      fclose( ca_train_file );

      return (0);
}

      /**
       * The method, CreateCATestingSet, chooses Caucasian faces
       * from the total set of Caucasian faces then prints the file
       * paths of the faces out to file to be read in later.  It first
makes
       * sure that testing faces are the second image or older image of
a face
       * in the known set.
       *
       * @return int
       */

int FaceReader::CreateCATestingSet()
{
```

```
      int i, r;

      FILE* ca_test_file;

      if( (ca_test_file = fopen( "facelists/ca_test.fpl", "w" )) == NULL
)
            printf(" error creating file\n");

      ::srand( time(NULL) + getpid() );

      fprintf( ca_test_file, "%i\n", NUM_CA_TEST );

      for( i = 0; i < NUM_CA_TEST; i++ ){
            if( i == NUM_CA_TEST - 1)
                  fprintf( ca_test_file , "%s",
ca_train_paths[ca_known_indices[i]+1].c_str() );
            else
                  fprintf( ca_test_file , "%s\n",
ca_train_paths[ca_known_indices[i]+1].c_str() );

      }

      fclose( ca_test_file );

      return (0);
}

      /**
       * The method, CreateCAKnownSet, chooses Caucasian faces
       * from the total set of CA faces and prints the file
       * paths of these faces out to a file to be read later.  The known
       * set of faces are unique from the training set.
       *
       * @return int
       */

int FaceReader::CreateCAKnownSet()
{
      int i, r;

      FILE* ca_known_file;

      if( (ca_known_file = fopen( "facelists/ca_known.fpl", "w" )) ==
NULL )
            printf(" error creating file\n");

      ::srand( time(NULL) + getpid() );

      fprintf( ca_known_file, "%i\n", NUM_CA_KNOWN );

      for( i = 0; i < NUM_CA_KNOWN; i++ ){

            r = (int)((((double)::rand() / (double)RAND_MAX) *
NUM_OF_CA)+0.5);

            if( r % 2 == 0 ){ // a younger face
```

```
                  if( InCATrainSet(r) || InCATrainSet(r+1) ||
InCAKnownSet(r) || InCAKnownSet(r+1) ){  // in the training set
                        i--;
                  }else{
                        if( i == NUM_CA_KNOWN - 1)
                              fprintf( ca_known_file, "%s",
ca_train_paths[r].c_str() );
                        else
                              fprintf( ca_known_file, "%s\n",
ca_train_paths[r].c_str() );

                        AddCAKnownIndex( r, i );
                  }

            }else{
                  if( InCATrainSet(r) || InCATrainSet(r-1) ||
InCAKnownSet(r) || InCAKnownSet(r-1) ){
                        i--;
                  }else{
                        if( i == NUM_CA_KNOWN - 1)
                              fprintf( ca_known_file, "%s",
ca_train_paths[r-1].c_str() );
                        else
                              fprintf( ca_known_file, "%s\n",
ca_train_paths[r-1].c_str() );

                        AddCAKnownIndex(r-1 , i);

                  }
            }


      }


      fclose( ca_known_file );

      return (0);
}

      /**
       * The method, CreateBLTrainingSet, chooses equal counts of
       * African American and Caucasian faces and prints their file
       * paths out to file to be read in later.  The faces in the
       * blended training set are unique from those in the known
       * and testing sets.
       *
       * @return int
       */

int FaceReader::CreateBLTrainingSet()
{
      int i, r;

      FILE* bl_train_file;
```

```
      if( (bl_train_file = fopen( "facelists/bl_train.fpl", "w" )) ==
NULL )
            printf(" error creating file\n");

      ::srand( time(NULL) + getpid() );

      fprintf( bl_train_file, "%i\n", NUM_BL_TRAINERS );

      for( i = 0; i < NUM_BL_TRAINERS; i++ ){

            if( i % 2 == 0 ){
                  r = (int)((((double)::rand() / (double)RAND_MAX) *
NUM_OF_AA)+0.5);
                  if( i == NUM_BL_TRAINERS - 1 )
                        fprintf( bl_train_file, "%s",
aa_train_paths[r].c_str() );
                  else
                        fprintf( bl_train_file, "%s\n",
aa_train_paths[r].c_str() );
                        AddBLTrainIndex( r , i , 'A');

            }else {
                  r = (int)((((double)::rand() / (double)RAND_MAX) *
NUM_OF_CA)+0.5);

                  if( i == NUM_BL_TRAINERS - 1 )
                        fprintf( bl_train_file, "%s",
ca_train_paths[r].c_str() );
                  else
                        fprintf( bl_train_file, "%s\n",
ca_train_paths[r].c_str() );
                  AddBLTrainIndex( r , i, 'C');


            }
      }

      fclose( bl_train_file );

      return (0);

}

      /**
       * The method, CreateBLTestingSet, chooses equal counts of
       * African American and Caucasian faces and then prints the file
       * paths of the faces out to file to be read in later.  It first
makes
       * sure that testing faces are the second image of a face in the
       * known set.
       *
       * @return int
       */

int FaceReader::CreateBLTestingSet()
{
      int i, r;
```

```
      FILE* bl_test_file;

      if( (bl_test_file = fopen( "facelists/bl_test.fpl", "w" )) == NULL
)
            printf(" error creating file\n");

      ::srand( time(NULL) + getpid() );

      fprintf( bl_test_file, "%i\n", NUM_BL_TEST );

      for( i = 0; i < NUM_BL_TEST; i++ ){

            if( i % 2 == 0 ){
                  if( i == NUM_BL_TEST - 1)
                        fprintf( bl_test_file , "%s",
aa_train_paths[bl_known_indices[i][0]+1].c_str() );
                  else
                        fprintf( bl_test_file , "%s\n",
aa_train_paths[bl_known_indices[i][0]+1].c_str() );

            }else{
                  if( i == NUM_BL_TEST - 1)
                        fprintf( bl_test_file , "%s",
ca_train_paths[bl_known_indices[i][0]+1].c_str() );
                  else
                        fprintf( bl_test_file , "%s\n",
ca_train_paths[bl_known_indices[i][0]+1].c_str() );
            }
      }

      fclose( bl_test_file );

      return (0);

}

      /**
       * The method, CreateBLKnownSet, chooses equal counts
       * of African American and Caucasian faces and prints the file
       * paths of these faces out to a file to be read later.
       *
       * @return int
       */

int FaceReader::CreateBLKnownSet()
{
            int i, r;

      FILE* bl_known_file;

      if( (bl_known_file = fopen( "facelists/bl_known.fpl", "w" )) ==
NULL )
            printf(" error creating file\n");

      ::srand( time(NULL) + getpid() );
```

```
      fprintf( bl_known_file, "%i\n", NUM_BL_KNOWN );

      for( i = 0; i < NUM_BL_KNOWN; i++ ){


            if( i % 2 == 0 ){
                  r = (int)((((double)::rand() / (double)RAND_MAX) *
NUM_OF_AA)+0.5);

                  if( r % 2 == 0 ){ // a younger face

                        if( InBLTrainSet(r, 'A') || InBLTrainSet(r+1,
'A') ){     // in the training set
                              i--;
                        }else{
                              if( i == NUM_BL_KNOWN - 1)
                                    fprintf( bl_known_file, "%s",
aa_train_paths[r].c_str() );
                              else
                                    fprintf( bl_known_file, "%s\n",
aa_train_paths[r].c_str() );

                              AddBLKnownIndex( r, i, 'A' );

                        }

                  }else{
                        if( InBLTrainSet(r, 'A') || InBLTrainSet(r-1,
'A') ){
                              i--;
                        }else{
                              if( i == NUM_BL_KNOWN - 1)
                                    fprintf( bl_known_file, "%s",
aa_train_paths[r-1].c_str() );
                              else
                                    fprintf( bl_known_file, "%s\n",
aa_train_paths[r-1].c_str() );

                              AddBLKnownIndex(r-1 , i, 'A');

                        }
                  }
            }else{
                  r = (int)((((double)::rand() / (double)RAND_MAX) *
NUM_OF_CA)+0.5);

                  if( r % 2 == 0 ){ // a younger face

                        if( InBLTrainSet(r, 'C') || InBLTrainSet(r+1,
'C') ){     // in the training set
                              i--;
                        }else{
                              if( i == NUM_BL_KNOWN - 1)
                                    fprintf( bl_known_file, "%s",
ca_train_paths[r].c_str() );
                              else
```

```
                                    fprintf( bl_known_file, "%s\n",
ca_train_paths[r].c_str() );

                              AddBLKnownIndex( r, i, 'C' );

                        }

                  }else{
                        if( InBLTrainSet(r, 'C') || InBLTrainSet(r-1,
'C') ){
                              i--;
                        }else{
                              if( i == NUM_BL_KNOWN - 1)
                                    fprintf( bl_known_file, "%s",
ca_train_paths[r-1].c_str() );
                              else
                                    fprintf( bl_known_file, "%s\n",
ca_train_paths[r-1].c_str() );

                              AddBLKnownIndex(r-1 , i, 'C');

                        }
                  }


            }

      }

      ::fclose( bl_known_file );

      return (0);
}

      /**
       * The method, CreateAAFaceSet, creates the African American face
       * set by allocating the African American faces given there
       * file paths.
       *
       * @return int
       */

int FaceReader::CreateAAFaceSet(){

      int i = 0;

      fprintf( outFile, "*** Creating AA Face Set ***\n") ;

      for(i = 0; i < NUM_OF_AA; i++){
            IplImage* temp = ::cvLoadImage(aa_train_paths[i].c_str());
            aa_faces[i] = ::cvCreateImage( cvSize (100,110),
IPL_DEPTH_8U, 1 );
            cvCvtColor( temp, aa_faces[i], CV_BGR2GRAY );
            cvReleaseImage( &temp );
      }

      fprintf( outFile, "*** Finished AA Face Set ***\n") ;
```

```
	return (0);
}

	/**
	 * The method, CreateCAFaceSet, creates the Caucasian face
	 * set by allocating the Caucasian faces given there file paths.
	 *
	 * @return int
	 */

int FaceReader::CreateCAFaceSet(){
	int i = 0;
	fprintf( outFile, "*** Creating CA Face Set ***\n") ;

	for(i = 0; i < NUM_OF_CA; i++){
		::IplImage* temp = ::cvLoadImage(ca_train_paths[i].c_str());
		ca_faces[i] = ::cvCreateImage( cvSize (100,110),
IPL_DEPTH_8U, 1 );
		cvCvtColor( temp, ca_faces[i], CV_BGR2GRAY );
		cvReleaseImage( &temp);

	}

	fprintf( outFile, "*** Finished CA Face Set ***\n");

	return (0);
}

	/**
	 * The method, PrintAAFaceSet, prints the filenames of
	 * the African American faces read in by the system.
	 */

void FaceReader::PrintAAFaceSet(){
	int i;
	fprintf( outFile, "\n\t AA Face Set\n");
	for(i = 0; i < NUM_OF_AA; i++){
		//fprintf( outFile , "%i: %s \n", i,
aa_train_paths[i].c_str() );
		fprintf( outFile , "%i: %s \n", i, aa_file_names[i].c_str()
);
	}
}

	/**
	 * The method, PrintCAFaceSet, prints the filenames of
	 * the Caucasian faces read in by the system.
	 */

void FaceReader::PrintCAFaceSet(){
	int i;
	fprintf( outFile, "\n\t CA Face Set\n");
	for(i = 0; i < NUM_OF_CA; i++){
		//fprintf( outFile , "%i: %s \n", i,
ca_train_paths[i].c_str() );
```

```
            fprintf( outFile , "%i: %s \n", i, ca_file_names[i].c_str()
);
      }
}

      /**
       * The method, GetNumFaces, returns the number of total number
       * of faces being read into the system.
       *
       * @return int -- the number of total faces
       */

int FaceReader::GetNumFaces(){
      return NUM_OF_FACES;
}

      /**
       * The method, GetNumAA, returns the number of total number
       * of African American faces being read into the system.
       *
       * @return int -- the number of African American faces
       */

int FaceReader::GetNumAA(){
      return NUM_OF_AA;
}

      /**
       * The method, GetNumCA, returns the number of total number
       * of Caucasian faces being read into the system.
       *
       * @return int -- the number of Caucasian faces
       */

int FaceReader::GetNumCA(){
      return NUM_OF_CA;
}

      /**
       * This is the constructor which is called when this class is
       * instanstiated.  The constructors opens an output which key
       * to evaluating the system and debugging.
       */

FaceReader::FaceReader()
{
      if((outFile = fopen("output/reader_out", "w")) == NULL)
            printf(" error creating file\n");

}

      /**
       * This is destructor which is called when the program exits to
clean
       * up and memory allocated by this class.
       */
```

```
FaceReader::~FaceReader()
{

}
```