

# **Facial Recognition on Mobile Platforms**

Student:	Terry Worona (100 603 984)
Organization:	Carleton University
Course:	Comp 4905 Honors Project December 2009
Supervised by:	Dr. Gerhard Roth, Adjunct Faculty Advisor Carleton University Computer Science

# Abstract

The introduction of smart phones configured with cameras, GPS systems and internet browsers have ushered in a new era of information technology. The report details the creation of *Look*, an iPhone application which addresses the lack of integration between computer vision and mobile platforms. *Look* is designed to locate a face within an image captured by the iPhone's camera and to retrieve a match from stored images in a remote database. The closest match is returned and displayed together with the original source image. The model database is created in an off-line process using an eigenspace algorithm which compresses images by storing them as eigenvector representations, while the actual face images are stored on a Flickr web server. The model database is compiled as an XML file and downloaded by *Look* at startup. As a match is located, the specific face image is returned to the iPhone via a Flickr JSON request.

The report determines that it is possible to cross-compile OpenCV libraries for mobile ARM architectures. Contributions are noted in the conclusion, providing specific instructions for porting OpenCV to the iPhone, representing an image database as an eigenspace XML and querying REST services such as Flickr.

# Acknowledgements

A sincere thank you is extended to Dr. Gerhard Roth for his supervision and continual support in the evolution of this project. I trust that that this report will shed light on the future of mobile applications in conjunction with computer vision topics. I would also like to thank the iPhone development community, in particular the users of the [iphonebook.com](http://iphonebook.com) forums; their support and encouragement allowed for the creation of the envisioned application.

# Table of Contents

Introduction	7
Related Work	9
Background	10
Mobile Device History	10
iPhone SDK	11
Computer Vision	11
Object Recognition	11
OpenCV	12
JSON	12
Objectives and Goals	13
Motivation	13
Project Requirements	14
Personal Objectives	14
Server Design	14
Face Space	15
Client Design	16
Recognition	16
Singleton Pattern	18
iPhone Application Design	19
MVC	19
Navigation Hierarchy	21
JSON Flickr Request	23
Contributions	25
Cross-compiling OpenCV 2.0	25
Configuring X-Code	25

Exporting Eigenspace as XML	25
Evaluation	26
Performance	26
Accuracy	26
Proposal Deviations	27
Conclusion	28
Results	28
Future Work	28
Bibliography	30
Appendix-A	A-1
Appendix-B	B-1

## List of Figures

FIGURE 1- <i>Face Space Deployment</i>	15
FIGURE 2- <i>Look's Image Capture Screen</i>	16
FIGURE 3- <i>Look's State Diagram</i>	17
FIGURE 4- <i>Singleton Structure</i>	18
FIGURE 5- <i>Look's Object Model</i>	20
FIGURE 6- <i>Navigation Hierarchy</i>	21
FIGURE 7- <i>Look's Main View</i>	22
FIGURE 8- <i>Look's Detail View</i>	23
FIGURE 9- <i>Flickr JSON Request</i>	24
FIGURE A1- <i>Images Representing a Single Model Instance</i>	A-6
FIGURE A2- <i>Flickr metadata</i>	A-6
FIGURE B1- <i>iPhone Simulator Runtime Configuration</i>	B-2
FIGURE B2- <i>iPhone Simulator Running Look</i>	B-3

## List of Tables

TABLE 1- <i>Look Face Class</i>	20
TABLE A1- <i>FaceSpace Directory</i>	A-4
TABLE A2- <i>loadDataFromServer Flag</i>	A-5
TABLE B1- <i>Deliverable CD Contents</i>	B-1

# Introduction

Many computer vision topics, particularly facial recognition, are restricted to experimenting on webcams or other wired image capturing devices. With the introduction of mobile devices that are packaged with straightforward development environments, programmers now have the capability to apply vision algorithms wirelessly. This report and its associated application *Look*, intend to create such a service within a mobile platform, at the same time addressing issues such as hardware restrictions and scalability.

OpenCV libraries are the backbone of computer vision, offering well-documented functions that allow developers to avoid extraneous and complicated mathematics. Without these essential libraries, vision-related topics on mobile devices would not be possible. The OpenCV framework has been deployed and seamlessly integrated into both iPhone and Windows Mobile devices with encouraging results. As technology and hardware developments continue to advance, there is no doubt that OpenCV libraries and mobile platforms will be used to complement each other.

*Look* offers nothing revolutionary to the area of computer vision in terms of algorithms or computational techniques. Instead, the prototype application aims to demonstrate the possibilities that will become apparent once users begin to take advantage of OpenCV libraries on mobile platforms. This study will discuss the pitfalls associated with executing memory intensive operations on a restricted hardware set such as the iPhone. Since Objective-C does not contain a garbage collector, processes such as converting images to and from OpenCV C++ libraries are prone to memory leaks. The report will illustrate techniques and design patterns used to eliminate memory leaks and ensure that the application was operating under optimal resource allocation. Although Gordon Moore's accepted law states that the power of a chip doubles every two years, it does not pertain to mobile devices in the same sense. Moore's Law, instead, is not about doubling transistor size rather than creating super-combo chipsets able to support common features associated with present mobile devices. Chips custom-tailored to supporting FM radios, TV tuners and video cameras are commonplace in present-day mobiles<sup>1</sup>. It is likely

---

<sup>1</sup> Malik: ¶2, ¶9

that a port of the framework to mobile devices and simple facial recognition application are only the beginning of what will manifest in the future.

The report introduces a brief history of mobile devices and provides an overview of the technologies applied to *Look*. Next, personal objectives and the functional requirements of the deliverable *Look*, as well as the motivation behind the project, are addressed. Pre-existing knowledge of frameworks and design patterns used to facilitate the application are described followed by a detailed synopsis of the application's final structure. Specific contributions, notably cross-compiling OpenCV to the iPhone platform, are elaborated in hopes that others may utilize the information to propel their own projects within the field of mobile vision. Lastly, conclusions are stated related to performance, accuracy and future considerations within the computer vision and mobile application realm.



## Related Work

Few existing papers deal with OpenCV libraries on mobile devices. One such paper, by Hadid et al, describes the successful port of OpenCV to a Windows Mobile phone in which a HAAR-like face detector<sup>2</sup> is constructed to identify facial representations within a photograph. The work addresses the limited resource availability on mobile devices and algorithm implementations that are tailored towards mobile performance<sup>3</sup>. Overall results are similar to that of *Look*; a serious tradeoff between performance and functionality on mobile devices running vision libraries.

A theoretical-based paper by Wittke et al, details the role mobile phones can play in conjunction with life-logging; recording an individual's day to day activities in explicit detail. A person's physical movements and conversations with others can be converted to data and mined for further analysis<sup>4</sup>. A prototype application built on a Unix-based mobile utilized a constant video feed to detect hand gestures and facial features. A Face locator, similar to HAAR was used once again with similar performance results.

*Look* aims to build upon these foundational mobile vision papers by utilizing the latest in mobile hardware and software technology, OpenCV and the Apple iPhone<sup>5</sup>. Consequently, *Look* will implement intensive image-related tasks aside from simple face detection, already proven to have acceptable results of identifying up to two faces per second<sup>6</sup>.

---

<sup>2</sup> Viola: p511

<sup>3</sup> Hadid: p101

<sup>4</sup> Wittke: p2182

<sup>5</sup> *Look* was developed and released under OpenCV 1.0 because 2.0 libraries were not yet released. However, instructions for porting OpenCV 2.0 were added to Appendix A.1 as a last minute addition.

<sup>6</sup> Hadid: p106

# Background

## Mobile Device History

In the early 1990's, mobile devices were segregated in two dominate groups: cell phones and Personal Data Assistants (PDA). The former could make and receive calls and text messages as well as store basic contact information. The latter could be synced serially to a computer to gather calendar, address and email related data, but did not connect to a wireless network. Towards the end of the 20th century, the two markets consolidated as cellphone companies encouraged customers to purchase what they coined 'Smart Phones', devices that offered the features of both cellphones and PDAs alike. Because cellphone manufacturers at the time offered a wide variety of models to many different providers, a common operating system was required. The major hardware manufacturers collaborated on the first-known, widely distributed Smartphone OS, the Symbian Operating System<sup>7</sup>. This operating system was stable enough to be distributed on a variety of different hardware platforms, but 'locked-down' to anyone wishing to build on it. Much like it's ancestor Sybian OS, the Apple iPhone was originally released as a closed system. It was not until Apple released a free public software development kit (SDK) in March 2008 that it began to significantly impact the mobile device market. The SDK permitted developers to create and distribute their own 'home brew' applications on the Apple App Store. The iPhone camera, GPS and quick internet connection were all at the fingertips of eager developers looking to 'strike it rich' by creating the next application. In just three months, the SDK was downloaded more than a quarter-million times<sup>8</sup>, which produced 100 000 unique applications to date<sup>9</sup>. Consequently, the Symbian OS together with other variants including the RIM Blackberry platform lost significant market share<sup>10</sup>.

---

<sup>7</sup> Anderson: p65

<sup>8</sup> Bowcock: ¶1

<sup>9</sup> Kerris : ¶3

<sup>10</sup> Anderson: p65

## **iPhone SDK**

The Software Development Toolkit (SDK) is a chain of tools used for the development, testing and deployment of software for the iPhone. The kit comes prepackaged with several core frameworks used for communication with iPhone's virtual memory, file system, networks and threading. The user is free to add and remove frameworks as deemed necessary. As this report will demonstrate, OpenCV libraries can be cross-compiled for the iPhone OS and deployed as any other native framework. OpenCV is primarily written in C++; subsequently, there is added overhead in converting and transferring objects to and from C++ to the SDK's native language Objective-C.

## **Computer Vision**

Computer vision can be defined as the "set of computation techniques aimed at estimating or making explicit the geometric and dynamic properties of the 3-D world from digital images"<sup>11</sup>. In other words, computer vision is the interpretation of digital images using recognized mathematical techniques. The most successful applications of machine vision stem from the automation of observable inspection tasks. In these scenarios, jobs traditionally held by humans, such as the quality inspection of items traveling down an assembly line, can be replaced by a camera system capable of detecting abnormalities<sup>12</sup>. Automatic Target Recognition (ATR) is another area of application where weapons can be retrofitted with cameras to increase target accuracy<sup>13</sup>. As the field continues to gain relevance, vision-related areas of study are becoming commonplace requirements in computer science curricula.

## **Object Recognition**

For the purpose of this report, object recognition will be restricted to the process of finding a source object within a set of sample images or video. Traditionally, this aspect of recognition is referred to as "identification". The second component of recognition, determining an object's 3-dimensional location,

---

<sup>11</sup> Trucco: p2

<sup>12</sup> Grimson: p46

<sup>13</sup> Ibid

is not relevant to *Look*'s overall goal. The object recognition model assumes that the sample set is predetermined and readily available; it is only possible to recognize an object with specified information beforehand<sup>14</sup>. Model-based recognition is a particular brand of the identification problem and focuses on determining whether a match for a source image can be found within a database of models. In order to solve the recognition problem, two necessary questions need to be addressed:

- I. Which data within a source image corresponds to the object required to be recognized?
- II. Is there an available database of models to search against and can the query be narrowed to a specific subset?

The answer to both of the questions, in relation to this report is 'No'; we do not know where within a sample image a specific face resides and the database model cannot be simplified. This particular combination of scenarios requires an efficient searching algorithm, as well as a means to locate a face within an image. Both aspects have been addressed and explained in detail throughout the report.

## OpenCV

Originally developed by Intel, OpenCV is a library written in C and C++, intended to simplify and extend the many mathematical procedures used in computer vision algorithms. The libraries are cross-compiled to run on both Unix and Windows-based operating systems. OpenCV abstracts confusing and time-consuming tasks associated with implementing vision-based applications, allowing the developer to maintain and construct programs with greater ease and fewer 'bugs'. The vision library contains a total of 500 different functions which have a variety of real world applications<sup>15</sup>. The overall goal of the OpenCV library is to propel the area of computer vision further by allowing more programmers to become involved and formulate relatively sophisticated applications quickly.

## JSON

Javascript Object Notion (JSON) is a lightweight data interchange format and a subset of Javascript. JSON syntax is simple and straightforward, making parsing of information efficient and more

---

<sup>14</sup> Trucco: p2

<sup>15</sup> Bradski: p1

desirable on mobile applications where processor and memory management is of the highest priority<sup>16</sup>. JSON also has the ability to directly represent basic data structures including strings and collections. While other data interchange mechanisms, such as XML, do have instruments in place to transform the markup into usable structures, it often comes at a significant cost in terms of execution time. JSON was the ideal choice for communicating to and from a web server on a mobile device because of its straightforward structure and lightweight design.

## Objectives and Goals

### Motivation

Both computer vision and mobile device development are two disciplines that are expanding at incredible rates. The open-source community in both areas is strong, offering help and support through the use of forums, chat rooms and newsgroups. There is a clear connection between the two topics and it is only natural to begin porting computer vision techniques to mobile platforms; this allows the applications to be wireless and decoupled from bulky webcams and desktop computers.

Carleton University, among other post-secondary educational institutions, focuses not only on teaching fundamental computer science theory, but also envisions the future. As a result, Carleton recently announced a mobile development stream within the Department of Computer Science. The motivation behind this project is to shed light on the possibilities involved when porting OpenCV to the iPhone. *Look* does not add new algorithms or techniques to the computer vision discipline; instead, it is a demonstration of what is possible when the libraries are taken to the mobile realm. Additionally, the report strives to demonstrate possible study areas for Carleton's innovative mobile development stream.

---

<sup>16</sup> Crockford: ¶13

## Project Requirements

The project requirements set out by the author are:

- I. Using existing OpenCV libraries, write an efficient algorithm capable of accepting a source image and compare it to a database for potential matches. The algorithm will address two key points of interest in model-based recognition:
  - i. Determine the 2-dimensional positions of a face within a sample image, based on a pre-specified object model;
  - ii. Compare the 'face' against a database of potential matches, based on co-ordinates found in the recognition phase. The algorithm will return the closest possible match.
- II. Develop an iPhone application which will provide an interface to the algorithm created in Objective I. The iPhone will capture an image (a face), compare it to a database representation (an xml file pulled from a web server at startup) and return the closest matching image.
- III. Add local persistence to the iPhone application in order that users can add, review and delete past verification history. Because storing images locally on the iPhone is memory and hard disk intensive, they will be retrieved using REST request to a Flickr account.

## Personal Objectives

The personal objectives set out by the author are:

- I. Port the OpenCV framework to the iPhone SDK.
- II. Familiarize myself with the iPhone SDK and Objective-C.
- III. Extend my knowledge of web-based services to include new technologies such as REST requests.

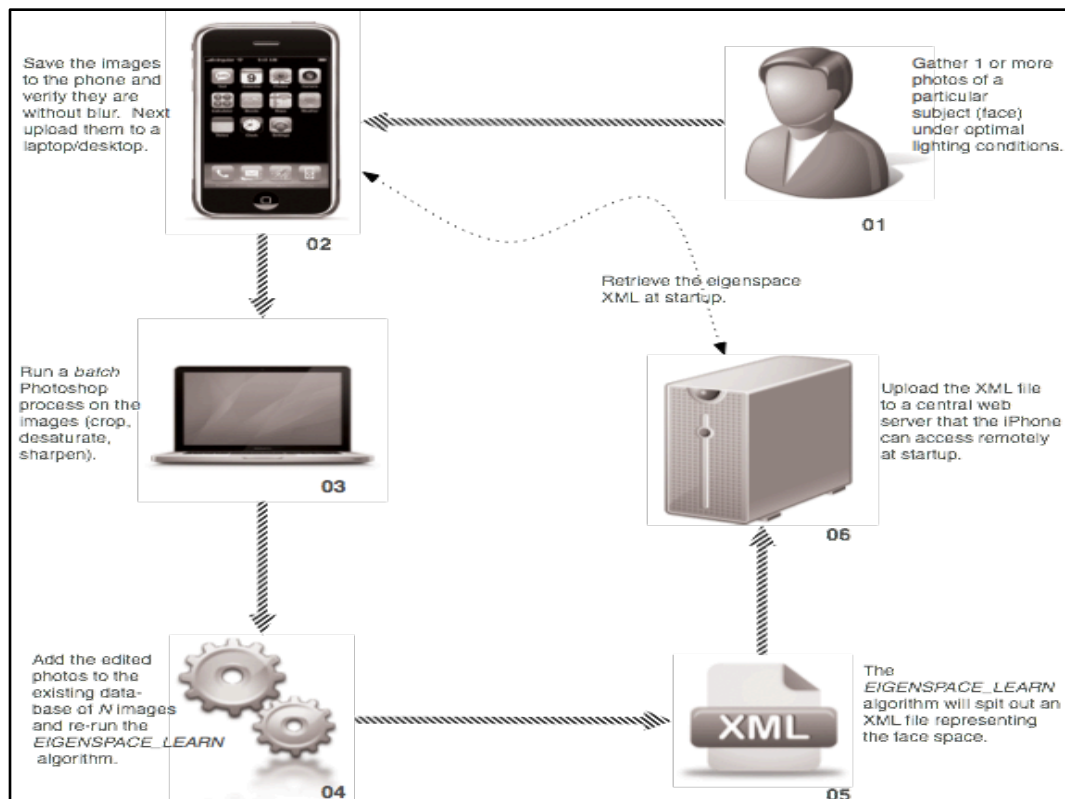
## Server Design

Mobile devices carry significant performance overhead from not only their lack of processing power, but also minimal physical memory. For this reason, the model database, which is to be represented in XML format as an eigenspace, will remain on a server. The generation, modification and general maintenance of the face space is to be carried out on a machine separate from the mobile device. In this sense, *Look* does not have the ability to directly manipulate the model database.

## Face Space

Information theory, simply stated, it is the task of converting abstract information to quantified terms that can be mathematically and statistically modeled to a variety of scenarios ranging from data compression to electrical engineering. Information theory is applied to facial recognition through Principal Components Analysis (PCA), a technique first addressed by Turk and Pentland<sup>17</sup>.

The EIGENSPACE\_LEARN<sup>18</sup> algorithm resides on a desktop machine separate from the iPhone application. It takes as it's input, N images, converts each to a vector representation and computes a master covariance matrix consisting of N eigenvectors. The algorithm extracts N' eigenvectors (highest eigenvalues) and creates an eigenspace with them, commonly referred to as a "face space"<sup>19</sup>. The face space is saved as an XML file and uploaded to a server to be downloaded at by *Look at* startup (Fig. 1).



**FIGURE 1-Face Space Deployment**

<sup>17</sup> Turk: p72-86

<sup>18</sup> EIGENSPACE\_LEARN is a generic name for the eigenface.c program (Appendix A.3).

<sup>19</sup> Trucco: p268

## Client Design

Tasks such as locating a facial image and retrieving a match within a database could easily reside on a server, increasing execution times significantly. Rather than explore what is possible, for example image processing on a central server with ample resources, *Look* aims to answer questions that have not yet been posed. Since little work has been accomplished in the area of consumer mobile vision, there are limited benchmark statistics to refer to. All other vision-related tasks, aside from the construction of a face space, reside on the iPhone itself. *Look* is intended to test the support that mobile hardware could provide to OpenCV and required tasks for image recognition.

### Recognition

The EIGENSPACE\_IDENTIFY<sup>20</sup> algorithm resides within the Utils class of *Look*. It takes as it's parameter a test image captured with the iPhone's two-megapixel camera as demonstrated in Fig. 2.



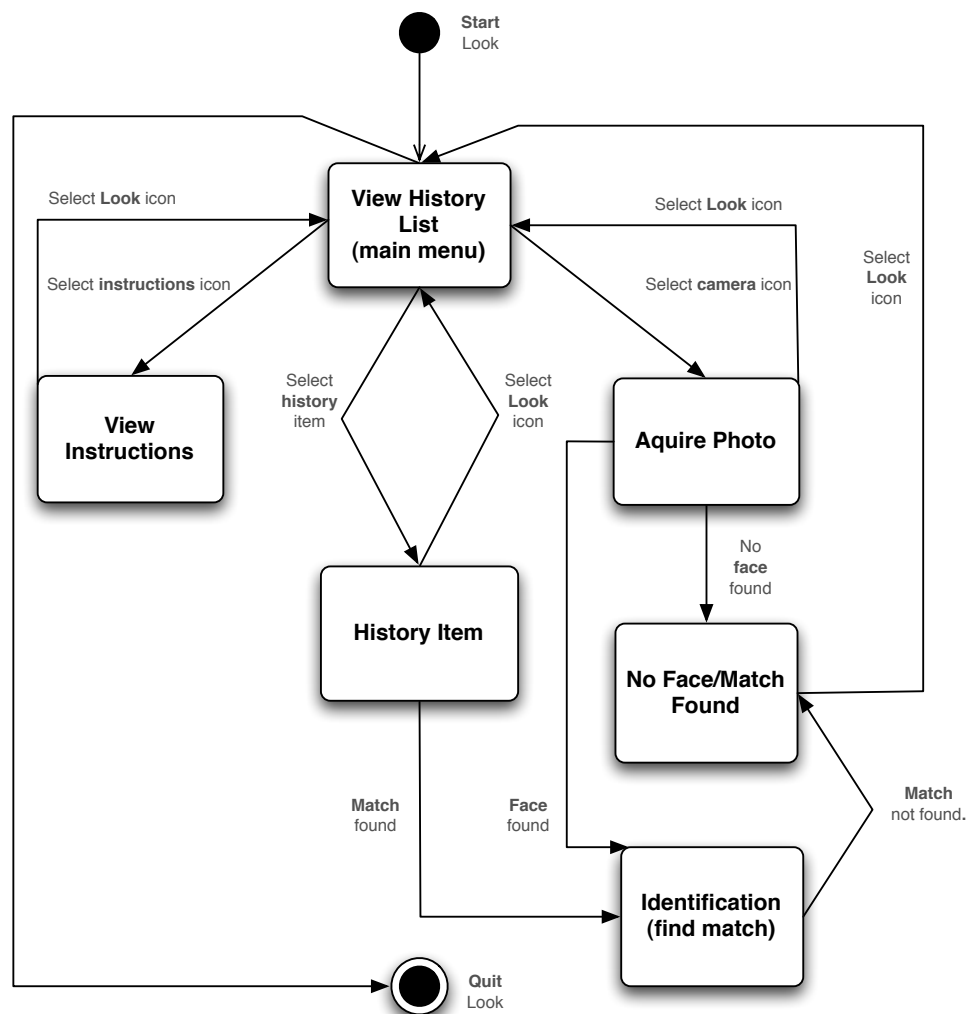
**FIGURE 2** -*Look's Image Capture Screen*

---

<sup>20</sup> EIGENSPACE\_IDENTIFY is a generic name for a collection of functions used with *Look*.



The algorithm first applies a preprocessing technique to locate and crop a face within the test image. Then, the face-image is converted to greyscale and projected onto the eigenspace obtained from the remote web server at startup. The projection calculates a Euclidean distance between the test image and all model images within the eigenspace. The returned value is a unique tag representing the shortest Euclidean distance found within the projection<sup>21</sup>. The tag is used to retrieve the matching image file via a REST request to a Flickr server. A state diagram representation of *Look* in Fig. 3 summarizes the recognition phase as well as other application features.



**FIGURE 3-Look State Diagram**

<sup>21</sup> Trucco: p268

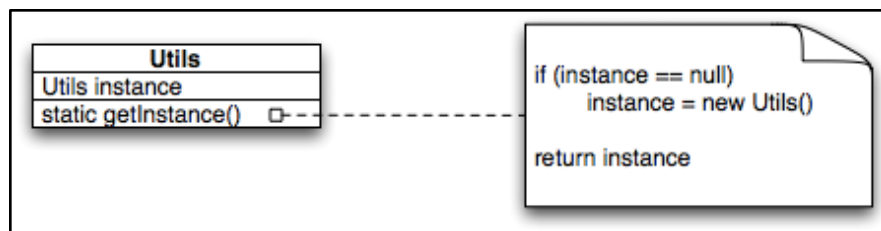
The techniques and algorithms used within *Look* are not constrained to a particular hardware set such as a specific DSLR Camera or webcam, making the application extensible platforms aside from the iPhone.

## Singleton Pattern

The singleton design pattern is used to restrict instantiation of an object to a single occurrence<sup>22</sup>. In *Look*, the Utils class represents a toolbox that performs various OpenCV procedures tailored to the application, for example locating a face within a source image and cropping it. Various procedures that convert Objective-C UIImage types to C++ ImplImage are present in the Utils class. It was chosen to be a singleton class for two reasons:

- I. It is the ideal choice for global variables and can be tailored to allow or reject access to specific objects. In the case of *Look*, specific controllers should have access to the singleton.
- II. The Utils class requires precise care and memory management. Its entire lifetime must be monitored from instantiation to destruction. If multiple instantiations of the class were present, the program would certainly bottom out due to insufficient memory.

Fig. 4 illustrates the use of the singleton pattern as it relates to the Utils class.



**FIGURE 4-Singleton Structure**

The application delegate handles application-level functionality such as initializing and destroying global objects and managing application-wide behavior<sup>23</sup>. Since the Utils singleton is a type of global

<sup>22</sup> Gamma: p128

<sup>23</sup> Mark: p44

variable that is needed by all controllers at any point in time, it resides within the delegate. The delegate can be accessed by any controller at any point of the program's execution.

## iPhone Application Design

The client and server sections collectively describe the business model behind *Look*. They serve to provide an overview of how the application is configured to address the overall goal of locating a face within a photograph and retrieving a match within a remote structure. The application design is a detailed look at the various components that support the business in achieving its goal by providing a means for the client to connect to the server and for a user to interact with the system via a user interface.

### MVC

The Model View Controller (MVC) framework, although apparent in most modern object-oriented systems, is nevertheless important to consider in *Look*. The model consists of business information, which may or may not be persisted to a local or remote database. The controller defines the way the model and user interface interact with one another. Lastly, the View is the presentation layer and most commonly what the user sees when using an application<sup>24</sup>. The latter two elements are provided via Apple's navigational hierarchy structure, elaborated in the next section of this study.

Apple's Core Data Framework is a robust and dynamic method of visually representing data models, although absent from all releases of the SDK until version 3.0<sup>25</sup>. Core Data requires a significant amount of overhead work to setup and configure. For simple applications such as *Look*, it provides little advantage over traditional persistence mechanisms that rely on XML schemas. Nonetheless, it was chosen for two important reasons:

- I. Core Data, to date, is the best model for storing and persisting images. Since *Look* stores the source images of a recognition attempt, it was the primary choice.

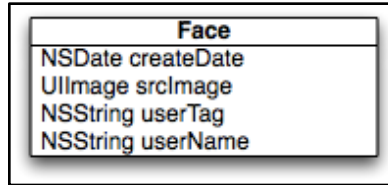
---

<sup>24</sup> Gamma: p2

<sup>25</sup> Mark: p379

- II. For large, complex systems, Core Data simplifies the model creation process significantly. Although in its current state *Look* is fairly straightforward, at any point it can be extended to include a more robust and detailed design. Thus, Core Data increases *Look*'s scalability for future additions.

*Look*'s current object model is a single, persisted class (Fig. 5). It's attributes are illustrated in Table 1.



**FIGURE 5-***Look Object Model*

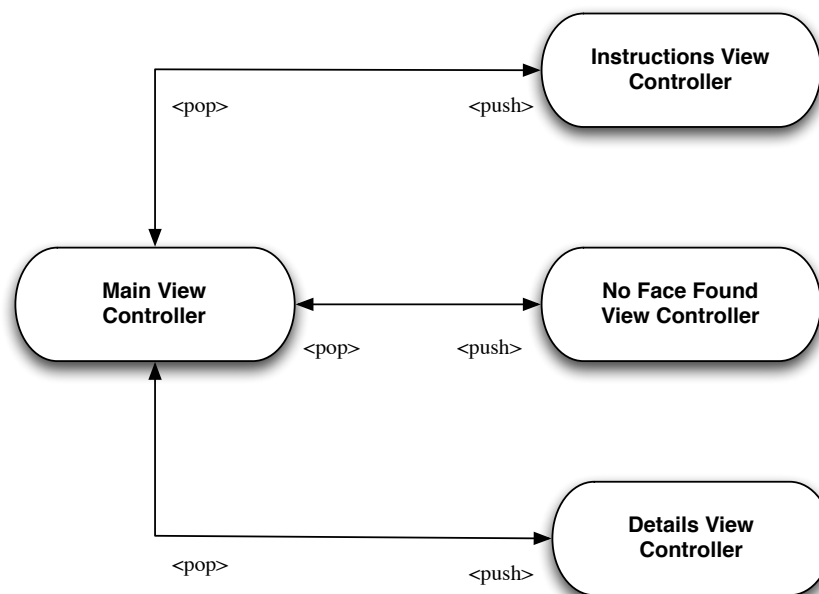
Attribute	Type	Description
createDate	NSDate	The date the recognition event took place.
srcImage	UIImage	Once the application captures an image with the iPhone's camera, the controller will employ a utility class to locate a face and crop it. The result is the srcImage.
userTag	NSString	When a face has been verified, it's closest match will be returned as a unique tag. This userTag can be utilized to retrieve a face image from a Flickr server using a JSON request.
userName	NSString	The user name of the match image is stored on the Flickr server and paired with the userTag. This attribute is retrieved using a JSON request.

**TABLE 1-***Look Face Class*

All Core Data model objects reside in a persistence store, which by default is a SQLite database<sup>26</sup>. The most beneficial aspect of the Core Data framework is that it incorporates all the work associated with loading, saving and updating the objects, provided through a class called a “Managed Object Context”. Core Data will even persist data to the store in the event of an application crash or if one simply forgets to flush it. The context resides in the applications delegate which is a central access point for all controllers within the application’s framework.

## Navigation Hierarchy

Apple provides an efficient framework for creating navigation-based applications, a perfect compliment to the Core Data model and the MVC paradigm. A root controller is the entry point to the application. *Look*’s root controller subclasses UINavigationController which provides functions to push and pop new UIViewControllers onto the stack, allowing the developer to build complex hierarchical systems iteratively. *Look*’s navigation hierarchy is demonstrated in Fig. 6. Each UIViewController acts as both a view and a control mechanism to the application’s delegate which moderates access to the business model and persistence stores.

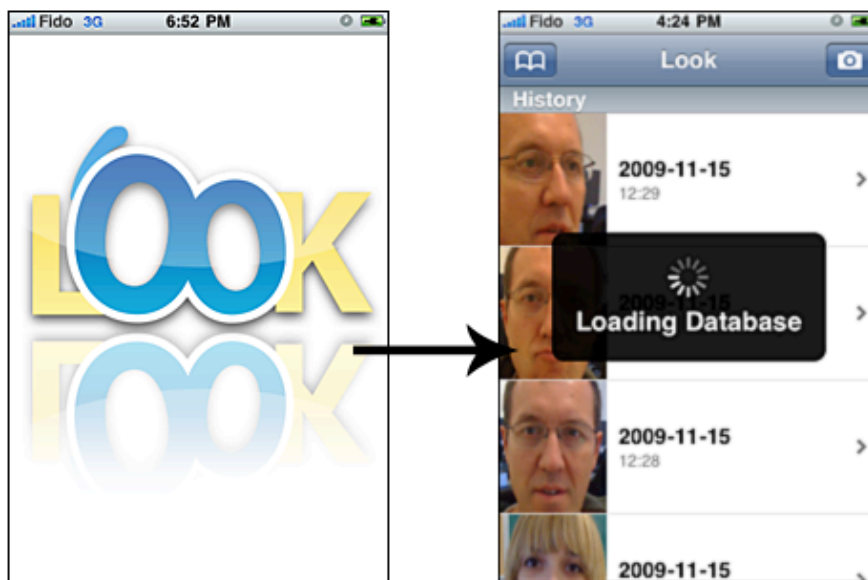


**FIGURE 6-***Navigation Hierarchy*

---

<sup>26</sup> Mark: p379

The iPhone has only 128MB of RAM and of that space approximately one-half is reserved for screen buffering and other system processes<sup>27</sup>. In order to stay within boundaries, match images are stored on a Flickr server and may be retrieved via the JSON API. Consequently, the hierarchical schematic lends itself to loading and unloading matching images at the user's request. When the Main View Controller is loaded, a list of face objects is acquired from the SQLite database and placed into a table view as shown in Fig. 7.

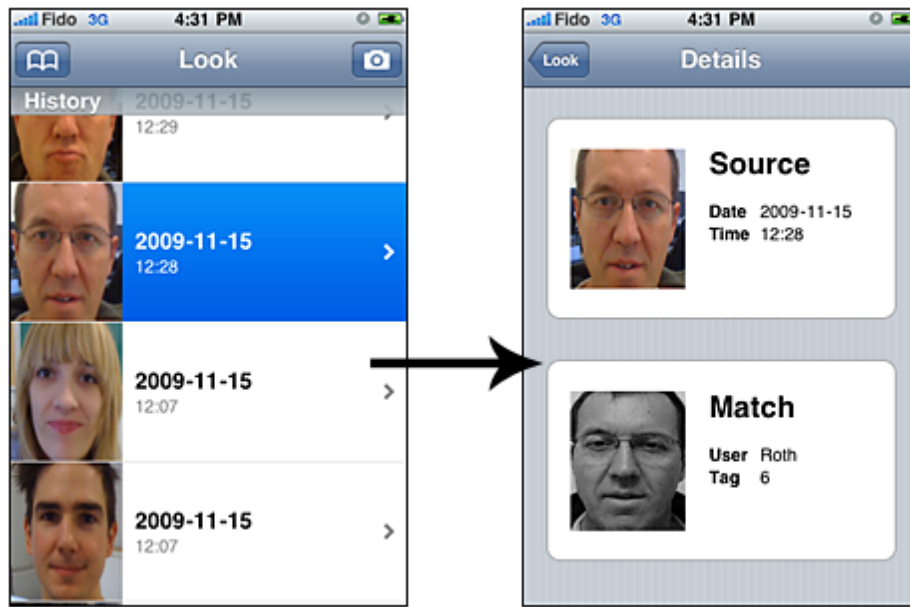


**FIGURE 7-***Look's Main View (Loading Recognition History)*

It is only when the user selects a particular row that the match image is retrieved from Flickr using the user tag. The match image, coupled with the user name and original Face object, is used to instantiate a Details View Controller and push it onto the stack as demonstrated in Fig. 8. The structure keeps hard disk space at a minimum by separating requests into smaller, more manageable pieces through each level of navigation

---

<sup>27</sup> Ibid: p6



**FIGURE 8-***Look's Detail View*

Information is passed in a single direction down the hierarchy from parent to child. The child does not need to know anything about its parent class other than what it provided in the initialization stage. This technique, known as 'loose coupling', increases the chances of the controller being reused later or swapped for a new type of implementation in the future<sup>28</sup>.

## JSON Flickr Request

Flickr is a popular online community where users can upload, edit and share their photographs amongst each other. Flickr provides the functionality to add descriptions, titles and tag annotations to any image that is uploaded. As previously noted, a Flickr account will store a representative image of each individual within the model database. Furthermore, all representatives are provided with a unique tag that can be used for retrieval. The unique tag will correspond to the userTag instance variable of the Face class. Recall that this userTag is populated when a match has been found within the model's eigenspace during Principal Components Analysis (PCA).

<sup>28</sup> Gamma: p26

Tags are a simple and direct annotation mechanism to be used in a variety of different scenarios, most of which are beyond the scope of this project. Tags provide an expeditious means of querying a database of images residing on a server. A tag framework has several drawbacks, the most noteworthy being its lack of query refinement<sup>29</sup>. Although irrelevant to the current structure of *Look*, if a requirement was added to group users by location, department or role, tags could not provide the functionality of refining queries to accommodate such a task.

Representation State Transfer (REST) is a popular web framework based on HTTP protocols. It is a set of architectural constraints that focuses on simplifying network communications by reducing latency and maximizing scalability. REST provides a simple mechanism for defining resource representation and information exchange between web servers and clients.<sup>30</sup> Flickr provides, among other REST services, a robust JSON API that allows developers to base queries on various attributes including date, tag, description, title and username. A user must obtain an alpha-numeric key value from Flickr that will be passed into all REST HTTP requests. The key is free and easily obtainable with a valid Flickr account. The basic HTTP request in *Look* includes the username, Flickr key and tag as depicted in Fig. 9.

```
// assume we have variables api_key, user_id and user_tag from a Face object
NSString *urlString = [NSString stringWithFormat:@"
    http://api.flickr.com/services/rest/?method=flickr.photos.search
    &api_key=%@
    &user_id=%@
    &tags=%@
    &format=json", api_key, user_id, user_tag];
```

**FIGURE 9-Flickr JSON Request**

---

<sup>29</sup> Schmitz: p1

<sup>30</sup> Fielding: p116



The urlString serves to initialize an NSURLConnection which if successful, will return the URL to a unique photo coinciding with the userTag. The URL can then be employed to load and/or unload past recognition matches as requested by the user.

## Contributions

### Cross-compiling OpenCV 2.0

In order for *Look* to take advantage of OpenCV, a universal static library must be created for both Intel and ARM processors. The Intel library will be used in conjunction with testing, debugging and developing within the iPhone simulator provided by the SDK. ARM is a reduce instruction set processor that is suitable for low power applications. ARM processor's are relatively small size and ease of use have made them dominate in the mobile device market. In 2007, 98% of all mobile devices were utilizing ARM chips<sup>31</sup>. To cross-compile OpenCV for the architectures mentioned, refer to Appendix A.1.

### Configuring X-Code

Once the static cross-compiled library has been created for ARM and Intel-based processors, it may be included within any X-Code project. The libraries will be free to use just as any other library and packaged automatically with an application build. The steps outline in Appendix A.2 demonstrate how to configure X-Code for use with OpenCV static libraries.

### Exporting Eigenspace as XML

As previously stated, a face space is exported as an XML file on the server side. For convenience, a sample face space was packaged within the contents of the CD deliverable together with the code used to generate the XML file. Appendix A.3 reveals the 'FaceSpace' folder structure of the deliverable CD along with instructions on how to generate and import face spaces into *Look*.

---

<sup>31</sup> Krazit: ¶2

# Evaluation

## Performance

*Look* was primarily developed within the SDK's simulation environment which does not replicate the iPhone's hardware. In other words, *Look* was being developed and tested on an accelerated, modern computer with 4GB of RAM and a dual core CPU. Locating a face within an image involved mere milliseconds and retrieving a corresponding match, even within a database of 50 plus models, took no more than a few seconds. Fortunately, when the application was deployed to the iPhone, the results did not differ dramatically from what was expected. Albeit the unit runs on a modest 333 MHz CPU and 128MB of RAM, the entire recognition phase took no longer than 30 seconds to complete.

Intel Performance Primitives (IPP) is a low-level signal and image processing library created by Intel to optimize OpenCV performance<sup>32</sup>. Unfortunately such libraries are proprietary to Intel chips and cannot be ported to ARM processors or similar variants. While executing various OpenCV procedures, speed increases can be as high as 38% when using 64bit chipsets with IPP turned on<sup>33</sup>. As demonstrated, if Moore's Law continues as predicted into the 21st century as a model to describe mobile chipset features, perhaps manufacturers will take advantage of Intel based CPUs that are coupled with IPP libraries in order to maximize computer vision techniques. Currently, ARM processors do not have the features and qualities found in most entry level Intel chipsets.

## Accuracy

The information theory approach to facial recognition may produce principal components within a model that do not discriminate against facial features, but from the environment in which the images were captured. Characteristics including background, side lighting and color temperature can be misunderstood as representing key facial features instead of spurious data representing the surrounding

---

<sup>32</sup> Landré: p1

<sup>33</sup> Ibid: p5

context<sup>34</sup>. A variety of pre-processing techniques may be applied to both model and test data to reduce the risk of obtaining false acceptance or false rejection results. The pre-processing occurs prior to determining the principal components of a model image and after a test image is projected onto the space.

As explained, *Look* will locate a face within a test image before any recognition takes place; this itself is a form of preprocessing. Removing extraneous data which represents an object's surroundings greatly reduces the likelihood of unwanted data being represented as a principal component. After a face is located, it is converted to greyscale in order to remove any color temperature inconsistencies. Color normalization methods were considered to be included in *Look* such as the universally recognized 'Intensity Normalization' algorithm, which attempts to average all three color channels each time the image's intensity increases by a predefined factor. Another respected technique, known as 'Grey World', improves images with large color variances by having each of the RGB channels average to a common grey value. The aforementioned algorithms are impressive on paper, but are surprisingly error prone due to loss of information. Particular features such as image edges are lost from intensity and color reduction techniques, making the images more difficult to segregate into principal components<sup>35</sup>.

Algorithms that increase the intensity of edges within a sample image have yielded encouraging results, at times reducing errors rates (the chance of returning a false match or false rejection by 11.6%<sup>36</sup>. Although *Look* does not implement such techniques, it is still worthy to note them for consideration in future releases.

## Proposal Deviations

The final deliverable has minor deviations from the original proposal which suggested that *Look* return a list of matches along with the accuracy (Euclidean distance) of each. The final product returns a single match, based on the shortest Euclidean distance within a face space. The most elegant and

---

<sup>34</sup> Heseltine: p678

<sup>35</sup> Ibid: p685

<sup>36</sup> Ibid

intuitive designs are usually the simplest, so rather than inflate the application with features the author decided to keep it straightforward.

Secondly, the configuration of the server-client architecture differs slightly than the original proposal. Rather than placing all of the image-related operations on a central web server, the student writer put almost all of the content on the mobile itself. The reason behind this choice was to test the iPhone's reliability and performance when handling complex OpenCV instructions. The report's direction took into account implementing what was presently possible to what may be possible in the future.

## Conclusion

### Results

The *Look* prototype serves as a stepping stone on the road towards mobile hardware and computer vision sharing a more common, unified platform. The project was not intended to break ground as far as adding new or improved content to existing OpenCV libraries; rather, its intent is to inspire others towards thinking about mobile vision in general. At this point, facial recognition techniques on mobiles are indeed possible and executable within a respectable time frame. As mobile chipsets move towards 'feature-based' approaches to expansion, perhaps manufacturers will be inclined to include elements such as IPP to improve performance for visual related tasks.

### Future Work

In its current form, *Look* is not a complete distributed application. That is, it does not have a complete client-server framework. Ideally, the iPhone application would include the ability to add users to the database via their mobile, a task that is beyond the timeline and scope of this project.

Social media outlets such as Facebook and Myspace contain an enormous amount of public information. Facebook alone provides a profile page for each and every user, currently estimated at 300 million plus worldwide<sup>37</sup>. Each public profile contains names, birthday and location information as well as

---

<sup>37</sup> Facebook: ¶1

a thumbnail image of the user's face. Social websites and the enormous repository of information they supply are attracting the interest of some big name companies. Yahoo!, for example, returns public profile pages in search results<sup>38</sup> and Face.com's application 'Photo Tagger' automatically tags images with usernames based on the Facebook database<sup>39</sup>. The present client-server architecture of *Look* lends itself to scalability. The client can essentially be 'plugged' into any eigenspace provided, assuming it has meta-information for retrieval purposes. The real task lies in converting a potentially huge image database into a compressed eigenspace representation.

---

<sup>38</sup> Agarwal: ¶1

<sup>39</sup> Perez: ¶2

# Bibliography

- Agarwal, Amit. "Yahoo! Integrates Images from Facebook Profiles in Search Results". Digital Inspiration. April 21, 2008. <<http://tinyurl.com/3k7qsc>>.
- Anderson, Eric and Sharon P. Hall. "Operating Systems For Mobile Computing". Journal of Computing Sciences in Colleges. December 2009: 64-71.
- Bowcock, Jennifer and Simon Pope. "iPhone SDK Downloads Top 250,000". Apple Inc. June 9, 2008. <[http://www.apple.com/pr/library/2008/06/09/iphone\\_sdk.html](http://www.apple.com/pr/library/2008/06/09/iphone_sdk.html)>.
- Bradski, Gary and Adrian Kaehler. Learning OpenCV. California: O'Reilly Media, 2008.
- Crockford, Douglas. "JSON: The fat-free alternative to XML". XML 2006, Boston. December 6, 2006.
- Facebook. "Facebook Statistics". Facebook. November 27, 2009. <<http://www.facebook.compress/info.php?statistics>>.
- Fielding, T. Roy and Richard N. Taylor. "Principled Design of the Modern Web Architecture". ACM Transactions on Internet Technology (TOIT). May 2002: 115-150.
- Gamma, Erich, et al. Design Patterns: Elements of Reusable Object-Oriented Software. Massachusetts: Addison Wesley Longman, 1995.
- Grimson, W.E.L. and J.L. Mundy. "Computer Vision Applications". Communications of the ACM. March 1994: 44-51.
- Hadid, A, et al. "Face and Eye Detection For Person Authentication in Mobile Phones". Distributed Smart Cameras (ICDSC). September 2007: 101-108.
- Heseltine, Thomas and Nike Pears and Jim Austin. "Evaluation of image pre-processing techniques for eigenface based face recognition". International Conference on Image and Graphics. 2002: 677-685.
- Kerris, Natalie and Simon Pope. "Apple Announces Over 100,000 Apps Now Available on the App Store". Apple Inc. November 4, 2009. <<http://www.apple.compr/library/2009/11/04appstore.html>>.
- Krazit, Tom. "ARMed for the living room". CNET News. April 3, 2006. <<http://tinyurl.com/p8g3n5>>.
- Landré, Jérôme and Frédéric Truchetet. "Optimizing Signal and Image Processing Applications Using Intel Libraries". QCAV 2007. May 2007.
- Malik, Om. "Moore's Law Reconsidered." Business 2.0 Magazine. April 3, 2007. <<http://tinyurl.com/yzr7l9p>>.

- Mark, Dave and Jeff LaMarch. Beginning iPhone 3 Development : Exploring the iPhone SDK. California: Apress, 2009.
- Niwa, Yoshimasa. "Using OpenCV on iPhone". Yoshimasa Niwa. March 14, 2009. <<http://tinyurl.com/cqedlk>>.
- Perez, Sarah. "Photo Tagger: Facial Recognition for Auto-Tagging Facebook Photos". Read Write Web. July 21, 2009. <<http://tinyurl.com/l9uexf>>.
- Schmitz, Patrick. "Inducing Ontology from Flickr Tags". Collaborative Web Tagging Workshop. May 22-26, 2006.
- Trucco, Emanuele, and Alessandro Verri. Introductory Techniques for 3-D Computer Vision. New Jersey: Prentice Hall, 1998.
- Turk, Matthew and Alex Pentland. "Eigenfaces for Recognition". Journal of Cognitive Neuroscience. 1991: 72-86.
- Voila, Paul and Michael Jones. "Rapid Object Detection using a Boosted Cascade of Simple Features". IEEE Conference on Computer Vision and Pattern Recognition. 2001: 512-518.
- Whisenhunt, Phillip. "Porting OpenCV to the iPhone". iPhone/Cocoa/Objective C/ Biometrics/Everything. February 7, 2009. <<http://tinyurl.com/yhkoqoc>>.
- Whittke, Michael, et al. "Activity Recognition Using Optical Sensors on Mobile Phones". Mobile and Embedded Interactive Systems (MEIS). September 2009: 2181-2194.

# Appendix-A

## A.1 Cross-compiling OpenCV 2.0

- I. Create a staging folder called '/Staging' anywhere on the machine.
- II. Download and unpack the OpenCV 2.0.0 source library (located on Sourceforge). Move the unpacked OpenCV-2.0.0 source folder into /Staging. Create two new directories in the source folder labeled build\_simulator and build\_device.
- III. Copy the 'configure\_opencv' and 'cvcalibration.cpp.patch\_opencv-2.0.0' shell scripts from the OpenCV/Staging folder of the deliverable CD package into the directory /Staging that was created in step II. Ensure that the system root has read and write access to these scripts.
- IV. Navigate to /Staging/OpenCV-2.0.0 and type:

```
$ patch -p0 < ../cvcalibration.cpp.patch_opencv-2.0.0'
```

- V. Navigate to /Staging/OpenCV-2.0.0/build\_simulator and type:

```
$ ../../configure_opencv
$ make
$ make install
```

- VI. Navigate to /Staging/OpenCV-2.0.0/build\_device and type:

```
$ ARCH=device ../../configure_opencv
$ make
$ make install
```

Unfortunately, this process creates two types of static libraries: one to be used within the iPhone simulator (/Staging/opencv\_simulator), the other to be used on the iPhone device (/Staging/opencv\_device). These compilations are for i386 and ARM processors respectively. Because of the inconvenience of having to manually swap static libraries whenever the underlying environment was changed, the 'lipo' command will be used to create a universal static library composed of both i386 and ARM compilations.



## VII. Navigate to /Staging folder and type:

```
$mkdir universal_build

$ lipo -create opencv_simulator/lib/libcv.a opencv_device/lib/libcv.a -output
universal_build/libcv.a

$ lipo -create opencv_simulator/lib/libcxcvcore.a opencv_device/lib/libcxcvcore.a -output
universal_build/libcxcvcore.a

$ lipo -create opencv_simulator/lib/libcvaux.a opencv_device/lib/libcvaux.a -output
universal_build/libcvaux.a

$ lipo -create opencv_simulator/lib/libml.a opencv_device/lib/libml.a -output
universal_build/libml.a

$ lipo -create opencv_simulator/lib/libhighgui.a opencv_device/lib/libhighgui.a -output
universal_build/libhighgui.a
```

The /universal\_build folder will contain 5 static libraries (\*.a) for both Intel and ARM processors<sup>1</sup>.

If for any reason the build did not work, there is a working compiled version on the deliverable CD located at :

```
/OpenCV/Staging
```

---

<sup>1</sup> Instructions to cross compile OpenCV have been referenced from Niwa: ¶5-¶10

## A.2 Configuring X-Code

- I. Create a new folder on your desktop titled 'OpenCV.lib' and copy all the \*.a files from the universal\_build directory into /OpenCV.lib.
- II. Create another folder on your desktop titled 'headers' and copy all of the .h & .hpp files from the original OpenCV-2.0.0 source folder into it.
- III. Create a new application in X-Code.
- IV. Drag the /headers directory to the /Classes group of the project's "Groups & Files" explorer.  
When prompted, select 'Copy items into destination group's folder'.
- V. Include some header files such as cv.h and highgui.h in the code (for compilation purposes).
- VI. Now that X-Code has created a project folder structure, navigate to it's root via the system finder and copy the OpenCV.lib into it.
- VII. Double click the projects 'Target' and select the 'build' tab.
- VIII. Add paths to the static compiled libraries in the 'Other Linker Flags' section (all as one string) for both 'Release' and 'Debug' Configurations :

```
-cclib -lstdc++  
OpenCV.lib/libcv.a  
OpenCV.lib/libcvaux.a  
OpenCV.lib/libcxcvcore.a  
OpenCV.lib/libhighgui.a  
OpenCV.lib/libml.a
```

- IX. Notice that libstdc++ was added as well. Since OpenCV.lib was copied to the root of the project in step VI. , the paths are relative to the static library files. We can now compile and run<sup>2</sup>.

If for any reason the configuration did not work, there is a working configured version on the deliverable CD located at :

```
/Application/Look
```

---

<sup>2</sup> The X-Code configuration process has been referenced from Whisenhunt: ¶2, ¶15,

### A.3 Exporting Eigenspace as XML

The Application/FaceSpace directory, located on the deliverable CD, is subdivided as follows (Table A1):

Location	Subdirectories/Files	Description
/Database	/Sx where x is a digit.	Contains the model database that is used to create a face space. To add a new entry to the model, create a new folder /Sx where x is the next sequential digit in the database. All photos within the folder must be 92x112 greyscale 72dpi and in PGM format.
/Face Locator	/facedetect.c	Sample code that was used to test face finding techniques. The program locates a face within a image, crops it and saves it as C:/tmp.jpg. The program also displays the original photo with a red rectangle superimposed on the face that was located.
	/images	Sample images used to test the face finder algorithm.
/Recognition	/eigenface.c	Two parameters may be run the program :  - "train" - generates an face space using the images found within the /Database directory. The train.txt file specifies which image directories will be used to generate the face space. The XML file is saved as /output/facedata.xml  - "test" - projects a test face whose path is defined in test.txt.

**TABLE A1-FaceSpace Directory**

To generate a face space:

- I. Ensure the model database is correct and all photos within the /Database folder are 92x112 greyscale 72dpi and in PGM format.
- II. Configure /Recognition/train.txt to reflect which images within the model database will be included in the face space.
- III. Run eignface.c with the parameter 'train' via the command line or an appropriate development environment. The outputted face space will be located at /Recognition/output/facedata.xml
- IV. For convenience, the facedata.xml resides within the root of the project directory. To update a face space, simply drag the new facedata.xml into the *Look* workspace directory (Application/Look on the deliverable CD). When asked if you wish to overwrite, click 'Yes'. Alternatively, the facedata.xml can be uploaded to a web-server (see instructions below).

A flag variable located in the Utils.m class named 'loadXmlDataFromServer' can be toggled to yield the following two scenarios (Table A2).

loadXMLDataFromServer	Description
0	The facedata.xml will be loaded locally from within the workspace.
1	The facedata.xml will be loaded from a web-server at startup. Once loaded, it will be saved to the local workspace and replace any existing facedata.xml files that currently reside there.

**TABLE A2-loadDataFromServer Flag**

If the flag is set to true (1), then the url to the facedata.xml must be specified within the Util.m class:

```
NSString *const faceDataXmlUrl = @"URL TO FACEDATA.XML";
```

A representative face for each database model instance should be uploaded to a Flickr account.

The Flickr tag for each representative is assigned in the train.txt file during face space creation. For example, the following lines in train.txt specify '2' as a tag to images corresponding to the same model instance.







```
2 ../Database/s1/1.pgm
2 ../Database/s1/2.pgm
2 ../Database/s1/3.pgm
2 ../Database/s1/4.pgm
2 ../Database/s1/5.pgm
2 ../Database/s1/6.pgm
2 ../Database/s1/7.pgm
2 ../Database/s1/8.pgm
2 ../Database/s1/9.pgm
2 ../Database/s1/10.pgm
```

In other words, images 1.pgm through 10.pgm are all the same person (tag '2'), just under different lighting conditions or facial expressions as shown in Fig. A1.



**FIGURE A1-**Images representing a single model instance

When an image is uploaded to Flickr representing the above person, it should be tagged '2'. Fig. A2 demonstrates the uploading of 6 representative images to the Flickr database-the provided tag and title information will be pulled from *Look* when a match is retrieved.

 Title: <input type="text" value="Roth"/> Description: <input type="text"/> Tags: <input type="text" value="6"/>	 Title: <input type="text" value="Paul"/> Description: <input type="text"/> Tags: <input type="text" value="5"/>	 Title: <input type="text" value="Adam"/> Description: <input type="text"/> Tags: <input type="text" value="4"/>
 Title: <input type="text" value="Alex"/> Description: <input type="text"/> Tags: <input type="text" value="3"/>	 Title: <input type="text" value="Meg"/> Description: <input type="text"/> Tags: <input type="text" value="1"/>	 Title: <input type="text" value="Terry"/> Description: <input type="text"/> Tags: <input type="text" value="2"/>

**FIGURE A2-**Flickr metadata

To configure *Look* to use a specific Flickr account:

- I. Launch the *Look* workspace by double clicking *Look.xcodeproj* within the Application/Look folder.
- II. Navigate to MainViewController class found within the /Controllers folder of the Groups & Files view. Change the static instance variables FlickrAPIKey and FlickrUserId to ones that are associated with your account (obtained at [www.Flickr.com](http://www.Flickr.com)).

```
NSString *const FlickrAPIKey = @"your Flickr API Key";  
NSString *const FlickrUserId = @"Your Flickr User Id";
```

To simplify the configuration, the paper has provided a Flickr account which currently holds representatives 1-6 of Figure A2. The FlickrAPIKey and FlickrUserId are already configured within the *Look* workspace. Flickr credentials for database maintenance are as follows:

- I. Flickr username: "Look\_Application"
- II. Flickr password: "Carleton12#"

# Appendix-B

## B.1 CD Deliverable Contents

Location	Subdirectories/Files	Description
/Application	/Look	<p>The iPhone application <i>Look</i>. Directly altering this folder structure is not recommended. All changes should be through X-Code. To open up the <i>Look</i> X-Code project, double click <i>Look.xcodeproj</i>.</p> <p>Requirements : Mac OS X / XCode with iPhone SDK</p>
	/FaceSpace	<p>Contains all of the code that is used to create a face space as described in the 'Server Design' section.</p> <p>Requirements: Windows XP/Vista/7 / MS Visual Studio or C++ environment variant.</p>
/OpenCV	/Staging	A completed cross-compiled static OpenCV library. The folder contains the two scripts that are needed to cross compile the OpenCV libraries for use with the iPhone SDK.
/Media	/Look_Instructional_Video.m4v	<p>An instructional videos that shows <i>Look</i> in action.</p> <p>Requirements: VLC / Windows Media Player</p>
/Report	Mobile_Face_Recognition.pdf	<p>A copy of this report.</p> <p>Requirements: Adobe Acrobat</p>
	Mobile_Face_Recognition_Proposal.pdf	<p>The original proposal.</p> <p>Requirements: Adobe Acrobat</p>

**TABLE B1-Deliverable CD Contents**

## B.2 Software Requirements

- I. OpenCV libraries (2.0).
- II. X-code & iPhone SDK.
- III. MS Visual Studio or C++ development environment variant.

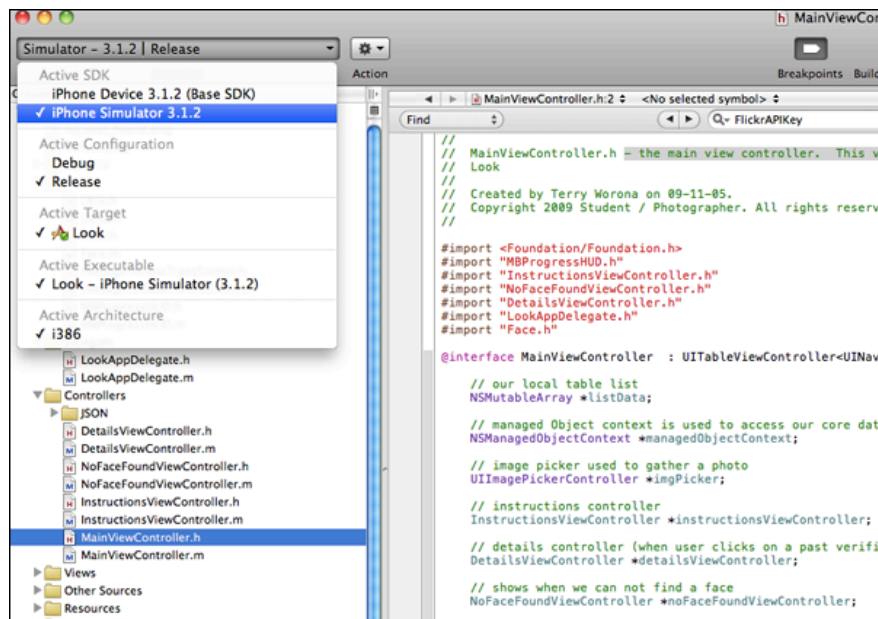
## B.3 Hardware Requirements

- I. Macintosh Computer (OS X 10.5 or later).
- II. Windows Computer (Windows XP/Vista/7).
- III. iPhone (3G or 3Gs) or iPod Touch.

## B.4 Deploying *Look* to the iPhone Simulator

The following steps must be followed to run *Look* on the iPhone simulator:

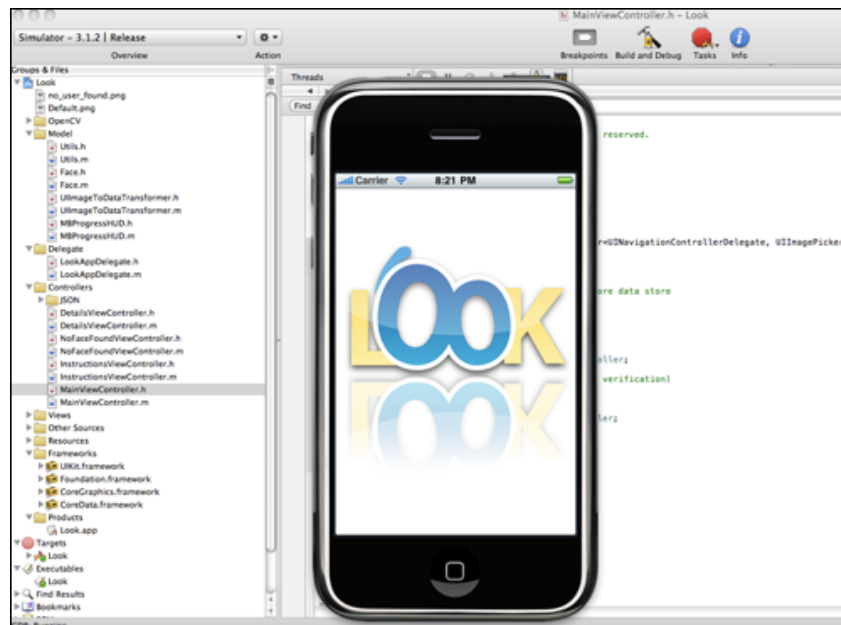
- I. Launch X-Code via double clicking *Look.xcodeproj*.
- II. The X-Code workspace will already be configured as per the steps outlined in 'Cross Compiling OpenCV 2.0'. The static OpenCV libraries are also located within the workspace.
- III. Select the dropdown menu in the top left corner and select 'iPhone Simulator' as the active SDK and 'Release' as the active configuration (Fig. B1).



**FIGURE B1**-iPhone Simulator Runtime Configuration



IV. Select Project->'Build & Debug' and *Look* will launch in the simulator environment (Fig. B2).



**FIGURE B2-** *iPhone Simulator Running Look*

## **B.5 Deploying *Look* to the iPhone Device**

As a result of Apple's strict policy on installing third party applications, the deployment of *Look* to the iPhone device is more much more involved than the simulator. Please refer to the 'provisioning' instructions within the iPhone developer portal at <http://developer.apple.com/iphone/manage/overview/index.action>.

## **B.6 Instructional Video**

Also packaged with the deliverable CD is an instructional video which takes the user through all the actions of *Look*. The video is located in the /Media folder of the deliverable CD as well as on youtube via the following URL:

[http://www.youtube.com/watch?v=RaGBt4hRh\\_s](http://www.youtube.com/watch?v=RaGBt4hRh_s)