INSTITUTE OF TECHNOLOGY TRALEE

Determining the Feasibility of Facial Verification Technologies in a Point-of-Sale Payment System

Darren Keane

B.Sc. (Hons) Computing with Software Development

Primary Supervisor: Mr. Owen Mackessy

Secondary Supervisor: Mr. Billy Stack

Acknowledgements

First and foremost, I would like to thank my supervisor Owen Mackessy for his support throughout the academic year. His advice, suggestions and feedback were of great benefit and enhanced both this thesis and the accompanying software implementation.

Secondly, I would like to acknowledge the efforts of the project supervision pool. Their advice and assistance throughout the duration of this project did not go unnoticed.

Finally, I would like to thank my family and friends for their support not only this year, but throughout the past four years. This project would not have been possible without them.

Abstract

Facial recognition technology has had significant research attention in recent years. This research has led to advancements in both two-dimensional photometric techniques and three-dimensional geometric techniques.

This study evaluated the effectiveness of eigenfaces, a photometric facial recognition algorithm, in the field of user verification. The primary motivation for this thesis was to examine the possibility of applying facial recognition as an alternative method of user verification in point-of-sale payment systems.

A prototype system was implemented during this study to analyse the eigenfaces algorithm. Particular interest was given to the ability of depth data, retrieved from a Microsoft Kinect sensor, to reduce background noise in source imagery. Faces which were present within these images were extracted and normalised using digital imaging techniques. They were then processed using the eigenfaces algorithm.

Analysis of the experimental data showed successful recognition results ranging from seventy to ninety percent under ideal environmental conditions. External factors, primarily lighting, had a notable impact upon the results of the algorithm. A worst case scenario produced recognition results with a success rate of less than twenty percent. The results suggested that photometric recognition technologies are not suitable for application as a user verification methodology in point-of-sale payment systems.

Table of Contents

Table of Figures

Table of Tables

Table of Code Snippets

Declaration of Originality

I hereby certify that this document titled *Determining the Feasibility of Facial Verification Technologies in a Point-of-Sale Payment System,* and the included prototype, are of my own creation and any information which was sourced from elsewhere has been referenced.

Darren Keane

____

Darren Keane

1st May 2015

Chapter 1: Introduction

Facial recognition can be applied as a biometric substitute to passwords, and other means, in the field of user verification in computer systems. User verification is important – it is often implemented to protect access to a user's personal information such as bank accounts or email correspondence. As security breaches and password thefts at reputable companies, including Google and Apple, have become common, it is important to investigate other means of authentication.

This thesis will examine the feasibility of applying facial recognition techniques as a means of verification. Specifically – the application of photometric recognition methods in point-of-sale payment systems will be analysed. To accomplish this goal, a literature review was undertaken in three key areas: i) facial recognition, ii) imaging hardware and iii) payments systems and verification methods.

Facial recognition is a field in computer science and a large amount of research has been completed in this area, particularly since the 1970s. In general, facial images are first extracted from source imagery using object detection methodologies such the Viola-Jones Framework. Then, either photometric (appearance based) or geometric (feature based) algorithms may be applied to create a numerical representation of the face. Recognition is achieved by comparison with a facial database. Facial recognition systems have been applied at airports around the world and recently, Facebook claimed to have developed an algorithm with results which are comparable to human recognition levels.

Imaging hardware is an important component of a facial verification system – they provide the source data upon which the system can function. Factors such as image quality must be considered. Image capture may be achieved using either webcams or IP cameras which are now available in high definition specifications. This thesis will give particular attention to the Microsoft Kinect. Its array of sensors provide a multitude of information. In this context, its colour images and related depth data are of particular interest.

Payment systems, like all areas in technology, have evolved in recent years. Point-of-sale systems which traditionally utilised a centralised architecture are today moving to cloud based models. The development of PayPal revolutionised online payment systems and allowed businesses and individuals to electronically accept payments directly into their bank accounts. Similar services such as Stripe and Braintree have followed, permitting merchants to build their payment system using any one of a variety of platforms. Verification methods presently used in point-of-sale and online payment services include PIN numbers and password verification. Recent advancements by companies such as Apple have introduced biometric verification techniques that perform fingerprint analysis.

Following the literature review, the Methodology and Project Specification chapters will outline the means by which this project will be developed. An agile software development methodology will be used – the developed use cases, sprint plan and test plan are discussed. This projects software architecture was split into a number of components which included a front-end software system, a photometric facial recogniser, a facial database and a high speed cache. In terms of imaging hardware, a Microsoft Kinect sensor was utilised. Its depth sensor was applied to reduce noise in source imagery.

The implementation chapter will outline the programmatic approach that was taken to develop the described architecture. It will show the methodologies used to achieve both facial detection and facial recognition using C# and GNU Octave, along with the implementation of the facial database for which CouchDB was utilised and the cache for which Redis was utilised. When completed, the system will capture a user's facial image, located within a specified distance from the Kinect sensor, analyse it and compare to a database to ascertain if a match exists.

Experimental data will be collected to analyse the viability of the implemented recognition algorithm. From the evaluation of this data the research question will be answered and recommendations for future work and research will be made.

Chapter 2: Facial Recognition

2.1 Introduction

Facial recognition, in computing terms, is the means of recognising and validating a person's face by means of analysing either a still image or a video frame captured from a digital source. Facial recognition has numerous uses, ranging from biometric verification to tracking individuals within a crowd. At a high level, the primary means of achieving facial recognition is by comparing a source facial image against a facial database to determine if a match exists. According to Lin, facial recognition has been an active area of research since the 1970s – it has shown increasing levels of accuracy as the techniques employed have developed while having the advantage of being relatively unobtrusive. (Lin, 2000)

The process of facial recognition can be divided into two distinct stages:

1. Extraction of facial images from a given source image,
2. Determination of an individual's identity by analysing these facial images.

This chapter will discuss some of the approaches that have been taken to solve the problem of facial recognition. We will first discuss an algorithmic approach that has shown a high level of success in facial detection. Then we will review methodologies that utilise either photometric or geometric techniques to determine an individual's identity. These facial detection and recognition algorithms have encountered several challenges that have hindered their success levels. Nonetheless, computer vision is today an active research area and a wide variety of software implementations, which we will see in Section 2.6, have been developed that utilise both facial detection and recognition techniques.

2.2 Facial Detection Algorithms

It is necessary to provide a biometric verification system with source data before it can operate. In the area of facial recognition this data is provided to the system in the form of an image. The location of all faces present in the given image must first be attained by utilizing a facial detection algorithm. Recognition algorithms can subsequently be applied to the detected faces following extraction from the source image. Various facial detection algorithms have been developed. These include:

- Detection using Edge-Orientation Matching,
- Detection using the Hausdorff Distance,
- Detection using Weak Classifier Cascades.

A cascade of weak classifiers is the most common algorithm now used in facial detection. A cascade classifier, following excessive training, can produce accurate facial detection results. The breakthrough in using cascade classifiers occurred in 2001 with the publication of the Viola-Jones Object Detection Framework. (FaceDetection, 2015) It is this method that will be employed during the implementation of this project and thus the only method which will be discussed in detail here.

2.2.1 The Viola-Jones Object Detection Framework

The Viola-Jones framework is a methodology for detecting objects in images, the development of which was primarily motivated by the area of facial detection. It was published by Paul Viola and Michael Jones. (Viola & Jones, 2001) Their framework employs feature based Haar Cascade Classifiers to determine whether the sought object exists within a given image. The cascade classifier must be trained to detect objects by using a large set of positive images (those which contain faces) and negative images (those which do not contain faces) that are scaled to the same size. There are three distinct areas within the Viola-Jones framework: Haar-like features, the integral image and the cascade classifier.

2.2.1.1 Haar-like Features

Instead of directly analysing an entire image, Viola and Jones decided to examine rectangular features within the image. These are called Haar-like features. Their naming originates from their similarity to Haar wavelets which are an area in mathematics. The Haar-like features are used to calculate the difference in intensity values between adjacent rectangular groups of pixels. Features can be scaled and repositioned to allow for the detection of features of any size within an image. *Figure 1* below shows the four Haar features that were used by Viola and Jones.

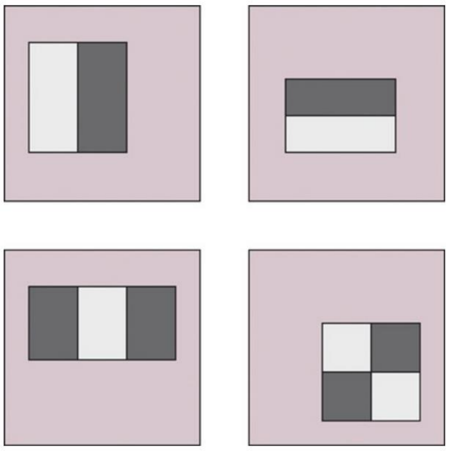


Figure 1 The Haar-like features used by Viola and Jones

Each Haar-like feature is represented as a single value. This value is calculated by subtracting the sum of the intensity values of the pixels under the light rectangle from the sum of the intensity values of the pixels under the dark rectangle. The calculated value for a feature is compared to the classifiers training data to determine if that feature is present within the image area being analysed. (Tech Radar, 2010)

Figure 2 shows Haar-like features overlaid on a typical training image. This clearly shows the application of the chosen features based upon the knowledge that human faces share similar features. One can observe that the first feature calculates the difference between the eyes and the upper cheeks, exploiting the observation that the area around the eyes is generally darker. The second feature compares the eyes and the bridge of the nose. (Viola & Jones, 2001)

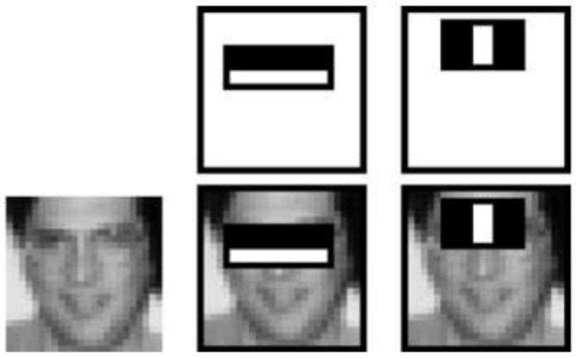


Figure 2 Haar-like features overlaid on a facial image

2.2.1.2 The Integral Image

The rectangular feature values are calculated using an intermediate image called the integral image – this is an array which contains the sum of the pixels intensity values to the left and above a pixel at location (x, y) in the source image. The integral image technique is applied to improve the computational efficiency of calculating Haar-like features throughout an image. It allows feature values to be calculated in *constant time*.

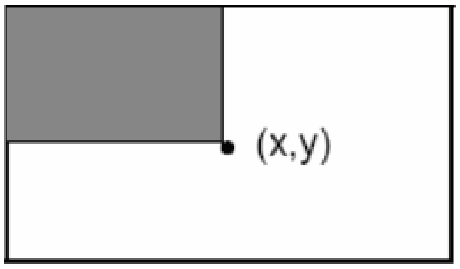


Figure 3 The area in dark is the integral image at the point x, y

Using the integral image, the sum of the intensity values for a rectangle of any size within the original image can be found. From here, the value of a Haar-like feature can be calculated by finding the difference in intensity values between connected rectangles. Feature calculation is an efficient operation – all features, regardless of size, are calculated in the same manner.

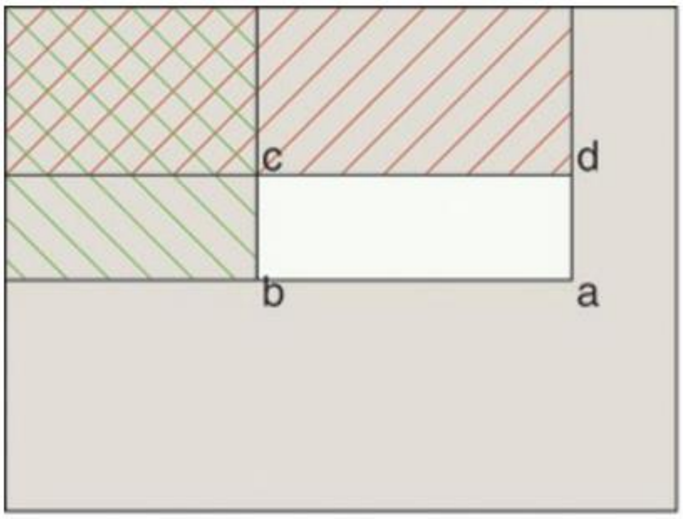


Figure 4 Calculating intensity value using integral images

Figure 4 shows the calculation of an intensity value for the rectangle $abcd$. This example calculation can be achieved by taking the values that represent the points a, b, c and d in the integral image array and applying the formula: $(a - b - d) + c$

2.2.1.3 The Cascade Classifier

The calculated feature values are used as the input to a cascade of classifiers. The cascade is a decision tree which contains a variable number of stages. Each stage is trained to detect the expected object (faces in this instance) and reject all other objects. The number of features used to detect an object is increased at each stage. Viola and Jones trained each stage of their classifier using an artificial intelligence technique called AdaBoost. AdaBoost allowed for a strong classifier to be trained based on a large set of weak features, each of which are assigned a weight value. (Tech Radar, 2010)

The first stage of the classifier utilizes a small number of features. The number of features are increased at each stage. Every sub window of the input image is tested against the first stage of the classifier. If it passes that stage it is tested against the second stage, and so on. A sub window that passes all stages of the classifier is determined to contain a facial image. A higher number of stage iterations produces more accurate results. *Figure 5* on the next page provides a visual representation of the functionality of a cascade classifier.

Viola and Jones demonstrated a facial detection accuracy rate of up to 95% using a 32 stage cascade classifier and a total of 4,297 features. Furthermore, they showed that their method could be up to 15 times faster than any previous detection algorithm, primarily through the use of the integral image technique that was used to calculate feature values. (Viola & Jones, 2001)

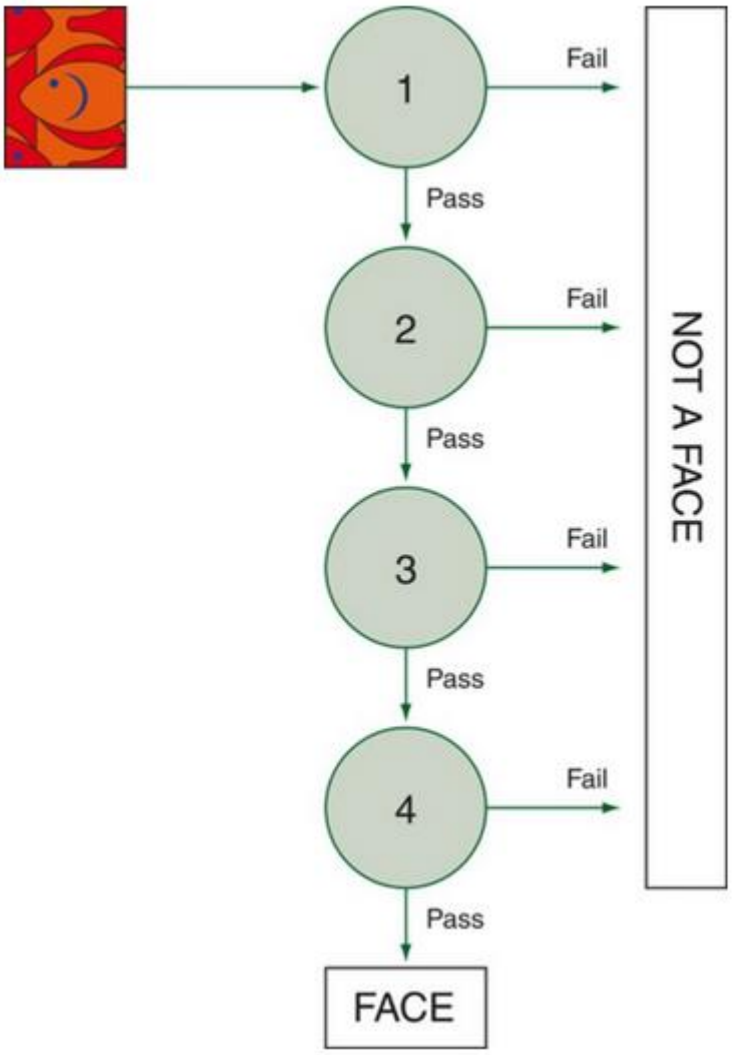


Figure 5 The functionality of a cascade of classifiers

2.3 Facial Recognition Methods

There are many facial recognition algorithms, each of which are required to analyse data in various forms. These facial recognition methods can be divided into two distinct groups – appearance based or feature based.

2.3.1 Appearance Based Methods (Photometric)

Appearance based methods, also referred to as photometric methods, typically operate on two dimensional facial images. The pixel data within an image is analysed and then compared to that of other facial images to find a closest match. The eigenface algorithm is amongst the most prominent photometric facial recognition methods. This method has been shown to be produce accurate and reliable results when analysing frontal facial images under constant lighting conditions. It is, however, sensitive to pose and illumination.

A later developed photometric recognition method known as the fisherface algorithm attempted to address these issues. It works with a larger data training set to the eigenface method, allowing for greater analysis of variations in pose and illumination. As a result it is not only more complex that the eigenface method, but also more computationally expensive. (Woodman, 2007)

2.3.2 Feature Based Methods (Geometric)

Feature based or geometric facial recognition methods use a set of geometric feature points that measure and compare distances and angles of a facial image. Woodman described a method by Starovoitov which used 28 facial points on a two dimensional image and produced accurate results in cases where there were no variations in pose. (Woodman, 2007) Recognition accuracy was found to decrease when the pose of the subject varied.

A relatively modern technique called Elastic Bunch Graph Matching overlays a grid onto a facial image. A number of grid points, placed at pre-defined feature locations, are used to determine feature vectors. A matching algorithm is applied to these vectors to find reliable facial matches.

Geometric feature points have been applied to three dimensional facial models and have been shown to be more reliable than using two dimensional images. Three dimensional representations are more dependable and allow for considerable variations in pose.

2.4 Facial Recognition Algorithms

A sizeable number of facial recognition algorithms have been developed during research in this area and are continually developing today. Some commonly applied algorithms include Principal Component Analysis (eigenfaces) and Linear Discriminate Analysis (fisherfaces), both of which are appearance based (or photometric) processes. Elastic Bunch Graph Matching is a prevalent feature based method of facial recognition.

2.4.1 Principal Component Analysis

According to Smith, Principal Component Analysis (PCA) was first invented by Karl Pearson in 1901 and was later further developed and titled by Harold Hotelling during the 1930s. (Smith, 2002)

PCA is a statistical procedure that has attained significant application in facial recognition. It is a technique for finding patterns in data of high dimension. PCA can be used to uniquely express the given data and consequently find contrasts and comparisons between sets of data. It is a powerful analysis tool that allows data to be represented in a mathematical manner. Following the analysis of the patterns in a given set of data using the PCA methodology, the data can be compressed by reducing the number of dimensions without losing much information. The remaining data is essentially the principal components (the identifying attributes of that data) and they account for the majority of the variance in the observed dataset. The principal components can be used as predictor variables in subsequent analysis. (O'Rourke & Hatcher, 2013)

There are six distinctive steps required to perform Principal Component Analysis on a given set of data, as detailed by Smith. The application of these steps in facial recognition will be detailed in Section 2.4.2. These steps are:

1. Acquire the data to be analysed,
2. Subtract the mean of the data from each of the data dimensions,
3. Calculate the covariance matrix,
4. Calculate the eigenvectors and eigenvalues of the covariance matrix,
5. Perform dimensionality reduction by choosing the number of eigenvectors to retain and storing them in a feature vector,
6. Represent the original data in terms of the retained eigenvectors.

2.4.2 Eigenfaces

The idea of using Principal Component Analysis in computer vision software was first approached by Sirovich and Kirby in 1987. (Sirovich & Kirby, 1987) They generated a set of basis features called *eigenpictures* by applying PCA to a set of images. Their work was further expanded the Turk and Pentland who in 1991 introduced the eigenface method. (Turk & Pentland, 1991) Their system classified faces by analysing the significant features within a set of known facial images. They referred to these significant features as *eigenfaces* as they are the principal components (or eigenvectors) of the given set. An unknown face can be recognised using their method by calculating and then comparing its eigenface value to those of known individuals and finding the closest match.

It is important at this point to note that the terms eigenvectors and eigenfaces are used interchangeably in this document when discussing facial recognition. An eigenface, by definition, is an eigenvector that when displayed has a face like appearance.

2.4.2.1 Training the Recogniser

The following is a mathematical outline of the process of calculating eigenfaces for a given data set and is based upon the published work of Turk and Pentland.

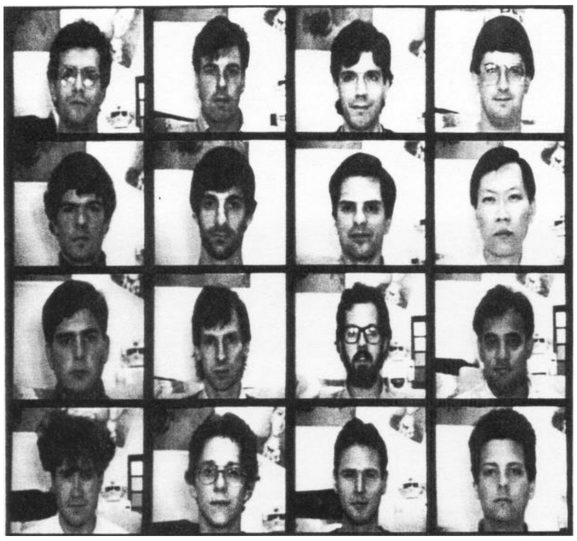


Figure 6 A sample of a training set of facial images

Adhering to Principal Component Analysis, the first step required in the calculation of eigenfaces is to acquire a training set of M facial images. Let each facial image be represented as T_i. *Figure 6* above is a sample of the training set used by Turk and Pentland. Each image in the set should be of the same dimensions, containing N number of pixels. Every image must be converted to a column vector. The images are then represented as a matrix of size $M\ x\ N$. Each column in this matrix represents a separate image in the training set and each row contains the corresponding pixel value at that point in each image.

The next step of PCA is to calculate the mean of this set of images. The pixel values for each row are summed and then divided by M, the total number of images in the set. The formula to find Ψ, the average facial image, is shown.

$$\Psi = \frac{1}{M}\sum_{n-1}^{M} T_n$$

To complete this step, the average facial image must be subtracted from each image in the original set of training faces. The result of this calculation for the i^{th} face in the training set is stored in the i^{th} column of the matrix Φ.

$$\Phi_i = T_i - \Psi$$

Subjecting this set to Principal Component Analysis seeks a set of M orthonormal eigenvectors vectors that best describe the distribution of the data. An orthonormal vector is a vector which is both orthogonal (perpendicular) and a unit vector. (Smith, 2002) To achieve this the covariance matrix C must be calculated.

$$C = AA^T$$

Where the matrix A = $[\Phi_1\Phi_2 . . . \Phi_M]$.

However, this covariance matrix has a dimensionality of $N^2 x\ N^2$. Computationally, this is a very inefficient task when working with typical image sizes, especially as most of the eigenfaces will not be useful. To overcome this, the following approach is used. If the number of images in the set is less than the dimensionality of the set ($M < N^2$), then there will be $M - 1$ (as opposed to N^2) meaningful eigenvectors. This is because any further eigenvectors have an associated eigenvalue of zero. (Turk & Pentland, 1991) The covariance matrix can be recalculated as follows.

$$C = A^T A$$

This will produce a matrix with dimensionality of $M\ x\ M$ which is more much computationally efficient.

The eigenvectors and eigenvalues of this covariance matrix are calculated. The eigenvectors which have the highest corresponding eigenvalues, and thus which best represent the data, are maintained. In the case of Turk and Pentland, they generally retained a number of eigenvectors which equalled about 10% of M, the number of images in the training set.

These eigenvectors v are of dimension M. The eigenfaces u_l of the original set of training images can be determined by returning these eigenvectors to the dimension N^2 using the following formula:

$$u_l = \sum_{k=1}^{M} v_{lk}\Phi_k \quad l = 1, \ldots\ldots, M$$

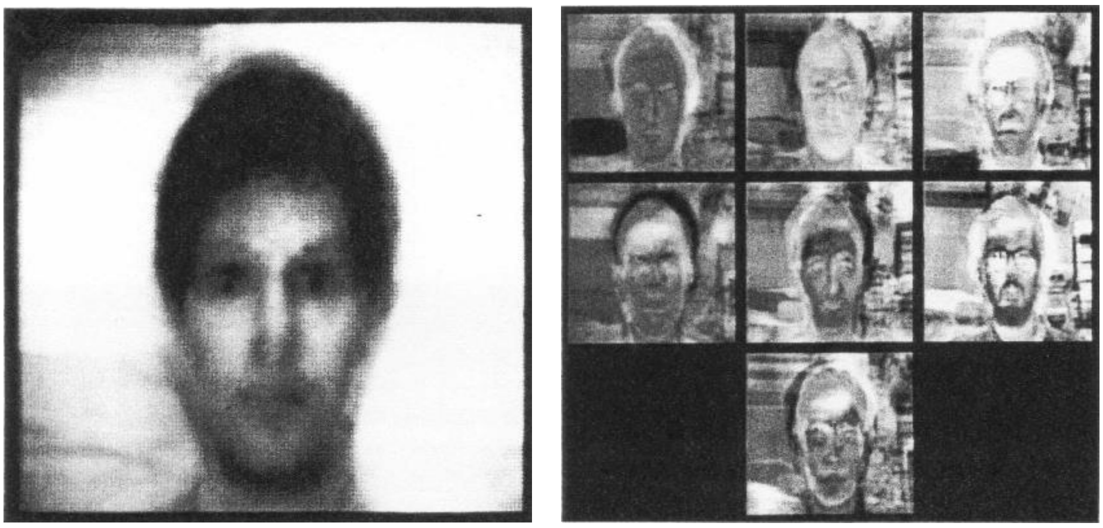


Figure 7 Average face and a sample of calculated eigenfaces

2.4.2.2 Recognising an Unknown Face

Recognition using eigenfaces requires a weight vector to be determined for each image in the training set. It is determined by multiplying each image with the calculated eigenfaces.

$$\Omega_k = u\,\Phi_k$$

To recognise an unknown image, T, its weight must also be calculated.

$$\omega = u(T - \Psi)$$

The closest facial match to the unknown face in the training data is that from which it has the shortest Euclidean distance.

$$\epsilon_k = \sqrt{||(\omega - \Omega_k)||^2}$$

Finally, Turk and Pentland considered the use of a threshold value. This is a Euclidean distance value. If the determined minimum distance of the unknown face from the training data is less than this value then it is considered to be a face. Otherwise, it is likely not a face. (Turk & Pentland, 1991) A threshold value can be determined by processing non-facial images using the above recognition methodology and finding the average of their minimum Euclidean distances.

2.4.3 Fisherfaces

Fisherfaces are based upon a method of dimensionality reduction called Linear Discriminant Analysis (LDA); more precisely the method published by Ronald A. Fisher in 1936 titled *Fisher's Linear Discriminant Analysis*. Contrasted with PCA which maximises the total variance of the given data, LDA instead maximises the ratio of between-class scatter to within-class scatter. (Wagner, 2012)

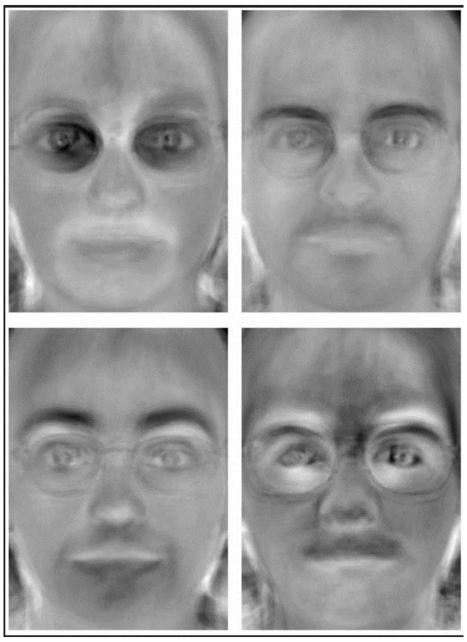


Figure 8 Sample of calculated fisherfaces

Figure 8 shows four example fisherfaces calculated by applying the algorithm to a set of training data. (Scholarpedia, 2014)

The fisherface algorithm considers the ratio of the variation between the facial images of one subject and that of another subject. The algorithm obtains feature vectors from an unknown facial image and then attempts to recognise the given face by comparing it to a set of predetermined feature vectors. (Hyung-Ji Lee, 2001)

An algorithmic overview of fisherfaces, as described by Wagner, follows. (Wagner, 2012) It was decided to not discuss this algorithm in detail as it will not be applied during the implementation phase of this study.

1. Create a matrix of column vectors containing the training images, with a corresponding vector storing their classes,
2. Project the image matrix into the subspace whose dimensions are specified by the number of training images minus the number of unique classes,
3. Determine the between-class scatter of the projection,
4. Determine the within-class scatter of the projection,
5. Apply LDA and maximise the scatter ratio – this calculates eigenvectors from which those with a non-zero eigenvalue are retained,
6. The fisherfaces are attained by finding the product of the results of steps 2 and 5. Sample fisherfaces are shown in *Figure 9* below.

An implementation of this algorithm by Wagner, on a dataset of celebrity facial images, had a correct recognition rate of 90%.

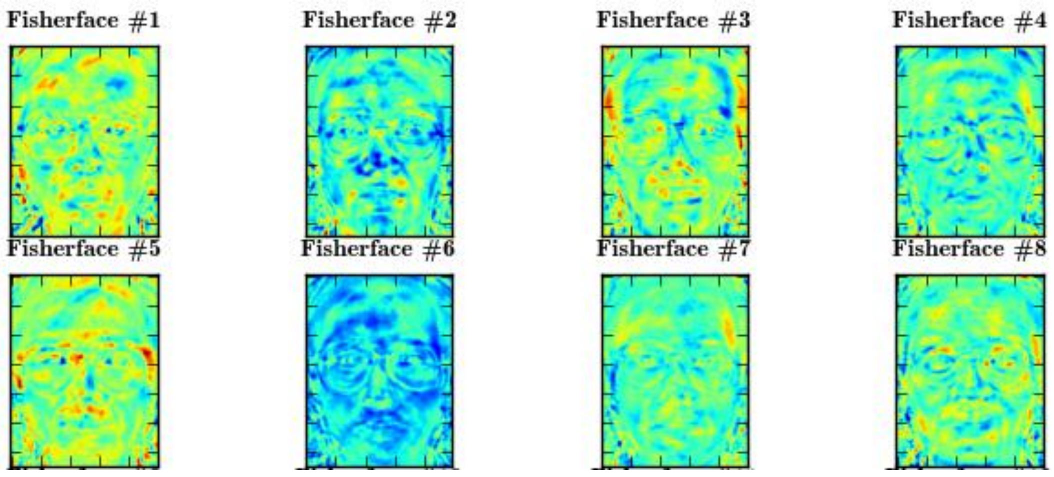


Figure 9 Fishfaces calculated by Wagner, with a heatmap applied

According to Belhumeur et al., experimental results comparing the fisherface method to the previously discussed eigenface method produced some interesting results. (Belheumer, et al., 1997) Notably, both algorithms performed well when lighting is frontal. Also, they found that removing the first three principal components calculated by the eigenface method (as they are primarily related to lighting conditions) resulted in better performance. They claimed that the fisherface method may have lower error rates and also require less computational resources than the eigenface method. The reliability of the eigenface methodology improves as more principal components are retained. Belheumer et al. found that this improved performance, however, becomes negligible at around 45 principal components.

2.4.4 Elastic Bunch Graph Matching

Elastic Bunch Graph Matching (EBGM) is a facial recognition methodology where faces are represented using labelled graphs. The graphs are based on a Gabor wavelet transform. Faces are extracted using the elastic graph matching process and are subsequently compared using a similarity function.

A face is represented using a graph of N nodes that are connected by E edges. Each node is located at a landmark on the face (called a fiducial point). Nodes include the pupils, the nose and the ears, amongst other identifying facial features. Each edge connects two nodes.

A jet is a representation of a local texture. Jets are extracted from each fiducial point of the graph. They are based on a wavelet transform. According to Wiskott et al., “a jet is defined as the set of 40 complex Gabor wavelet coefficients obtained for one image point.” (Wiskott, et al., 1997)

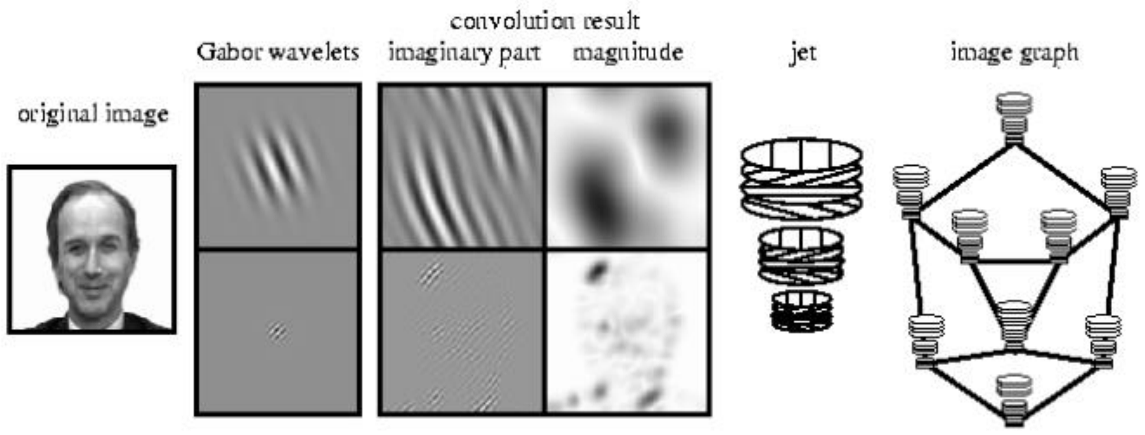


Figure 10 Creating an image graph for a source image

The first set of graphs must be generated manually. That is, nodes are defined at fiducial points while edges are defined between these nodes. This is called a Face Bunch Graph (FBG). Using the FBG, graphs for new facial images can be automatically determined using EBGM. The FBG may require manual corrections, but at a size of approximately 70 graphs automatic matching is reliable.

Figure 11 below shows the FBG. It is a general representation of a face. The stacks at each node represents a bunch of jets. For an unknown face, jet values are calculated at each fiducial point. Then, the best suiting jets from the FBG are selected as matches. In this case, the dark grey jets represent the best matches at each fiducial point.

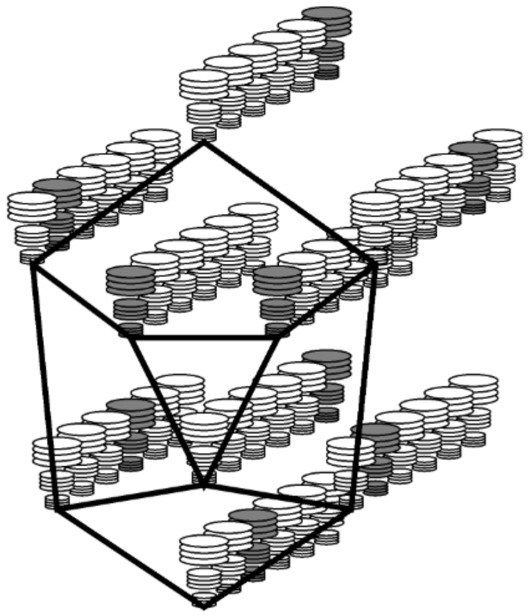


Figure 11 The face bunch graph

Recognition requires relatively little computational resources. Each facial image in the database is already represented as a graph. A match can be determined by comparing the graph established from the unrecognised face with those in the database. The database graph with the greatest similarity is determined using pairs of corresponding jets.

Wiskott et al. reported recognition rates of up to 90% when testing the EBGM based face recognition process that they designed. (Wiskott, et al., 1999)

2.5 Challenges to Facial Recognition

Variations in facial images present a significant challenge to facial recognition and is one of the most difficult problems in this research area. Images taken in a natural environment could, for instance, have a complex background and drastic variances in lighting conditions. Shang-Hung Lin, in his 2000 paper, categorized the following as some of the possible image variations. (Lin, 2000)

- Camera distortion and noise,
- Complex background,
- Illumination,
- Translation, rotation, scaling and occlusion,
- Facial expression,
- Makeup and hair style.

Lin outlined some developments that have been made in overcoming these challenges. For instance, established image enhancement methods can be applied to overcome the illumination challenges. These include histogram equalization – a method of contrast adjustment used to improve intensity distribution in an image. Rotation along both the X and Y axes can cause occlusion of the face and make it unsuitable for recognition – as shown in *Figure 12* below.

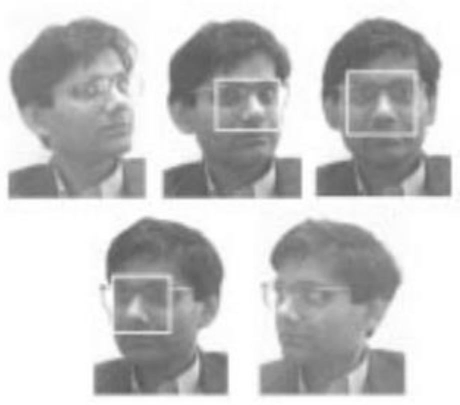


Figure 12 Rotation along X and Y axis impeding facial detection

Fluctuations in expression and hairstyles can be overcome by using a significant facial region as opposed to the whole face – as shown by the marked facial regions in *Figure 12*. This excludes the mouth, ears and hair while allowing the face to remain recognisable.

Similar to Lin, Peter Belhumeur identified variations in illumination conditions amongst the most critical challenges to facial recognition technologies. (Belhumeur, 2006) He noted the success of photometric based recognition algorithms, but emphasised the importance that "the face has been previously seen under similar circumstances." This issue may be minimalized by storing a large number of images of an individual that capture variations in both pose and illumination conditions. *Figure 13* shows the same individual under a variety of illumination conditions.

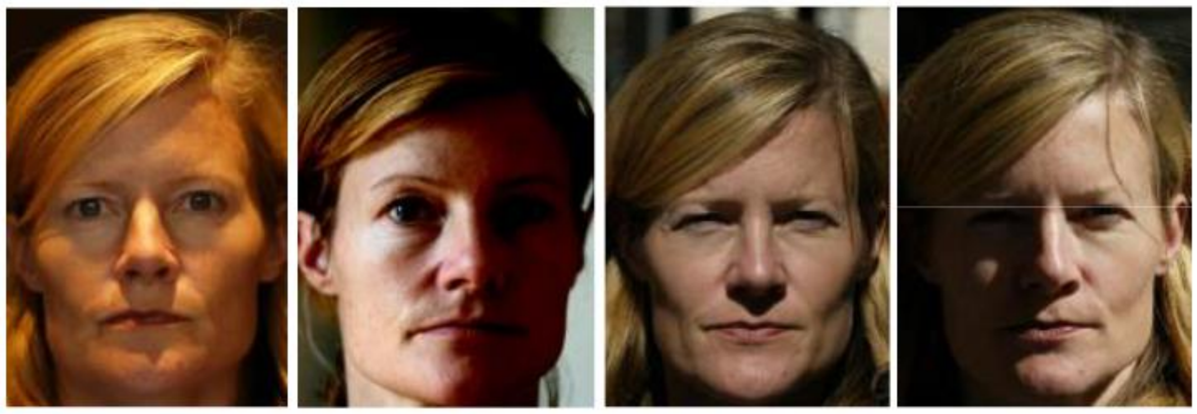


Figure 13 The same person photographed under various illumination conditions

Finally, Ramchandra and Kumar, in their research of facial recognition challenges, outlined a number of requirements that increase the likelihood of a successful facial match being found. (Ramchandra & Kumar, 2013) Their requirements included:

- Capturing sharp and clear images,
- Using colour neutral images,
- Normalising brightness and contrast,
- Capturing images where the eyes should be open and clearly visible,
- Capturing images where the eyes should not obscured by glasses.

2.6 Facial Recognition Applications

Facial recognition has been applied in many software solutions – far too many to discuss here. The concept is an ongoing and active research area that continues to evolve to this day. Existing facial recognition implementations have been used to predict and avoid bottlenecks at airports, to verify an individual's access rights to a computer system and to aid investigators in solving crimes.

2.6.1 Facebook (DeepFace)

Facebook has employed facial detection techniques for a number of years, most notably in the *tagging* feature that is applied to automatically highlight facial images in photographs uploaded by users of the social network. More recently, the company published a research paper outlining the development of a facial recognition project called DeepFace. The paper claims that DeepFace can determine if two facial images are of the same person with an accuracy of 97.25%. This claim puts it very close to human level recognition – which according to Facebook has an accuracy of 97.53%. (Taigman, et al., 2014)

Facebook trained its DeepFace system on a large dataset containing 4.4 million images that represent 4,030 people. The system utilises deep learning techniques. It applies a neural network (a software model that represents a biological nervous system and its connections) that is nine layers deep and involves more than 120 million parameters.

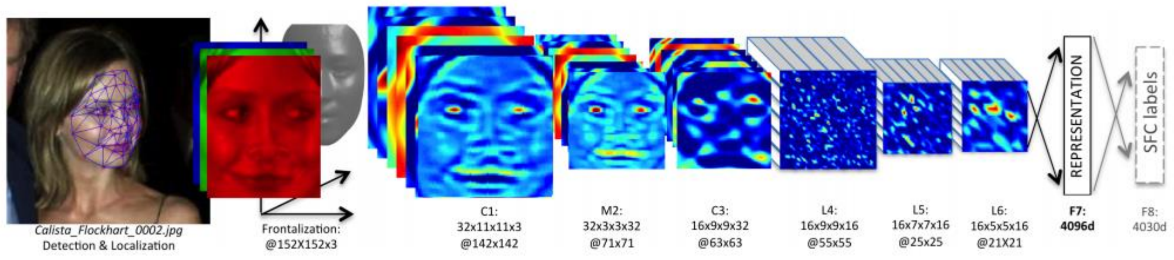


Figure 14 DeepFace architecture - how it "sees" a face

Many existing facial recognition approaches analyse and compare physical features using two dimensional images. Facebook took a different approach with DeepFace. It builds a three dimensional model of a face using a process called normalization. The face can then be mathematically represented and compared against existing faces in in the facial database.

2.6.2 MFlow (Human Recognition Systems)

MFlow is a facial detection and recognition system designed and developed by Human Recognition Systems. MFlow is a passenger management application developed for airports and airlines whose goal is to reduce queue times for passengers. The system works by capturing an image of a passenger's face as they enter the terminal, analysing it and storing a numerical representation of that face in a database. The passengers face is then tracked by an array of cameras on their journey through the airport. Using analysis techniques the software can find and attempt to predict potential bottlenecks, especially at downstream areas such as security checks. Airport staff are notified and can respond, for instance, by opening extra lanes and diverting passengers to the shortest queues. This system has been deployed at various airports in the United Kingdom including Belfast, Gatwick and Manchester. (Human Recognition Systems, 2014)

2.6.3 KeyLemon

KeyLemon are a Swiss company that offer biometric solution to developers and manufactures in various industries. Their technologies includes facial recognition, voice recognition and motion tracking. They aim to develop technologies that can be integrated into any project. Their technology has been applied from the medical to transport industries and includes semiconductor embedded software solutions. KeyLemon's technologies have been designed to recognise humans from biometric features and reduce the reliance on traditional passwords. The relevance here is unquestionable – amongst KeyLemon's biometric solutions is a utility that allows users to access their personal computer using facial verification. Companies such as Fujitsu, Toshiba and Panasonic have worked with KeyLemon's technologies. A Rest API is offered to developers to allow them to access KeyLemon's available services and wrappers have been developed for a variety of programming languages. (KeyLemon, 2014)

2.6.4 EvoFit

The Gardaí unveiled a new facial recognition technology in early 2015 named EvoFit. The technology is used to identify suspects during criminal investigations. It is reported to have a success rate of up to 74 per cent. It is used to allow witnesses and victims of crimes who are helping Garda investigations to identify persons of interest by selecting facial photographs from a database that best represent the person they saw. Multiple images may be selected which are then combined into one single image. This image is processed against up to 60 other facial databases to produce the most accurate possible depiction of the sought individual. The databases available, according to the Conor Lally of the Irish Times, include “facial images for both male and female suspects covering all ethnic groups, age brackets and other features.” (Lally, 2015) It is a modern advancement to the technique of employing a sketch artist to create a depiction of persons of interest.

Chapter 3: Imaging Hardware

3.1 Introduction

In terms of biometrics, particularly the area of facial verification, imaging hardware is employed to retrieve raw image data of those who are requesting authentication. This data can have many different types and formats. Facial images must be extracted from the captured data utilising a detection algorithm. Then, it is these images that are passed to the recognition algorithm to determine if a facial match exists.

Devices such as webcams directly connected to the verification system or IP cameras that are streaming across the network may be utilised to capture raw image data. A brief overview of these technologies will be given. However, this chapter will focus on the Microsoft Kinect sensor which will be employed for data capture during the implementation phase of this project. The Kinect sensor provides data in numerous forms, the most relevant of which are its colour images and depth sensor data.

The image quality delivered by devices utilised in facial verification is of importance – a higher quality image allows for improved facial detection and, particularly in the case of photometric algorithms, improved facial recognition. The devices which will be discussed can provide images up to full high definition resolution. The Kinect, in particular, allows for images to be captured using both a variety of resolutions and image formats.

3.2 Webcams

Webcams are imaging devices that provide a live video stream to the computer they are connected to, generally via a USB interface. Webcams may also be built directly into the hardware of the computer, such as in the vast majority of modern laptops. The intended use of webcams is to provide a video stream to software applications which subsequently transport the video data across a network, for example, in video chat applications like Skype.

The first webcam dates back to 1991 when researchers at Cambridge University developed a means of monitoring a coffee pot in the science department using a camera that was directly connected to their computers. They took this idea further in 1993 by transporting the cameras video data across the internet, thus creating the world's first webcam. (Devaney, 2014) Webcams became prominent in the early 2000's as video chat grew in popularity with the widespread uptake of high speed internet connectivity. Today, they have many uses and remain popular on social media platforms.

From the perspective of a biometric verification system that utilises facial images, a webcam could be employed to capture source imagery of an end user. This could be achieved by capturing a frame from the cameras video stream or by capturing an image directly in a photographic manner, if such functionality is supported by the camera. Webcams are advantageous due to their ubiquity; an end user would likely not be required to purchase any additional hardware. KeyLemon, discussed in Section 2.6.3, utilise webcams in their verification software that analyses biometric facial data to provide access to personal computers.

According to Alan Henry of Lifehacker.com, numerous webcams which provide high definition imaging quality have reached general availability and affordability. (Henry, 2012) The Logitech C920, for example, can stream data in 1080p and utilises an autofocusing sensor. It can capture still images at up to 15 megapixels in resolution. Similar cameras are available from Microsoft, including the LifeCam and LifeCam HD, which offer image resolutions of up to 720p.

3.3 IP Cameras

IP cameras are imaging devices that directly stream video data across a network. They are accessed from a remote computer through a network interface. This interface may be either hardwired via Ethernet or provided wirelessly using Wi-Fi technology. IP cameras are commonly employed in surveillance systems and offer a degree of adaptability as they can be utilised in any location once communication is available with a network access point.

The world’s first IP camera, the AXIS 200, was developed by Axis Communications and released for general purchase in 1996. The camera contained an integrated web server which allowed remote access to its video stream. The cameras purpose was to be utilised in remote monitoring and surveillance systems. (Axis Communications, 2014)

The Dropcam Pro is a particular IP camera that has obtained critical acclaim. It provides a variety of features that include full high definition video as well as night vision technology by utilising infra-red LEDs. It can be accessed and configured from a remote device via an integrated web interface. The Dropcam Pro delivers a 130 degree field of vision and an 8x digital zoom. It applies end to end SSL encryption to all transported data, a feature desirable not only in security systems but also within the scope of verification technologies that utilise biometric data. (Ha, 2014) IP cameras have qualities which are suitable in a facial verification system – the provision of remote access to their image data allows for the cameras to be placed at advantageous locations with their only requirement being the establishment of a connection to the local network.

Figure 15 The Dropcam Pro IP Camera

3.4 Microsoft Kinect

The Microsoft Kinect, codenamed as "Project Natal", was first announced during Microsoft's keynote at the annual E3 conference in 2009. The Kinect was first launched for Microsoft's Xbox 360 gaming console in November 2010. A Windows SDK followed in February 2012. (Zhang, 2012) The Kinect is a device aimed at transforming human computer interaction. It provides multiple data streams in the form of colour images, depth images, infra-red images and voice sensing. The technology has seen applications in gaming as well as research applications in industries such as education, healthcare and transportation.

There have been two major versions of the Kinect hardware. The original release, version 1, is compatible with the Xbox 360 gaming console. Version 2 was released alongside Microsoft's Xbox One gaming console in 2013. Both versions later added support for the Windows operating system through software updates.

3.4.1 Kinect Version 1

3.4.1.1 Overview of Functionality

The Kinect version 1 hardware, shown in *Figure 16*, contains a colour camera, a depth sensor and a four microphone array. The device can perform three-dimensional motion capture, voice recognition and facial tracking.

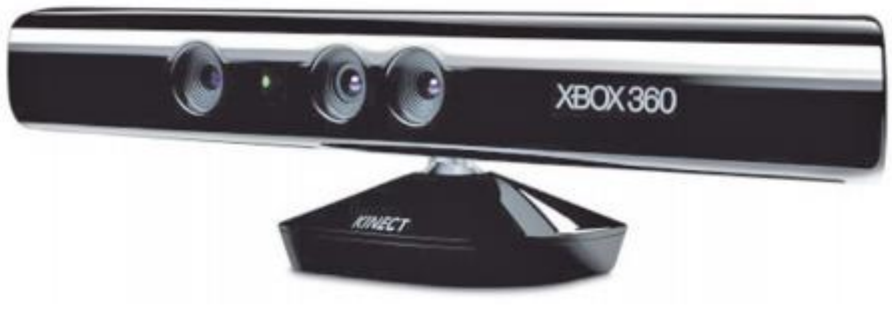


Figure 16 Microsoft Kinect Version 1 (Zhang, 2012)

The depth sensor consists of an IR projector along with an IR camera. The camera is a CMOS sensor. Microsoft have licensed the depth sensing technology from an Israeli company called PrimeSense and much information on its underlying technology isn't publicly available. It is known that the IR projector passes the light from an infra-red laser through a diffraction grating. This produces a set of infra-red dots on the visible surfaces which can be "seen" by the camera. A three-dimensional depth image can be constructed using triangulation by matching the dots in the captured depth image with those in the projector pattern. (Zhang, 2012) The depth map produced can be deciphered using its shading. For instance, in a depth map encoded with grey pixels, the darker a pixel is then the closer its location is to the camera.

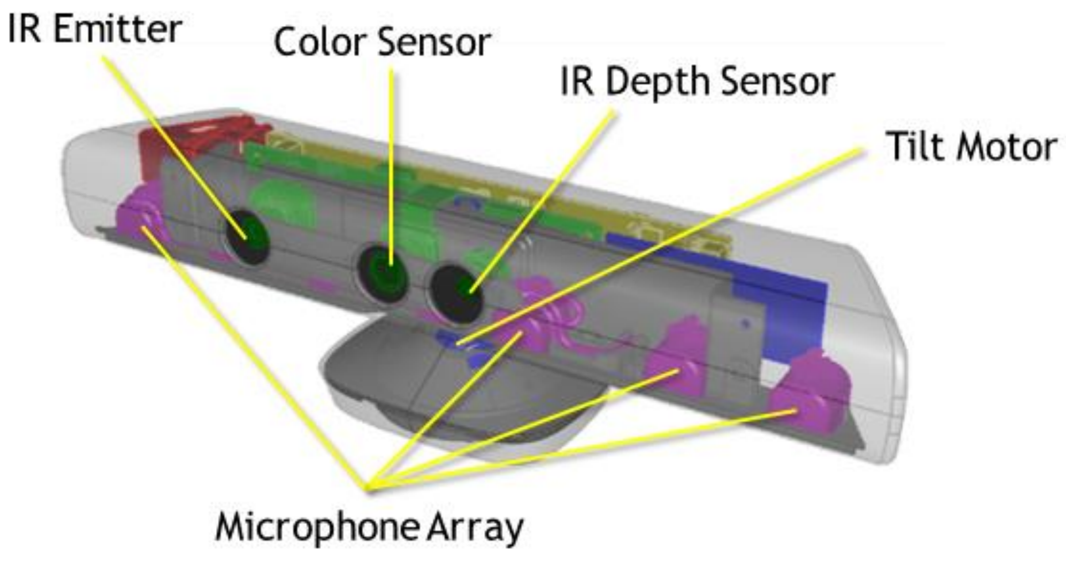


Figure 17 Kinect Version 1 hardware breakdown

Amongst the most innovative functions of the original Kinect hardware was its skeletal tracking functionality. This had to be performed in real-time using the relatively limited computing power of an Xbox 360 gaming console. In skeletal tracking a human body is represented by a number of joints located at points between body parts. These include the head, shoulders, arms and knees. Microsoft introduced a technique called “per pixel, body part recognition” to implement skeletal tracking with Kinect. The skeletal tracking algorithm that was developed can process a frame in under 5ms on the gaming consoles hardware. This provides a theoretical rate of 200 frames per second for skeletal tracking using a Kinect sensor.

The Kinect can capture both two-dimensional colour and three-dimensional depth images. This allows much flexibility in the implementation of facial tracking functionality. Microsoft developed a tailored algorithm to implement three-dimensional facial tracking using the Kinect.

Facial tracking uses up to 100 two-dimensional points which represent the co-ordinates of a feature within the facial image. These are the 87 points which are shown in *Figure 18* along with another 13 which include the centre of the eyes, centre of the nose and corners of the mouth. According to Microsoft, “the X, Y, and Z position of the user’s head are reported based on a right-handed coordinate system.” The Kinect captures three dimensional head pose by utilising three angles: pitch, roll and yaw. (Microsoft, 2014)

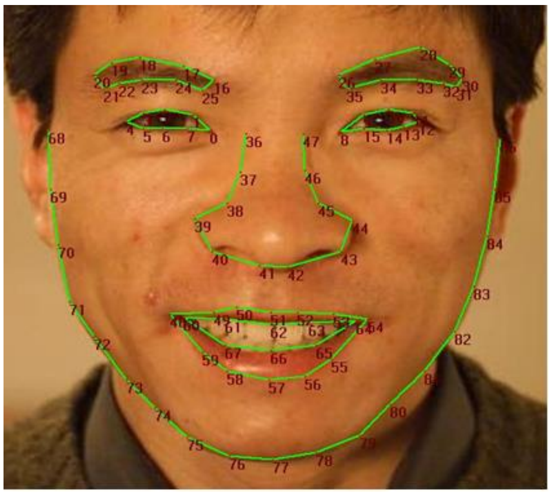


Figure 18 Kinect Version 1 2D Facial Tracking Points (Microsoft, 2014)

The Kinect colour and depth streams can provide data in a variety of resolutions and formats. The raw information provided by these streams can be utilised to produce valuable source data within the area of facial verification.

3.4.1.2 The Kinect Colour Stream

The colour stream is a succession of still images delivered from the sensor at a rate between 12 and 30 frames per second. Data can be provided by the colour stream in three different formats which can be specified programmatically: RGB, YUV and Bayer. These are shown in *Table 1*.

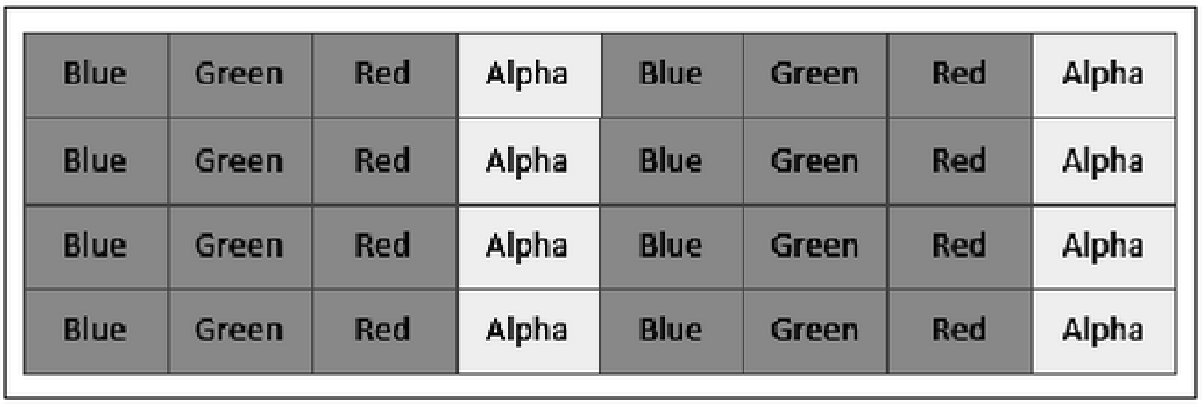
Kinect Colour Stream Data Formats

Format	*Resolution*	*Maximum FPS*
RGB	640 x 480	30
	1280 x 960	12
YUV	640 x 480	15
Bayer	1280 x 960	12
	640 x 480	30

Table 1 Colour Stream Data Formats (Microsoft, 2014)

The RGB Colour Stream Format

The first image format available is presented using the red-green-blue colour space known as RGB. Each RGB pixel in a colour stream frame is represented as a byte array of size four – the first three of these values are for the red, green and blue intensity values while the final array item, alpha, stores the transparency of the pixel. RGB data is provided by the sensor at 32 bits per pixel and can be specified to use two different resolutions at up to 30 frames per second. *Figure 19* shows a group of eight RGB pixels.



Blue	Green	Red	Alpha	Blue	Green	Red	Alpha
Blue	Green	Red	Alpha	Blue	Green	Red	Alpha
Blue	Green	Red	Alpha	Blue	Green	Red	Alpha
Blue	Green	Red	Alpha	Blue	Green	Red	Alpha

Figure 19 Representation of RGB (Red-Green-Blue) colour space

The Luminance YUV Format

The second image format is Luminance YUV. There are three channels – Y is the luminance channel, U is the blue channel and V is the red channel. YUV differs from RGB in how the colour space is represented. YUV data is returned from the sensor at 16 pits per pixel. Only a resolution of 640x480 is supported at a maximum of 15 frames per second. The YUV format colour stream is less memory intensive than RBG though it compromises on image quality. (Jana, 2012)

The Bayer Format

The third and final format provided by the Kinect is the Bayer colour image format. This format uses a combination of red, green and blue colour, represented in a pattern. The colour pattern is 50 percent green, 25 percent red and 25 percent blue. This pattern is called a Bayer filter. According to Jana, “the colour filter is used in this way because the human eye is more sensitive to green and can see more changes in green light.” (Jana, 2012) The Bayer format supports two resolutions with a maximum rate of 30 frames per second.

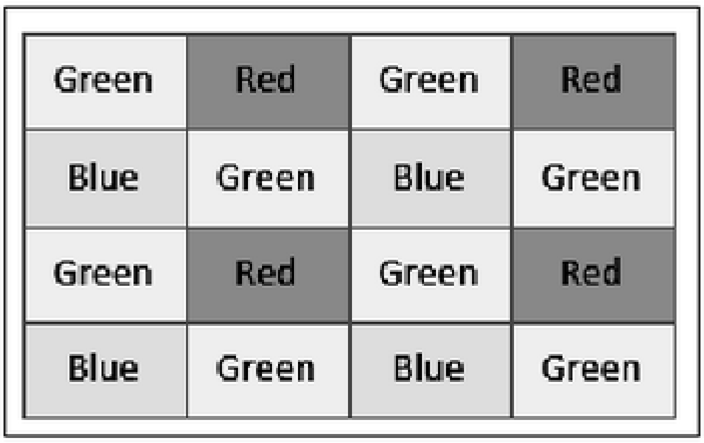


Figure 20 The Bayer filter pattern

The variation in data formats available from the colour stream allow applications to be developed using a diverse range of image resolutions and frame rates for which different levels of memory resources are required. In terms of biometric verification, source data with higher quality levels (or resolutions) are desirable.

3.4.1.3 The Kinect Depth Stream

The depth stream consists of a constant flow of depth frames produced by the Kinect, the contents of which indicate the distance of visible objects from the sensor. The depth configurations available are shown in *Table 2*.

Kinect Depth Stream Data Formats

Format	*Resolution*	*Maximum FPS*
Raw Depth Frame	640 x 480	30
	320 x 240	12
	80 x 60	15

Table 2 Depth Stream Data Formats (Microsoft, 2014)

As briefly discussed in the overview, the Kinect depth stream data is generated using a combination of an infra-red projector (or emitter) and an infra-red camera. The projector emits a pseudorandom pattern of invisible dots in front of the Kinect. These dots are read using the infra-red camera which determines depth information – that is, the distance between the surface from which the dots were reflected and the sensor. A high level view of this process is shown in *Figure 21*.

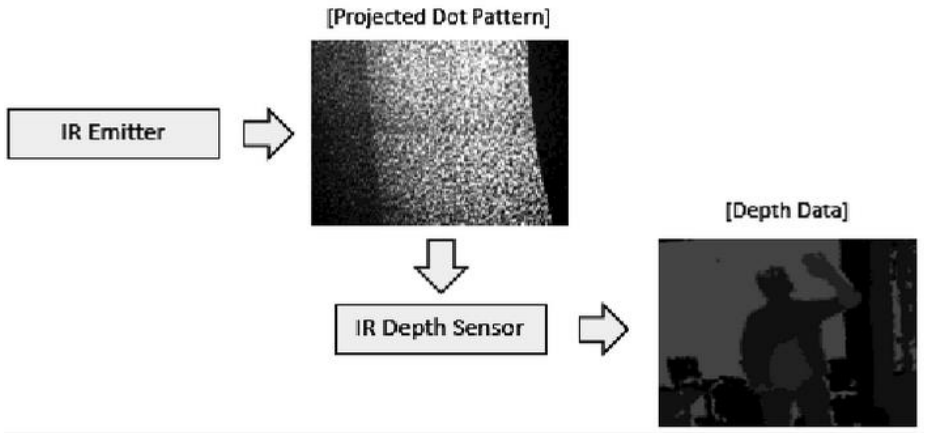


Figure 21 The production of depth data using the Kinect

This data is measured using a process called stereo triangulation. Defined by Jana, stereo triangulation is "an analysis algorithm for computing the 3D position of points in an image frame". This process requires two images to determine depth information. When determining depth using the Kinect, one of these images is that captured by the depth sensor. The other image used is invisible – it is the infra-red pattern that is emitted from the IR projector. These images are considered to correspond to different cameras and allow depth to be calculated using stereo triangulation. (Jana, 2012)

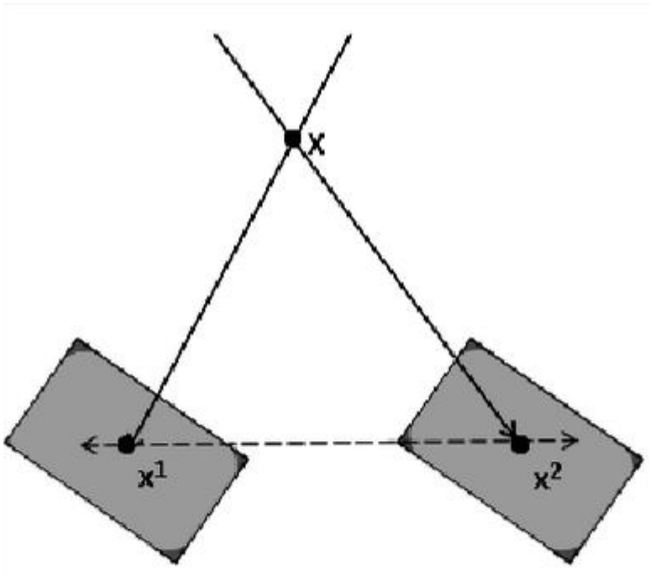


Figure 22 Stereo Triangulation calculating the depth at point x in the given scene

The Kinect sensor can calculate depth data from distances as close as 800mm to distances as far as 4000mm (2.6 to 13.1 feet). This is the sensors default range. The sensor can be reconfigured to near range mode which will instead capture data from as close as 300mm to as far as 3000mm from the sensor. It is possible to calculate depth data for further distances but the results of these calculations are considered unreliable.

Depth data is returned from the Kinect sensor in a 16-bit raw depth frame, as shown in *Figure 23*. The data uses one of the three available resolutions and has a maximum rate of 30 frames per second, as shown in *Table 2* above. The first three bits of the depth frame identify player data and will be ignored momentarily. The remaining thirteen bits represent the measured distance in millimetres.

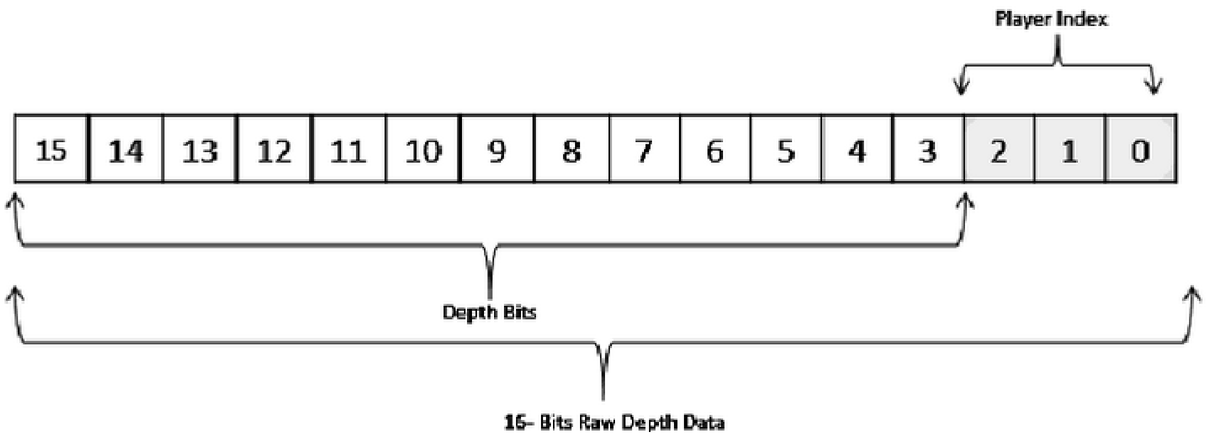


Figure 23 Visualisation of the 16-bit raw depth data frame

As the upper thirteen bits hold the depth value of the pixel, a bitwise shift operation must be applied to this data to move the bits to their correct position. The depth can then be calculated. This process is shown in *Figure 24* on the following page.

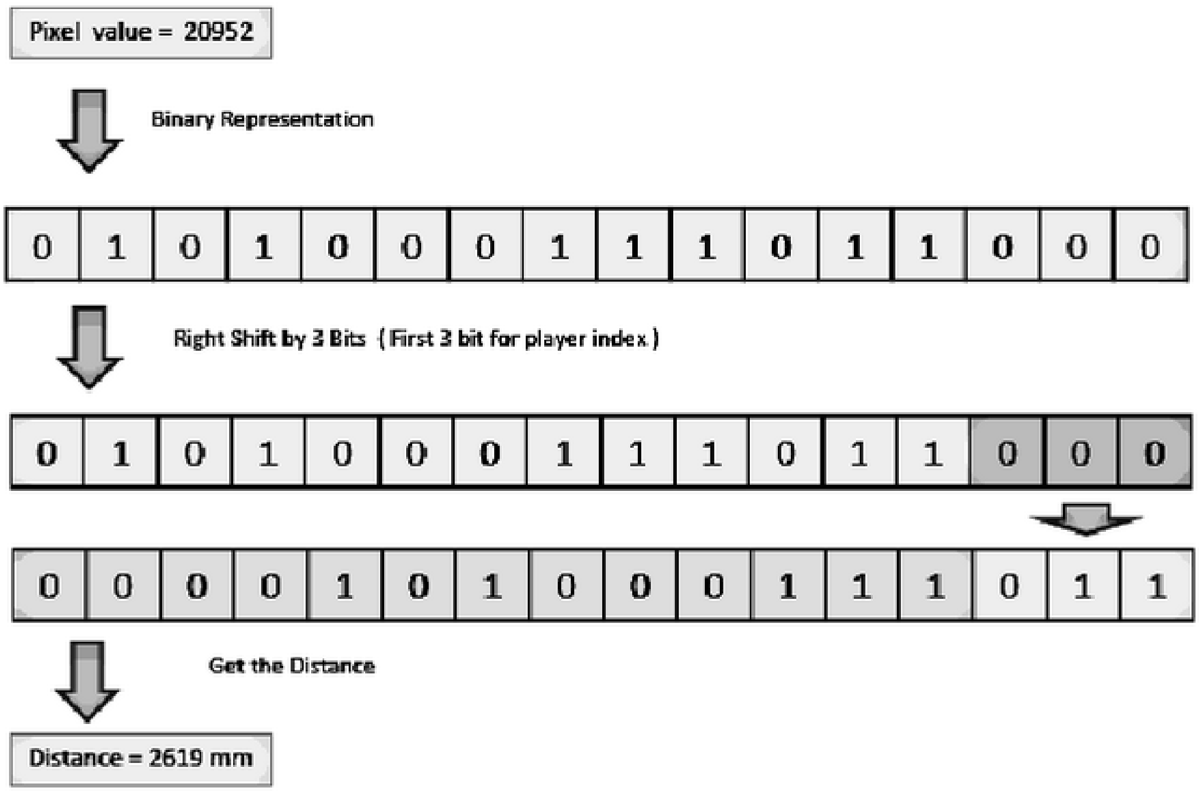


Figure 24 Visualisation of bitwise shift and depth calculation

This operation can be performed on any pixel to determine its distance from the sensor. The depth image can be converted to RGB format for a more colourful representation than its default grayscale format. Each pixel in the RGB frame will display a colour value based on distance. For example, objects that were closer to the camera will be displayed using a dark colour while objects which were further away will be displaying using a brighter colour.

The first three bits of the depth stream are referred to as the player index and are used to identify players. In terms of the Kinect, this is the number of people visible to the sensor. Persons in a depth frame can be tracked and their distance from the sensor determined from the corresponding depth data. This functionality is generally employed by game developers who utilise player motion tracking to control in-game characters.

While the raw data discussed above can be manually manipulated, the Kinect SDK provides pre-built implementations for much of the functionality.

The applications of the Kinect depth sensor go beyond two dimensions – the depth data can be used to create a three dimensional view. A three dimensional mesh can be constructed and the depth data processed to create a representation of an object, an example of which is shown in *Figure 25*. This functionality can be used to construct three dimensional images from captured objects or to identify three dimensional objects present in the image.

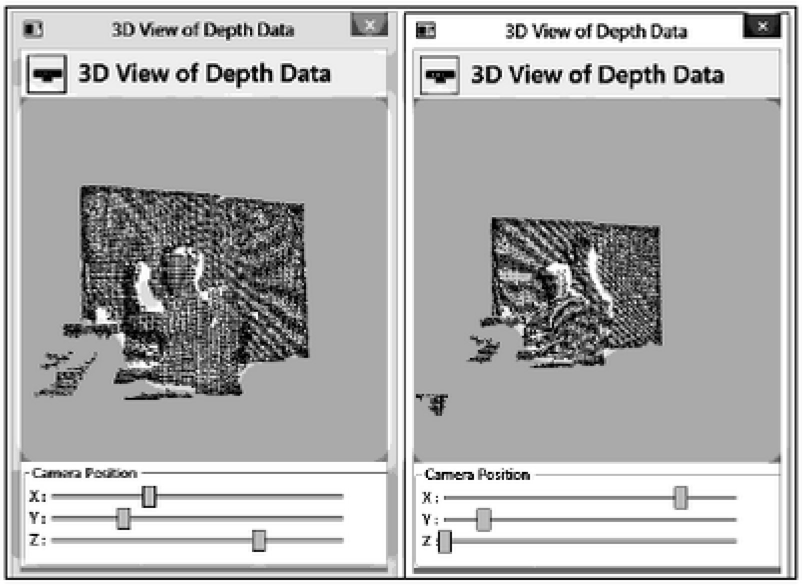


Figure 25 Example of three dimensional depth data

The depth data available from a Kinect sensor can be applied in facial verification. Its player tracking feature can ensure that an individual is present and not a static image, while the raw depth distances can be utilised to reduce a corresponding colour image by removing noisy background data that is outside a distance threshold.

3.4.2 Kinect Version 2

The second generation Kinect camera, Kinect version 2, was released in late 2013 alongside Microsoft's Xbox One gaming console. Similar to its predecessor, the sensor is available for use with both the console and with personal computers, in this case requiring the Windows 8 operating system or higher. The sensor is shown in *Figure 26*.

Figure 26 Kinect version 2 sensor (TechCrunch, 2014)

The sensors camera can capture colour video up to 1080p. The sensor can track up to six people, and 25 body joints for each. Compared to the original Kinect, Microsoft claims "the tracked positions are more anatomically correct and stable - and the range of tracking is broader." (Microsoft, 2014) Depth fidelity improvements are noticeable when compared to the version 1 sensor, as shown in *Figure 27*. The depth sensor supports resolutions up to 512 x 424 pixels.

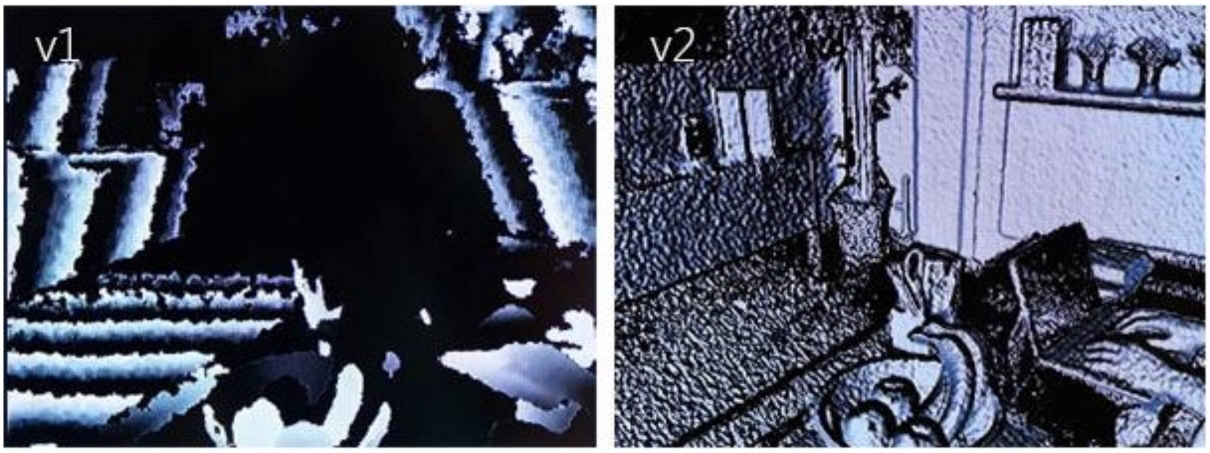


Figure 27 Comparison of depth sensing in Kinect Version 1 and Kinect Version 2

Version 2 of the Kinect provided new infrared capabilities that allowed the sensor to "see" in the dark. This has applications in machine learning and could be applied, relevantly, in analysing and recognising facial images under varied lighting conditions.

3.4.3 Software Development Kit

The Kinect Software Development Kit (SDK) was first released by Microsoft in June 2011. It is compatible with Windows 7 (or higher) and allows software developers to build applications that utilise the Kinect sensors in C++, C# or Visual Basic. The SDK provides developers with access to low level features of the Kinect – this includes access to the data streams (both the colour stream and the depth stream) in their raw format, if required. Other features of the sensor supported in the SDK include skeletal tracking and speech recognition.

The latest version of the Kinect SDK, at the time of writing, is version 2.0. It was released in October 2014. This version requires the Windows 8 operating system and supports only the newest model of the Kinect which was released in late 2013. (Microsoft, 2014) Kinect SDK 1.x versions are those that support the original Kinect sensor and remain available for Windows 7 and higher. The latest release, version 1.8, was published by Microsoft in September 2013.

Developers are given full access to the raw data that is returned from the Kinect. Programmatically, this data can be acquired using either an event model or a polling model – frames can be retrieved automatically using event handlers once they become available or in an on-demand basis by polling the sensor each time a frame is required. The raw data can be utilised as required. It allows one to not only display the image data but also to process it and apply a variety of effects. Colour and grayscale effects may be applied while brightness, contrast and saturation levels may be defined.

Chapter 4: Payment Systems and Verification Methods

4.1 Introduction

Payment systems are financial platforms that support the transfer of funds from payers to payees. They were traditionally paper based (cash, cheques) but have expanded rapidly into paperless mechanisms and electronic payments (credit cards, debit cards). Traditional point of sale systems that were centralised on one site are beginning to move to a cloud based model. The advent of online payment platforms such as PayPal during the last number of years has given developers an expansive number of options when it comes to the implementation of payments systems.

One of the most important aspect of any payment system is end user verification. The system must validate that the payee is who they claim to be and that they have provided the correct payment information. Traditional systems utilised signatures, and later PIN (Personal Identification Number) numbers for validation and verification. Both of these methodologies have fallen victim to fraud.

Modern payment gateways, such as Stripe, use encryption to protect sensitive payment information and are required by law to enforce strict PCI (Payment Cards Industry) standards. These services generally validate users by verifying a username and password combination.

Biometric based payment platforms have begun to grow recently. Both Apple and MasterCard have developed payment technologies that verify transaction by means of the payees fingerprint. Using unique biometric identifiers is a developing area and is not without its imperfections – Apples fingerprint analysis technology has been bypassed by security researchers.

4.2 Point-Of-Sale Systems

4.2.1 Traditional POS Systems

A traditional point-of-sale (POS) system is a network that contains one central computer and one or more checkout terminals. The purpose of a POS system is to exchange inventory for monetary value – in the form of either cash, cheques or credit cards. POS systems can be used to analyse sales data, track inventory and provide both physical and electronic payments options.

POS systems have become more sophisticated in recent years. Today, they must provide high usability, the ability to enter product information (for example, with a barcode scanner), provisions to provide pricing and security features. From a retailer's perspective, POS systems provide high business value. Their reporting features allow retailers to analyse sales data and determine which products or services are or are not selling and at which time they are most popular. Reports can be generated for any time period.

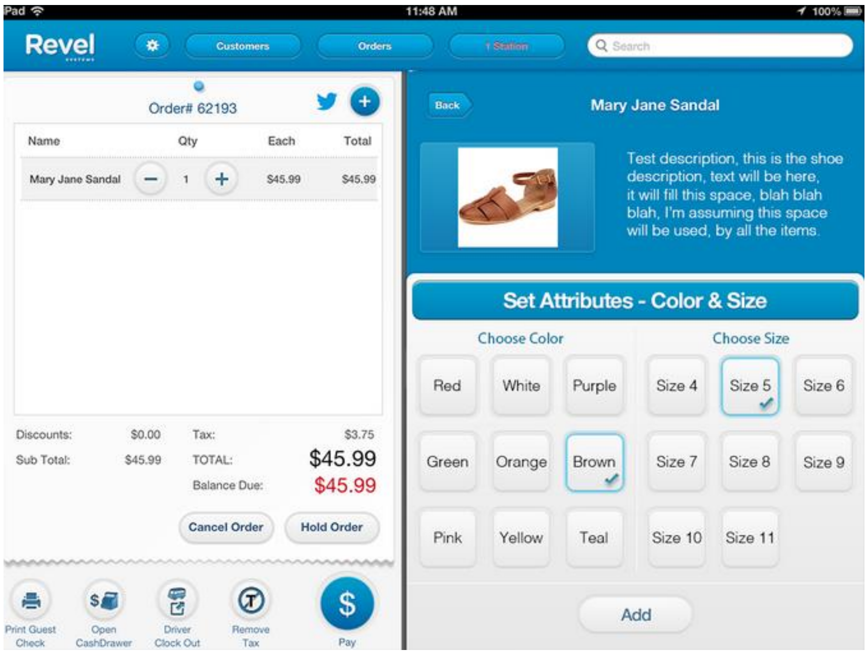
There are many providers of POS systems with each implementation varying in style. There are industry specific systems available. For example, game rental stores and beauty salons have varying business requirements and are suitable candidates for industry specific POS systems. (Entrepeneur.com, 2014)

4.2.2 Cloud Based POS Systems

Cloud based point-of-sale systems are a relatively recent advancement in this area, coming to the fore along with cloud technologies in the past number of years. These POS systems can be run from tablet devices, particularly those running the iOS and Android operating systems. Cloud systems provide merchants anytime anywhere access to data – including the systems reporting features as well as inventory data. (Belicove, 2013)

POS systems that are based on cloud software incur lower costs. There are no minimal, upfront or maintenance fees. Costs are typically charged on a monthly basis. Cloud based systems are straightforward to patch and upgrade by means of software updates that are delivered over the internet. Drawbacks associated with traditional systems such as hardware upgrades are eliminated. Data risks from corruption and malicious attacks are minimized using secure cloud servers which support replication and authentication.

The cost of downtime is a factor of a cloud based POS systems. Should an internet connection fail or should the cloud server experience downtime the vendor may be unable to continue processing transactions. (Wicks, 2014) One provider that offers offline access and synchronisation features are Revel Systems whose Revel POS software has been used by both retail and grocery stores as well as by those hosting large events. A view of the Revel POS interface, running on an iPad, is shown in *Figure 28*.



Order# 62193

Name	Qty	Each	Total
Mary Jane Sandal	1	$45.99	$45.99

Discounts: $0.00 Tax: $3.75
Sub Total: $45.99 TOTAL: $45.99
Balance Due: $45.99

Test description, this is the shoe description, text will be here, it will fill this space, blah blah blah, I'm assuming this space will be used, by all the items.

Figure 28 Revel Systems cloud-based POS software

4.3 Payment Verification Methods

Payment verification is amongst the most important factors when transferring funds from a payee to a recipient. Numerous security measures have been put in place to ensure positive verification and to prevent fraud. According to statistics compiled in 2014, the worldwide credit card fraud market was worth $5.5 billion. In America alone, 10% of the population have reported being affected by credit card fraud. (StatisticBrain, 2014) Verification measures used today include PIN numbers, bank supported measures such as *Verified by Visa*, username and password combinations, and biometric identifiers.

4.3.1 PIN Verification

PIN numbers are utilised to verify credit or debit card transactions at point-of-sale. The payee enters their PIN to verify the payment. The four digit numbers are assigned by the issuing bank and communicated to the user to ensure that predictable combinations such as "1234" are not used. The use of four digits allows a total of ten thousand possible PIN combinations. Some countries, such as Switzerland, use six digit numbers for additional security. To prevent the random guessing of PINs, issuers will "lock" the card and disallow further transactions following a set number of incorrect attempts. MasterCard disable the user of credit and debit cards after three incorrect PIN attempts. (MasterCard, 2015)

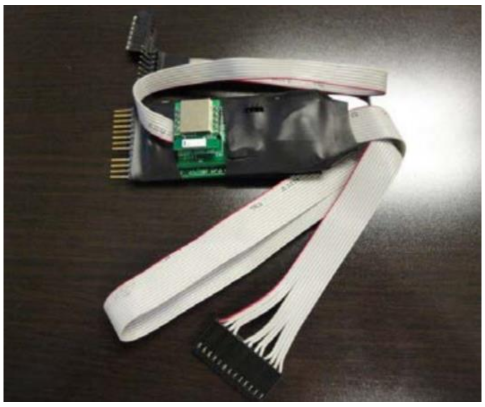


Figure 29 A Bluetooth enabled skimming device

The predecessor to PIN verification was signature verification – the user was required to sign a receipt as a means of proving their identity. The PIN method is widely considered to be much more secure. However, the method has numerous vulnerabilities. POS devices can be compromised with the installation of malicious software that can record and forward card information to hackers. ATM skimmers, like that shown in *Figure 29*, are fraud devices that criminals attach to ATMs which record both credit card numbers and their associated PINs. (Krebson Security, 2015)

4.3.2 Verified by Visa

Verified by Visa is an online security measure available for all cards supplied by Visa. It was introduced as a means of preventing unauthorised use of payment cards. When a user first attempts to make an online payment using a Visa supplied card they will be asked to choose a password to verify their identity. Then, this password will be required to confirm all future online transactions. *Verified by Visa* uses a secure connection when requesting end user verification – all data transmitted is encrypted. (Visa, 2015)

4.3.3 Username and Password Verification

Username and password verification is offered as an alternate means to providing payment information when using online payment systems. The platforms which will be discussed in Section 4.4 all offer username and password verification. It is a means of verification and fraud prevention whereby the user enters their payment information once, and it is then stored by the payments provider on a secure server. The stored information is encrypted. Username and password verification is an effective means of protecting credit card information. The user, however, may still be compromised if their username and password combination is exploited. PayPal, for instance, suffered from a vulnerability that would have allowed hackers to access and then change the passwords of accounts by exploiting the use of authentication tokens. (Halleck, 2014)

Many providers of username and password verification, PayPal included, have begun utilising two factor authentication to boost security. In these systems, the user must first enter their username and password. Then, a unique single use security code is delivered to a trusted device, such as their smartphone. The user is verified upon acceptance of the code. Thus, even if their username and password combination were to be comprised, the user's data and payment details would remain inaccessible.

4.3.4 Biometric Verification

Biometric verification is a means of identifying individuals by analysing their unique biometric identifiers and comparing them against a pre-existing database to determine a match. Verification methods include fingerprint analysis, facial recognition, retinal scanning and DNA analysis. The application of biometrics in payments verification will be discussed in Section 4.5.

4.4 Payment Platforms

Payment platforms offer Payments-as-a-Service (PaaS). They are a Software-as-a-Service (SaaS) methodology for performing credit and debit card payments on the internet. Complexity is hidden from the end user or merchant by the PaaS provider. The end user initiates a transaction with the platform through a single gateway and the platform handles the difficult back-end processing to complete the transaction.

4.4.1 PayPal

PayPal was first founded in 1998 by Max Levchin and Peter Thiel and began operating as a payments service in 1999. The company was originally known as X.com prior to changing its name to PayPal in June 2001. PayPal allows both individuals and businesses to make and accept payments in a choice of 100 different currencies. They launched their initial APIs in May of 2004, allowing developers access to its services for the first time. (PayPal, 2014)

PayPal provides each of its users with a digital wallet which they can fund from a source such as a bank account, credit card or debit card. When executing transactions with PayPal the end user is required to enter their PayPal username and password, at a minimum, for verification purposes. The user does not have to provide sensitive account information to the merchant and all payment processing is handled by PayPal. (Realex Payments, 2013)

Developers have a number of options to integrate with PayPal. Buttons and banners can be placed on websites to provide access to PayPal. PayPal provides a mobile SDK that is 100% platform native and allows developers to support payments within their applications. A third integration option is provided through a REST API. Developers can invoke PayPal services, accept payments from credit cards and implement a secure login system amongst other features by utilising this API. (PayPal, 2014) All PayPal services are PCI complaint. PayPal fees vary per country. In the United States, PayPal charges merchants up to 2.9% of the value of each transaction, plus an additional $0.30 per transaction.

4.4.2 Braintree

Braintree was founded in Chicago in 2007 by Bryan Johnson. They are a payments processing company that allow businesses to processing digital payments through their gateway. Since 2013, Braintree have been a subsidiary of PayPal, though the company continues to operate independently. Braintree are a PCI compliant company whose services are available in 40 countries and 130 currencies.

To process payments with Braintree a merchant account is required. Developers can then integrate the available services as required into websites and mobile applications. Payments are made using a “payment method nonce” identifier. This is a string value generated by Braintree that is used to represent a customer’s payments method. It can be employed by developers to store a customer’s details and to create a transaction. (Braintree, 2014)

Services available from Braintree include transaction management, customer management and recurring billing. Customer management services allow a merchant to store a client’s payments details for use at a later date. This features means users do not have to enter their payment details each time they interact with a merchant that is using the Braintree platform – the stored payment information associated with their account can be used. Recurring billing allows merchants to setup and provide subscription services through Braintree.

Following their takeover by PayPal, Braintree merchants can now accept payments from PayPal users. Merchants may also store PayPal users in their Braintree customer vault. All of this functionality is achieved using the “payment method nonce” approach mentioned above.

Braintree have partnered with Kount to provide fraud detection capabilities to all merchants who are using the platform. Fraud detection is an optional feature that checks a transaction for malicious activity before it is processed.

4.4.3 Stripe

Stripe is a company that allows businesses to accept payments online. Stripe was founded in 2010 by brothers John and Patrick Collison and the company launched its payments services in September 2011. Stripes services are being used by online companies including Shopify, Reddit and Lyft. (Sims, 2014)

Stripe are a PCI compliant payments gateway. To utilise Stripe its API must be implemented within an application by a developer. Stripe provides a method called tokenisation for creating transactions. When a user provides their payment information it is encrypted and sent directly to Stripe who create a single use token associated with that payment. This token can then be charged via to a second call to Stripe to complete the payment. Sending the payment details directly to Stripe for tokenization prevents the raw payment information from being stored on the merchant's server – only the token can be stored by the merchant. This process allows Stripe to enforce PCI compliance as no sensitive data will pass through the merchant's servers. (Keen, 2014)

Stripe's API is available across a number of different programming languages – including those used for mobile application development. A recently released feature called Stripe Checkout provides readymade payment forms that reduces the development code required. Stripe's complete payments stack includes services that allow the storing of payment details as well as the creation and processing of subscription plans. Merchants using stripe can accept payments in up to 138 currencies. A merchant can accept payments from global customers and they will automatically receive the payment in their own local currency. Fraud protection is offered by Stripe. (Stripe, 2014) The company has been valued at up to €1.3 billion. (Weckler, 2014)

4.5 Biometric Payment Platforms

Biometric payment platforms provide an alternative method of user verification in payment systems. Biometric identification measures one or more of the unique identifying physical characteristics of the user to verify their identity. These characteristics include, and are not limited to, fingerprints, facial features and iris or retinal scanning. Biometrics have been implemented, very recently, in payments systems such as Apple's Apple Pay technology and MasterCard's biometric enabled credit card.

4.5.1 Apple Pay

Apple announced their Apple Pay platform in September 2014 and it became available for use on October 20th of the same year. The platform allows users to complete everyday purchases using an Apple mobile device. Payment information is encrypted and then stored using a dedicated chip on the mobile device called *Secure Element*.

Apple CEO Tim Cook emphasised his companies desire to move away from traditional payment cards. He said "we're totally reliant on the exposed numbers, and the outdated and vulnerable magnetic interface — which by the way is five decades old — and the security codes which all of us know aren't so secure." (Jeffries, 2014)

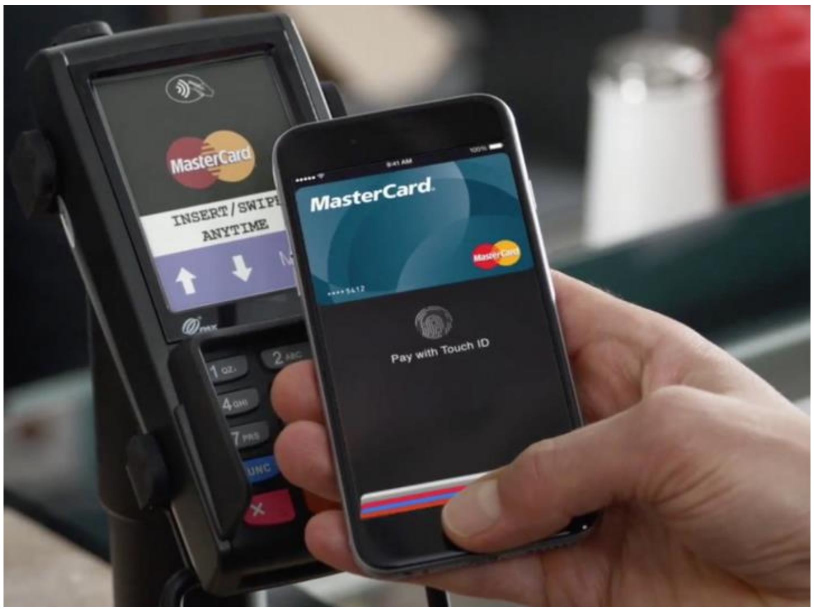


Figure 30 Biometric verification with Apple Pay

Apple Pay allows users to add their payment card information to its system. Users can then complete checkouts using Apple Pay at both physical point-of-sale locations and in online stores. The checkout is authenticated using Apple's Touch ID technology. Once Apple Pay has been chosen as the payment method the user must place their finger on the fingerprint sensor embedded in the Apple device to authorise the transaction, as shown in *Figure 30*. Here Apple have replaced traditional verification methods, including signatures and PIN verification, with a biometric solution.

An Apple Pay API has been made available to iOS developers, allowing them to integrate the payment platform into their mobile applications. Apple has partnered with retail outlets such as Subway and payment card providers such as MasterCard to increase the platforms availability.

From a security perspective Apple does not store the user's credit card details on their own servers – they are stored using the *Secure Element* chip. The encrypted data stored on this chip is not of the actual payment details, but is instead a unique user token that identifies only that user. As Apple doesn't store the user's payment information, it can never be shared with a merchant – a huge advantage in the event of a security breach. However, the TouchID technology introduced by Apple has been breached – experiments were conducted showing that the sensor could be tricked into a false authorisation. (Thompson, 2014)

Apple's wallet technology itself has been touted as secure. However, weaknesses exist in the verification methods used by some banks that allow criminals to add payment information for stolen credit card details to an Apple Pay account. According to Silicon Republic, criminals are achieving success with this methodology because "some banks are asking for the last four digits of a social security number, information that is easy enough for identity thieves to get their hands on." (Kennedy, 2015)

4.5.2 MasterCard

MasterCard launched its own biometric payment methodology in October 2014 following trials in Norway. They have developed a system that utilises a biometric enabled credit card which allows users to authorise payments at point-of-sale by using their fingerprint. MasterCard's solution was the first of its kind to be brought to the market.

The verification methodology is integrated into the card – a user must place their finger on the scanner contained within the card. Payment is contactless and initiated once the user holds the card within a close distance of the card reader.

The users fingerprint must be registered with their card prior to use. The fingerprint data is stored directly on the card itself, and not in a centralised database, as a security feature. Only the named cardholder can authorise a payment using this technology. The biometric cards relinquish the need for PIN numbers, often seen as point of weakness in credit card payments – many users continue to risk security by writing the numbers down.

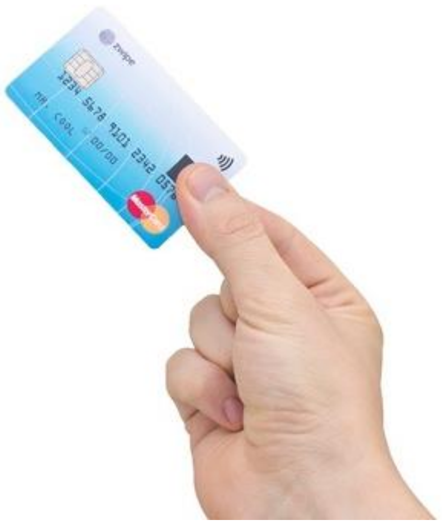


Figure 31 MasterCard's biometric enabled credit card

MasterCard’s current president of security, Ajay Bhalla, stated “our belief is that we should be able to identify ourselves without having to use passwords or PINs. Biometric authentication can help us achieve this – our challenge is to ensure the technology offers robust security, simplicity of use and convenience for the customer.” (Brignall, 2014)

MasterCard plan to introduce their biometric cards into numerous markets in the future – they plan on providing general availability within the United Kingdom by the autumn of 2015.

Chapter 5: Methodology

5.1 Project Scope

This project will determine if facial detection and recognition technologies could be applied as an alternate biometric verification method for payments systems. As payments processing continues to move away from traditional systems and towards cloud based systems it is important for payments verification to evolve with it. One such possible verification technique is the application of facial authentication of the cardholder. This will be investigated through the implementation of this project by determining the success rates of a facial recognition algorithm.

To achieve facial verification a photometric recognition algorithm will be implemented using GNU Octave. The photometric algorithm which will be applied is eigenfaces – as described in Section 2.4.2. This algorithm was chosen as it calculates a small set of eigenfaces which can subsequently be applied to determine the uniqueness of a face when compared to a large dataset. Utilising a small set of eigenfaces which represent a much larger image set is desirable for processing efficiency.

The front end of the application will be implemented through C#. Communication with the facial recognition algorithm in Octave is to be achieved using Redis. To assess the viability of the eigenfaces algorithm an image will be captured and processed using imaging hardware. Then it will be analysed for faces with an object detection algorithm. Object detection is not the goal of this project – a previously implemented algorithm will be utilised for this purpose. Microsoft's Kinect sensor is the hardware that will be employed to capture source imagery – its obtainable data streams provide the greatest flexibility for this project and will allow for image processing techniques such as background removal to be applied.

Detected facial images will be processed using the facial recognition algorithm. It shall compare the unknown facial image to its collection of known facial images to determine if a match exists. Should a match exist the previously unknown facial image will be considered identified. The eigenfaces algorithm must take into account possible variations in facial images such as lighting and pose along with features such as glasses.

The facial recognition algorithm will initially be trained using a previously compiled database that contains a wide variety of facial images for a sizeable set of individuals. This training database will be used as the initial seed data when implementing the recogniser.

A database will be utilised to store facial data representing other individuals. This facial data will be captured during the implementation of this project and applied to test the viability of the algorithm. The database provider that will be utilised is CouchDB.

The completed architecture will be tested using a number of individuals to determine the success rate and reliability of the facial recognition algorithm. A number of areas will be tested, including recognition under varied lighting conditions and recognition using a limited set of source images for an individual. Recommendations will be made with regard to the primary research question, based upon the data collected during the testing process.

5.2 High Level Features

The following presents a high level overview of the features that are proposed to be implemented for this project. The features are classified using the MoSCoW method. MoSCoW was first developed by Dai Clegg in 1994 while working at Oracle UK. This method breaks features into four separate priorities: must have, should have, could have and won't have. The MoSCoW approach is used in software development to ensure all stakeholders understand the most important requirements of the project and to separate what has to be delivered from what may not be delivered dependent upon the availability of resources. Must requirements represent the critical business needs and the success of a project depends upon their delivery. Should requirements are delivered if possible but project success does not rely upon them. Could requirements are those that would be nice to have but their implementation should not affect anything else in the project. Similarly, won't requirements are those which would be nice to have but definitely will not be implemented during the current iteration. (Haughey, 2014) The feature prioritisations utilised by MoSCoW are shown in *Table 3*.

MoSCoW Feature Prioritisation

Priority Name	Priority Number
Must Have	1
Should Have	2
Could Have	3
Won't Have	4

Table 3 Moscow feature prioritisation

The number in brackets following each feature represents its priority.

- Implement a photometric facial recognition algorithm, using a training set of facial images as the initial seed data (1),
- Implement a database to store facial data for captured individuals (1),
- Retrieve an image using imaging hardware and apply image processing techniques to ensure only desired facial images are captured (1),
- Execute facial detection on an image and extract any faces found (1),
- Execute facial recognition on captured facial images to determine if a facial match exists (1),
- Implement a payment processing module to complete transactions (4).

5.3 Research Question

Could facial recognition software be a viable identification and security measure at point-of-sale (POS) in payments processing systems?

5.4 Risk Analysis

The risk analysis for this project is shown in *Table 4.* These are the risks which are considered to have the most impact during the project implementation and upon achievable recognition results.

Risk Analysis

Risk	Risk Management Action
The facial recognition algorithms applied are shown to not have a 100% success rate. They are roughly 70% successful.	Possibly apply multiple algorithms to find a best match. Also, storing multiple images of the subject with a degree of variance will help increase recognition accuracy.
The proposed project is either hero or zero – it may be too dependent on existing APIs and SDKs.	Reduce reliance on APIs and SDKs by implementing algorithms from scratch. This includes facial algorithms, and also algorithms that will be required in image processing.
It has been proposed that the facial recognition algorithms will be implemented using GNU Octave (which is based on MATLAB). This is a high level language designed for numerical computations and presents a completely new environment.	This risk can be mitigated by referring to the GNU Octave documentation along with the relatively large volume of general information available on the internet.
Determining the correct facial image to verify could pose an issue. For instance, imaging hardware placed at point of sale may capture the face of both the intended person along with the faces of anyone present in the background.	The use of depth sensing may be important here. By knowing the expected distance of the user to be captured from the imaging hardware, depth sensing technology can be applied to ignore any background information present.
Image quality could vary in different situations. For example, lighting conditions may change. Also, the further a user is located from the imaging hardware then	Lighting conditions need to be constant. Full frontal lighting produces the most reliable results when applying photometric facial recognition algorithms. Image quality

the higher the level of image degradation will be.	is dependent on the imaging hardware. However, better quality should be expected if the hardware can be placed close to the point of sale location.
Image processing techniques are relatively hardware intensive and could impact the performance of this application, especially at point of sale where waiting times must, preferably, be as short as possible.	This project will be developed using capable hardware. Multithreaded operations will be considered if required to improve performance.

Table 4 Risk Analysis

Chapter 6: Project Specification

6.1 Use Case Diagram

Figure 32 below shows the use case diagram for the proposed facial recognition system.

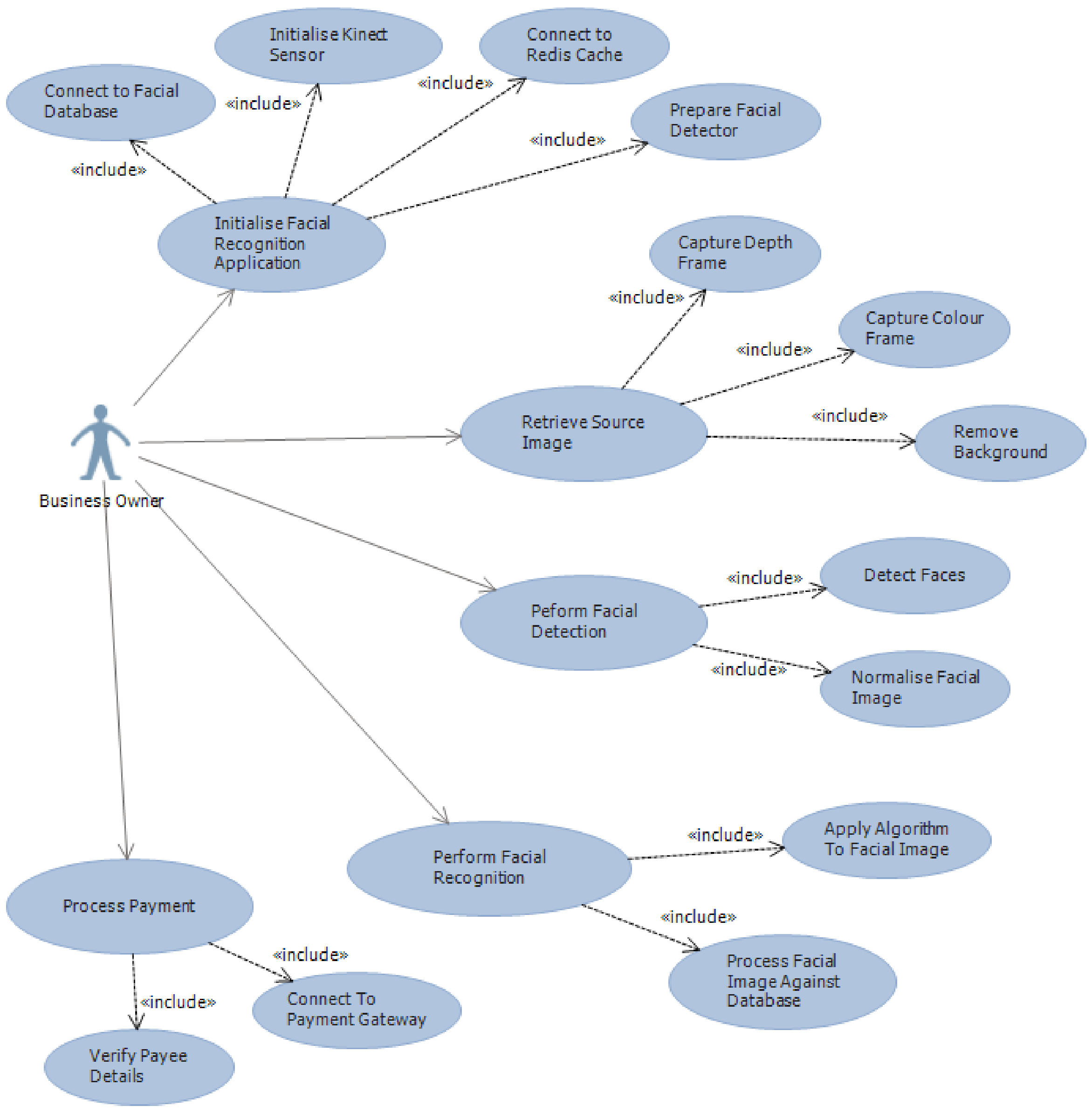


Figure 32 Facial Recognition Use Case Diagram

6.2 Use Case Analysis

6.2.1 Initialise Facial Recognition Application

Use Case Name	Initialise Facial Recognition Application
Use Case Actor	Business Owner
Description	The application will be launched by the use case actor and display the user interface for the system
Goal	The actors goal is to successfully launch the facial recognition application and be presented with the user interface
Success Measurement	The facial recognition application launches without error and the actor is presented with the user interface
Flow of Events	The actor will double click the applications icon which will begin the initialisation process that culminates in the user interface being displayed
Assumptions	The actor is familiar with how to use the application and it is executing on a system that meets the minimum computational requirements and contains all required dependencies

Table 5 Initialise Application Use Case

6.2.2 Retrieve Source Image

Use Case Name	Retrieve Source Image
Use Case Actor	Business Owner
Description	A full colour image frame and a depth image frame containing the subject will be captured simultaneously from the cameras data streams. The background (specified by a certain distance from the camera sensors) will then be removed
Goal	The actors goal is to acquire an image containing the visual data within a specified maximum distance from the camera
Success Measurement	Processed colour image is displayed on the user interface
Flow of Events	The actor will initialise the image capture sequence using the controls on the user interface. Two frames are captured from the sensor almost simultaneously – a full colour frame and a frame containing depth data. Using an image processing algorithm the background data will be removed from the colour image using distance measurements from the depth image. The data to be removed is specified using a predetermined value that denotes the maximum distance from the sensor for which data is to be kept
Assumptions	The cameras sensors have been initialised and are available to provide data to the application. The target is within the specified distance from the sensors

Table 6 Retrieve Source Image Use Case

6.2.3 Perform Facial Detection and Normalisation

Use Case Name	Perform Facial Detection and Normalisation
Use Case Actor	Business Owner
Description	The source image acquired from the camera is passed to a facial detection algorithm that finds any face(s) within that image. Following detection the facial image is normalised
Goal	The actors goal is to acquire a normalised facial image from the source image which was obtained during the *Retrieve Source Image* use case
Success Measurement	A normalised facial image is displayed on the user interface
Flow of Events	The actor will initialise the facial detection sequence once a source image has been acquired by the application. A facial detection algorithm will find the facial image, if any, within the source image. The detected facial image will then be normalised (in terms of image format and size) and displayed on the user interface
Assumptions	A source image has been captured with the target face visible. Any face detected will be assumed to be the desired facial image for comparison

Table 7 Facial Detection and Normalisation Use Case

6.2.4 Perform Facial Recognition

Use Case Name	Perform Facial Recognition
Use Case Actor	Business Owner
Description	The normalised facial image acquired during the *Perform Facial Detection* use case is analysed using a facial recognition algorithm. The data from this analysis is then compared to that in a facial database to determine if a match exists
Goal	The actors goal is to determine whether the detected facial image has a match in the database
Success Measurement	The results of facial recognition are displayed on the user interface and indicate if a match has or has not been located
Flow of Events	The actor will initialise the facial recognition sequence following successful facial detection during the *Perform Facial Detection* use case. A facial recognition algorithm will be executed on the image data. The result of this process will be compared against the images in the facial database to determine if a match exists
Assumptions	A facial image has been successfully detected prior to this use case beginning and that the face is not obscured in any way

Table 8 Facial Recognition Use Case

6.2.5 Process Payment

Use Case Name	Process Payment
Use Case Actor	Business Owner
Description	Payment processing will the executed following the successful identification of a facial image using the facial recognition system
Goal	The actors goal is to complete an electronic payment using the specified payment platform
Success Measurement	The users payment details are verified and a payment is processed without error
Flow of Events	The actor will initialise the payment processing sequence following successful facial recognition. The payee's payment details will be passed to the payment platform for verification. Payment processing will be completed by the payment platform following successful verification
Assumptions	Full and correct payment details have been provided to the system

Table 9 Process Payment Use Case

6.3 Prototype and Externally Visible Behaviour

A prototype has been developed during the design phase of this project that interacts with a Microsoft Kinect sensor and displays both the colour and depth stream data to the end user. It has been implemented in C# and uses the Kinect SDK to retrieve data from the sensor. Camera controls have been implemented to allow the user to control the pitch of the camera along the Y-axis. The colour and depth data is captured from the camera when it triggers an event handler indicating that new data is available. This is dependent on the frame rate of the streams which operate at a maximum of 30 frames per second.

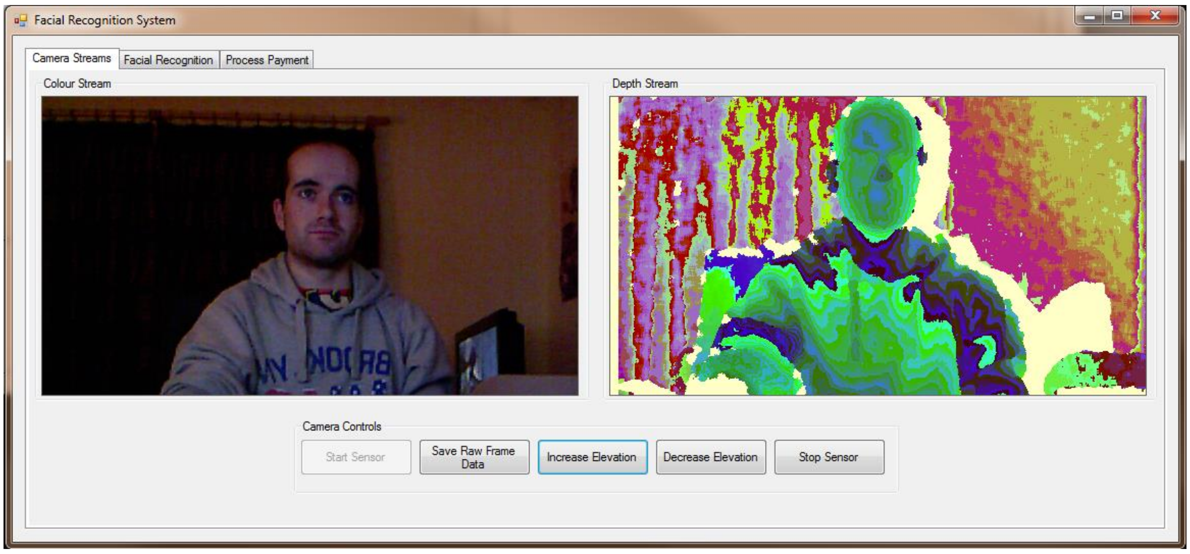
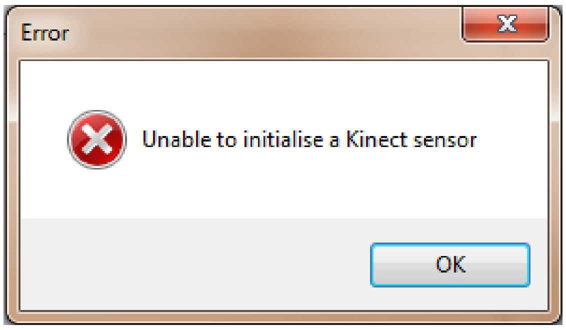


Figure 33 Facial Recognition System Prototype

Figure 33 above shows the tab that has been prototyped. This is the only tab for which prototype code has been implemented. The streams are displayed within their respective controls once a Kinect sensor has been detected by the application and has been started using the appropriate controls on the UI.


Error

Unable to initialise a Kinect sensor

Figure 34 Error message shown in the event that a Kinect sensor cannot be started

Disabled controls indicate that their functionality is not available in the running configuration. For example, if a Kinect sensor has already been started then the *Start Sensor* button will become unavailable. Error messages are shown as appropriate.

Externally visible behaviour will evolve throughout system development. It is anticipated that the end user shall be presented a visual representation of all the use cases described above. An option should exist to complete the facial detection, recognition and payment process automatically or, if required, to manually activate each step of the process.

6.4 Sprint Plan

The implementation phase of this project will use the agile method of software development where work is broken down into blocks referred to as sprints. Each sprint will last a period of two weeks and define a number of work items that must be completed. There are seven individual sprints planned.

Sprint	Work Items	Status
1	Implement Initial User Interface	*Complete*
	Interface with Microsoft Kinect Sensor	*Complete*
	Display Colour and Depth Data Streams	*Complete*
	Save Raw Colour and Depth Data to CSV Format	*Complete*
	Implement Depth Reduction Algorithm	*Complete*
2	Load Yale Training Database	*Complete*
	Reduce Training Set Using the Average Face	*Complete*
	Calculate Eigenfaces	*Complete*
	Calculate Weight Vectors	*Complete*
	Reconstruct a Known Face	*Complete*
	Recognise an Unknown Face	*Complete*
	Persist Data	*Complete*
3	Implement Facial Detection	*Complete*
4	Design and Implement Required Models	*Complete*
	Determine Generic Database Functionality	*Complete*
	Implement Concrete Database Functionality	*Complete*
	Design a User Interface for Database Interaction	*Complete*
5	Redis Protocol	*Complete*
	Implement Redis in Octave	*Complete*

	Implement Redis in C#	*Complete*
6	Normalise Image Data	*Complete*
	Implement Recogniser	*Complete*
7	Add Configuration Options to C# Application	*Complete*
	Improve Recogniser Efficiency	*Complete*
	Refactor C# Application	*Complete*
	Refactor Octave Application	*Complete*

Table 10 Sprint Plan

6.5 Test Plan

Sprint No.	Test No.	Description	Success Criteria
1	1	Test initialisation of a Kinect sensor	A single Kinect sensor should be detected as connected to the system and it should function without error when initialised
	2	Test retrieval of data from Kinect sensor	The Kinect data streams (colour and depth) should be initialised under different criteria and data should be retrieved as expected (correct format)
	3	Test save colour and depth data to CSV file	Two CSV files should be produced containing the raw colour and depth data
	4	Test background removal algorithm	Using supplied colour and depth data (of the same frame) the background of the colour image should be removed at all areas beyond a specified distance
2	5	Test load Yale training database	The Yale database should be successfully loaded from disk to memory
	6	Test calculate average face	Using the training database, the average face should be calculated
	7	Test calculate eigenfaces	Eigenfaces should be calculated using training data as the algorithms input
	8	Test facial recognition of an unknown face	An unknown facial image should be processed and its closest facial match determined
3	9	Test facial detection	Any facial images present within a source image should be found
4	10	Test connection to database	The application should successfully connect to and

			receive a response from the database
	11	Test data storage in the database	Data should be successfully stored and subsequently retrieved from the database
5	12	Test Redis integration	The applications developed should connect to and listen for messages being transferred using Redis
6	13	Test facial image normalisation	A given facial image should be normalised to the specifications required by the facial recognition system
	14	Test invoke recogniser	The facial recogniser should be invoked, and receive a response, from the main application
	15	Test payment system	A payment should be fully processed using the payments gateway

Table 11 Test Plan

6.6 Key Class Diagrams

Figure 35 below contains a high level overview of the key classes that will be required during the implementation of this project. These include a class to represent the actions of the Kinect sensor, an image processing class, an abstract definition for facial recognition algorithms and a definition of the initial functionality of the database. Also included are classes related to the sensor data – one for processing and one for performing input/output of raw data.

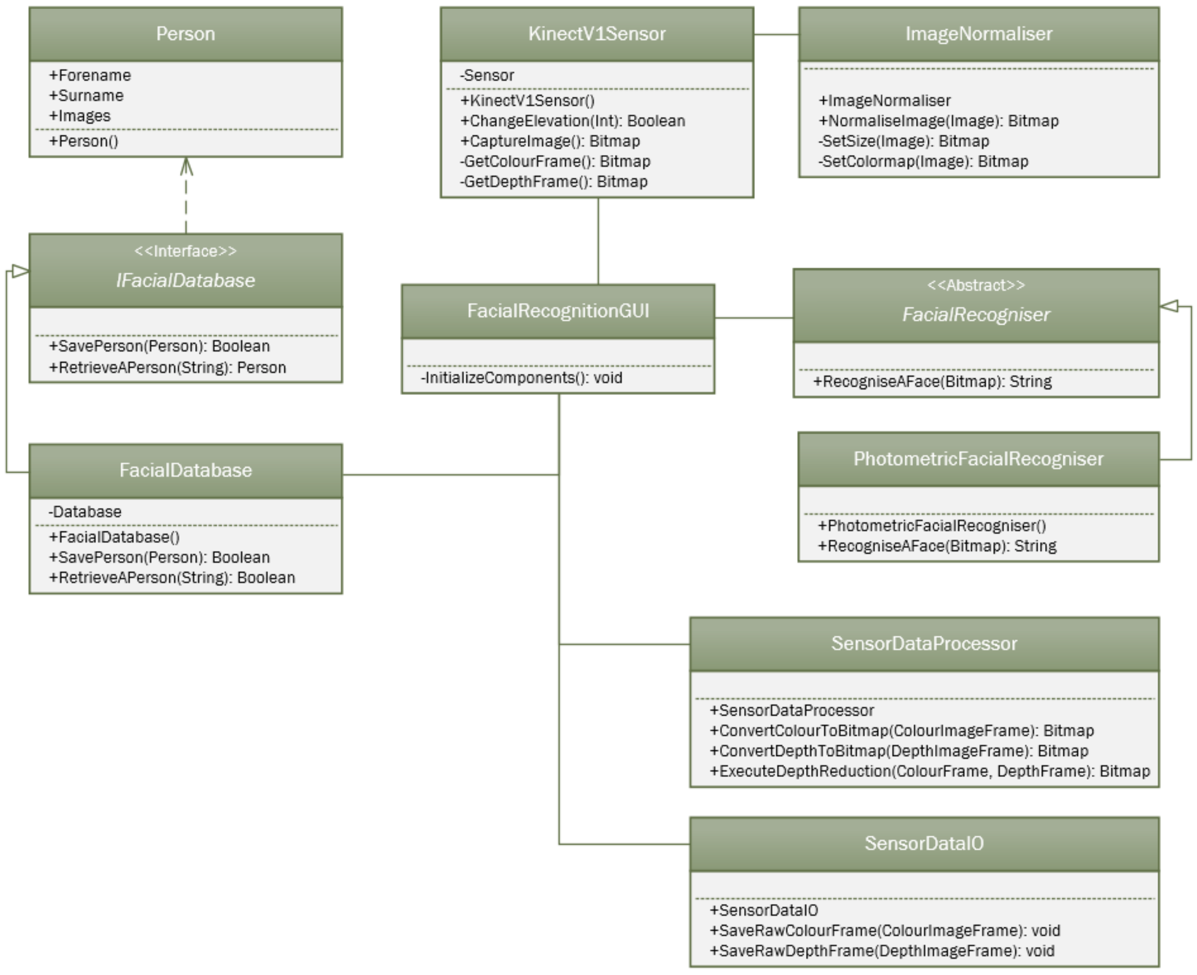

Person
+Forename
+Surname
+Images
+Person()
KinectV1Sensor
-Sensor
+KinectV1Sensor()
+ChangeElevation(Int): Boolean
+CaptureImage(): Bitmap
-GetColourFrame(): Bitmap
-GetDepthFrame(): Bitmap
ImageNormaliser
+ImageNormaliser
+NormaliseImage(Image): Bitmap
-SetSize(Image): Bitmap
-SetColormap(Image): Bitmap
<<Interface>>
IFacialDatabase
+SavePerson(Person): Boolean
+RetrieveAPerson(String): Person
FacialRecognitionGUI
-InitializeComponents(): void
<<Abstract>>
FacialRecogniser
+RecogniseAFace(Bitmap): String
PhotometricFacialRecogniser
+PhotometricFacialRecogniser()
+RecogniseAFace(Bitmap): String
FacialDatabase
-Database
+FacialDatabase()
+SavePerson(Person): Boolean
+RetrieveAPerson(String): Boolean
SensorDataProcessor
+SensorDataProcessor
+ConvertColourToBitmap(ColourImageFrame): Bitmap
+ConvertDepthToBitmap(DepthImageFrame): Bitmap
+ExecuteDepthReduction(ColourFrame, DepthFrame): Bitmap
SensorDataIO
+SensorDataIO
+SaveRawColourFrame(ColourImageFrame): void
+SaveRawDepthFrame(DepthImageFrame): void

Figure 35 Key Class Diagrams

6.7 Architecture Diagram

The architecture diagram shown in *Figure 36* below provides a general overview of the proposed system implementation. All required image data is captured using a Kinect sensor. Processing occurs within the main facial recognition application. This application can communicate with the facial database and payment gateway as required. It also communicates with the Octave Recogniser via the Redis cache.

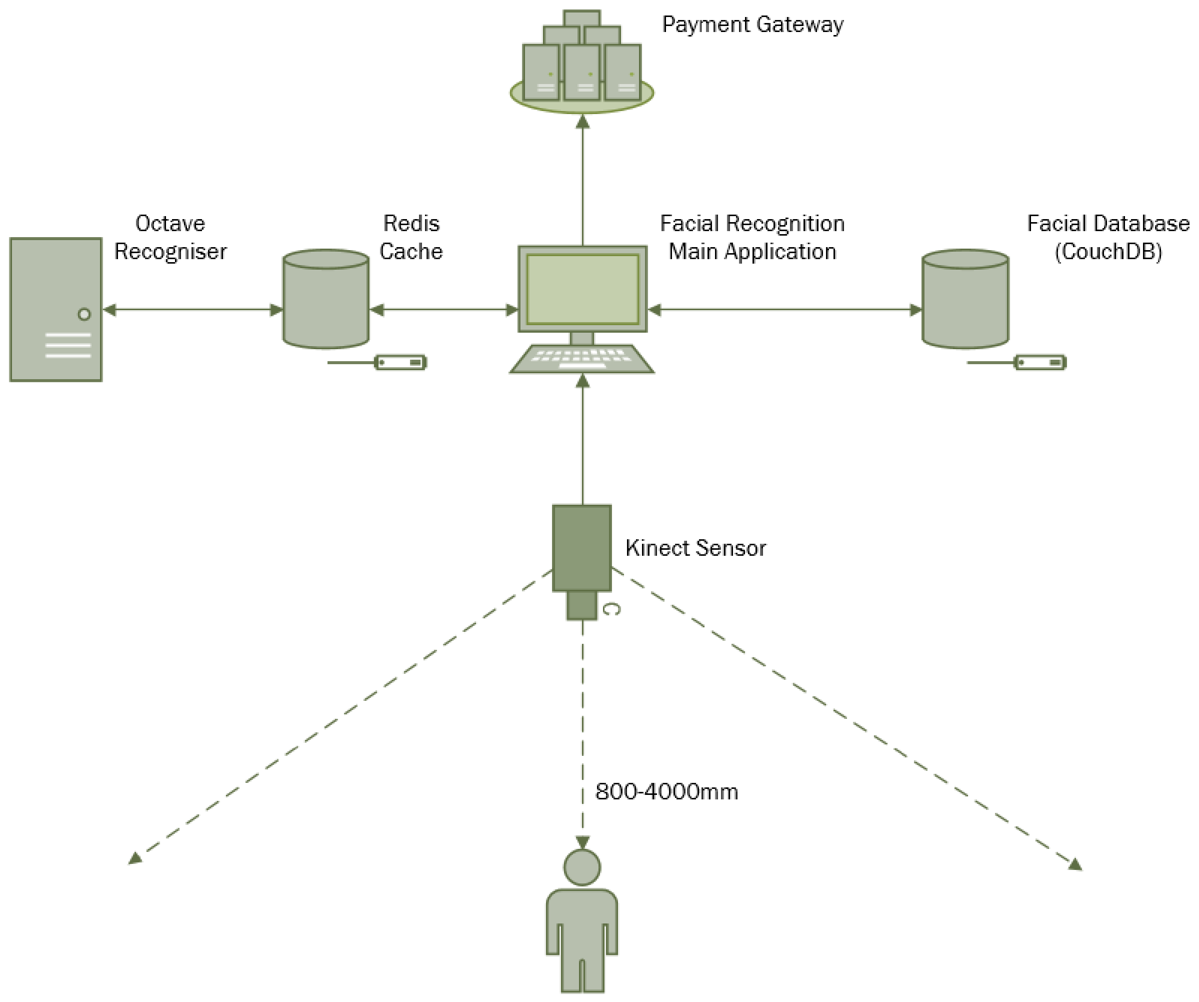


Figure 36 Architecture Diagram

6.8 Conceptual Diagrams

The follow diagrams provide a conceptual overview of the facial recognition system which will be implemented. *Figure 37* shows the process of training the training the facial recogniser and *Figure 38* shows the process of invoking the recogniser to identify an unknown facial image.

6.8.1 Training the Facial Recogniser

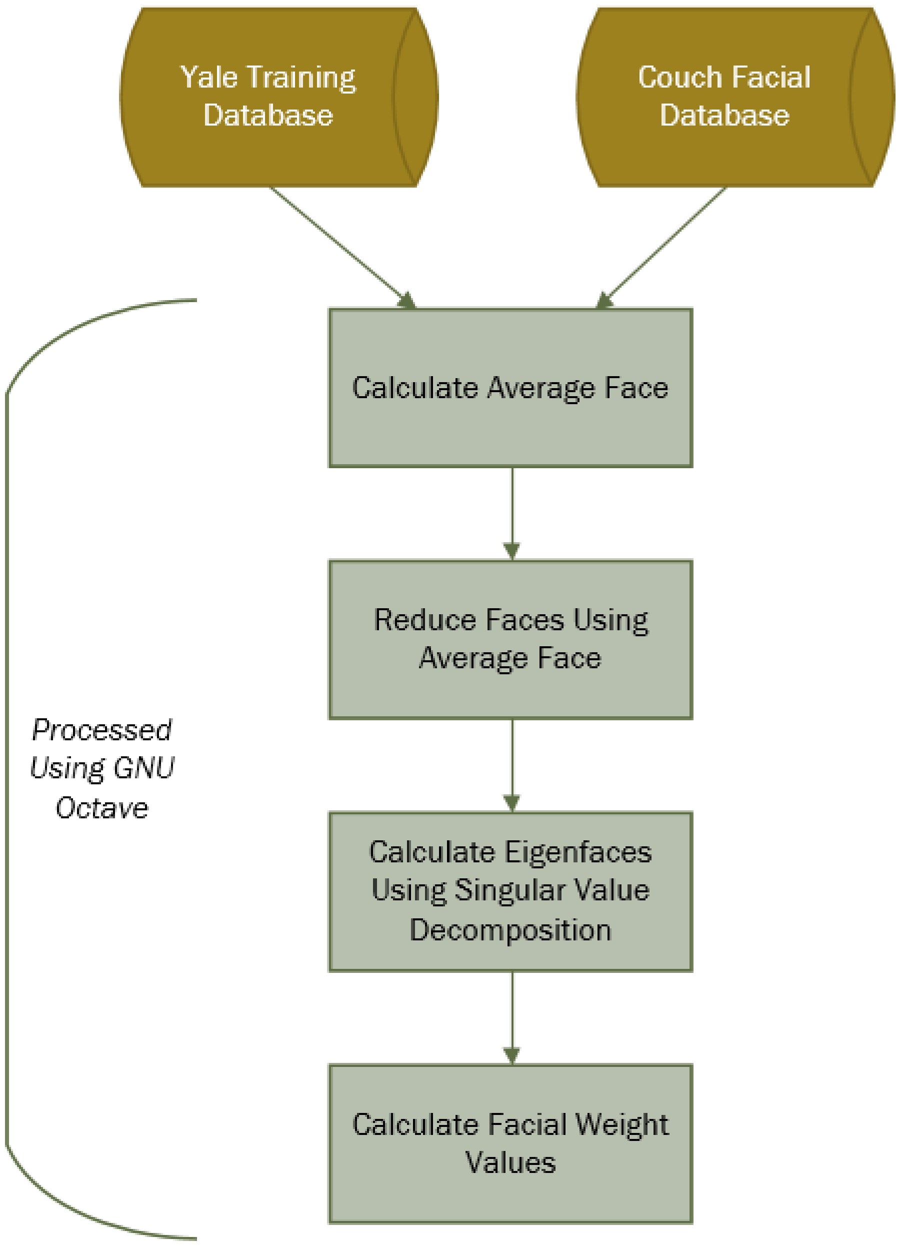
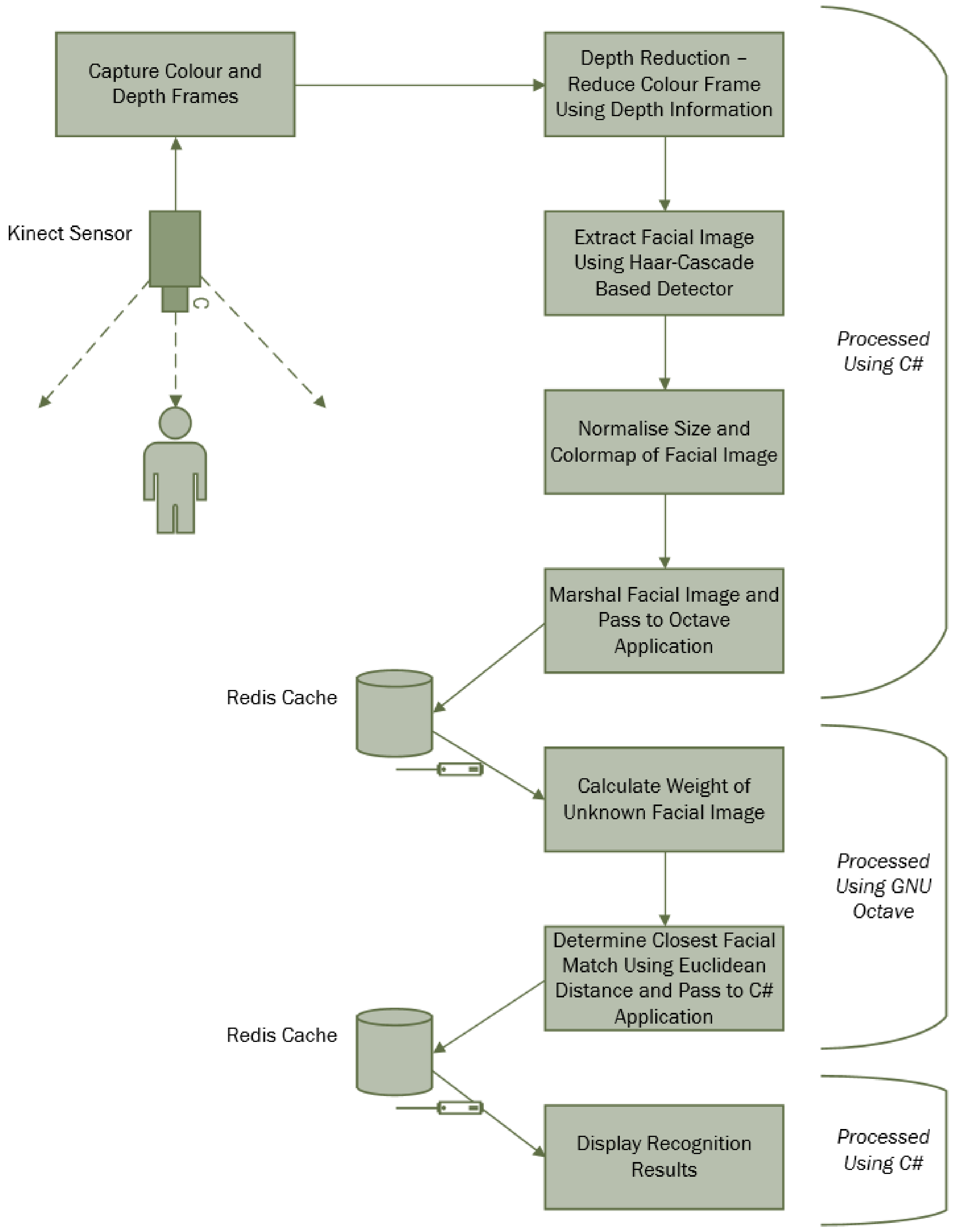

Yale Training Database
Couch Facial Database
Calculate Average Face
Reduce Faces Using Average Face
Calculate Eigenfaces Using Singular Value Decomposition
Calculate Facial Weight Values
Processed Using GNU Octave

Figure 37 Conceptual Overview - Training the Facial Recogniser

6.8.2 Recognising an Unknown Facial Image



Capture Colour and Depth Frames
Depth Reduction – Reduce Colour Frame Using Depth Information
Kinect Sensor
Extract Facial Image Using Haar-Cascade Based Detector
Normalise Size and Colormap of Facial Image
Marshal Facial Image and Pass to Octave Application
Processed Using C#
Redis Cache
Calculate Weight of Unknown Facial Image
Processed Using GNU Octave
Determine Closest Facial Match Using Euclidean Distance and Pass to C# Application
Redis Cache
Display Recognition Results
Processed Using C#

Figure 38 Conceptual Overview - Recognising an Unknown Facial Image

Chapter 7: Implementation

7.1 Implementation Methodology

This project was implemented using a variety of hardware and software technologies. The Git distributed version control system was utilised. A project repository was created on GitHub and is viewable at: https://github.com/dmk2014/FYP

7.1.1 Hardware

- Microsoft Kinect Sensor (Version 1),
- Primary Development Machine,
  - Intel Core i5-3570K CPU @ 3.8GHz,
  - 8GB RAM.

7.1.2 Software

- Microsoft Windows 7,
- Microsoft Visual Studio 2013,
- Microsoft Kinect SDK (Version 1.8),
- C# (for main application),
- GNU Octave (for facial recognition),
- Redis (for integration of C# and GNU Octave),
- CouchDB (for facial database),
- EmguCV (for facial detection),
- Git (for version control).

7.2 Sprint 1 – Kinect Functionality

The first sprint was built on what was implemented during the prototype phase of this project. The original code was largely refactored and some minor changes were made to the user interface. This sprint developed the core of the functionality that was required to interface with the Kinect sensor, including connecting to it, retrieving the required data streams and processing the information they provide.

7.2.1 PBI – Implement Initial User Interface

The first PBI to be completed during Sprint 1 was the implementation of the initial user interface. A four tab interface, as shown in *Figure 39*, was designed. It included a tab for camera controls and streams, a tab for facial recognition functions, a tab for database functions and a tab for application configuration. The *Camera Streams* tab was fully designed during this PBI and the others followed during later sprints. The interface designed allowed the end user to capture frames from a Kinect sensor, to control its elevation and to save raw colour and depth stream data to disk.

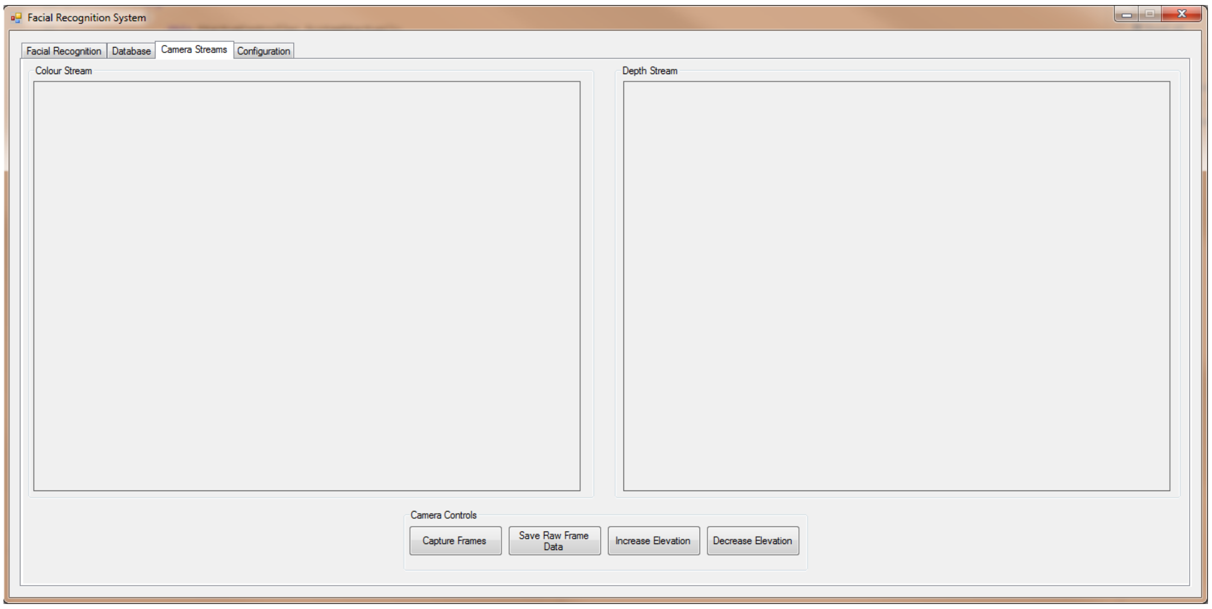


Figure 39 The Camera Streams tab which was implemented during Sprint 1

7.2.2 PBI – Interface with Microsoft Kinect Sensor

The next task during this sprint was to connect to a Microsoft Kinect sensor. To achieve this the Kinect SDK (available from Microsoft) had to be installed and its DLLs added as a reference within the C# project structure. This project utilised a Kinect version 1 sensor for which the latest compatible version of the SDK available at the time of writing was Kinect

SDK v1.8. The SDK provided an API to interface with the sensor and also handled the installation all required drivers.

The Kinect API provided a static array of all Kinect sensors that were attached to the system. In this project there was only ever be one sensor connected. Once the sensor had been physically connected to the system, a reference to it was acquired using the first item in the array, as shown in *Snippet 1*. An error message was generated in the case that there were none or more than one sensors connected.

```
Source File: KinectV1Sensor.cs

Sensor = KinectSensor.KinectSensors[0];

Sensor.Start();
```

Snippet 1 Connecting to and initialising a Kinect Sensor

7.2.3 PBI – Display Colour and Depth Data Streams

The Kinect sensor provided numerous data streams (described in Section 3.4.1), all of which were disabled by default. The streams which were required for this project, colour and depth, were enabled programmatically. The stream format had to be passed as a parameter when enabling it. For the colour stream the image type (e.g. RGB), resolution and frame rate had to be specified. In the case of the depth stream the resolution and frame rate had to be specified.

The Kinect sensors immediately began capturing data upon calling its *start* method.

A means for accessing the data had to be provided. This could have been achieved in two ways – either through an event handler or by polling the camera for new data after a specified time interval. In this case it was chosen to retrieve data using a polling model. Each stream was configured and then invoked programmatically each time new data was required. The maximum time to wait for data to be returned by the sensor had to be specified – during development a wait time of 700 milliseconds was found to be appropriate.

```
Source File: KinectV1Sensor.cs

this.Sensor.ColorStream.Enable(ColorImageFormat.RgbResolution640x480Fps30);
this.Sensor.DepthStream.Enable(DepthImageFormat.Resolution640x480Fps30);
this.Sensor.DepthStream.Range = Microsoft.Kinect.DepthRange.Default;
```

Snippet 2 Configuring sensor data streams and depth range

The data returned from both streams was in raw format. This data was converted to bitmap format and then displayed using the picture box components on the user interface. This process occurred each time new data was required from the sensor. For example, the colour data was transferred to a byte array using a method defined in the Kinect API. Then, this byte array was converted to a bitmap image. The conversion process is shown in *Snippet 3*.

```
Source File: SensorDataProcessor.cs

private Bitmap ConvertColorByteArrayToBitmap(byte[] colorData,
       int imageWidth,
       int imageHeight,
       int pixelDataLength,
       PixelFormat pixelFormat)
{
    var image = new Bitmap(imageWidth, imageHeight, pixelFormat);
    var imageRectangle = new Rectangle(0, 0, imageWidth, imageHeight);

    var bitmapData = image.LockBits(imageRectangle,
                                    ImageLockMode.WriteOnly,
                                    image.PixelFormat);

    var addressFirstPixel = bitmapData.Scan0;

    Marshal.Copy(colorData, 0, addressFirstPixel, pixelDataLength);
    image.UnlockBits(bitmapData);

    return image;
}
```

Snippet 3 Converting a Kinect Colour Frame to a Bitmap image

The entire process was tied together using the *Capture Frames* control on the user interface. This control called a method to capture both a colour frame and a depth frame and display them on the interface. A sample of the captured frames is shown in *Figure 40*.

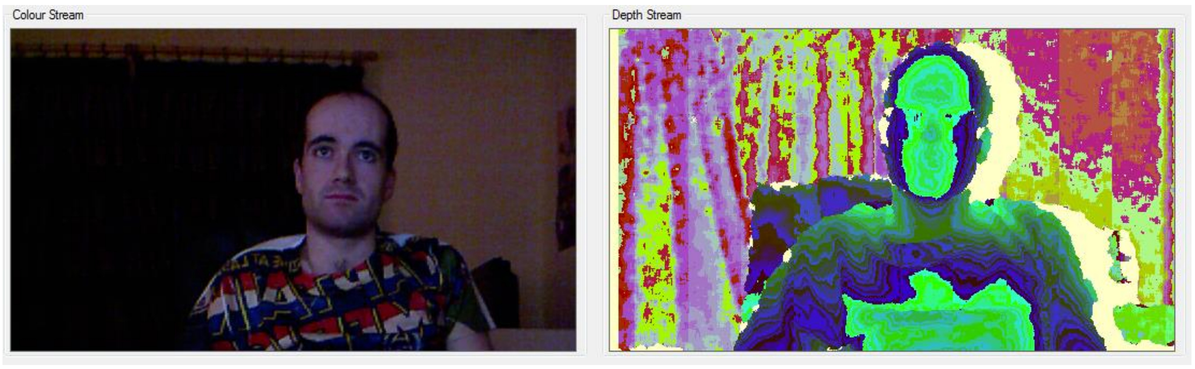
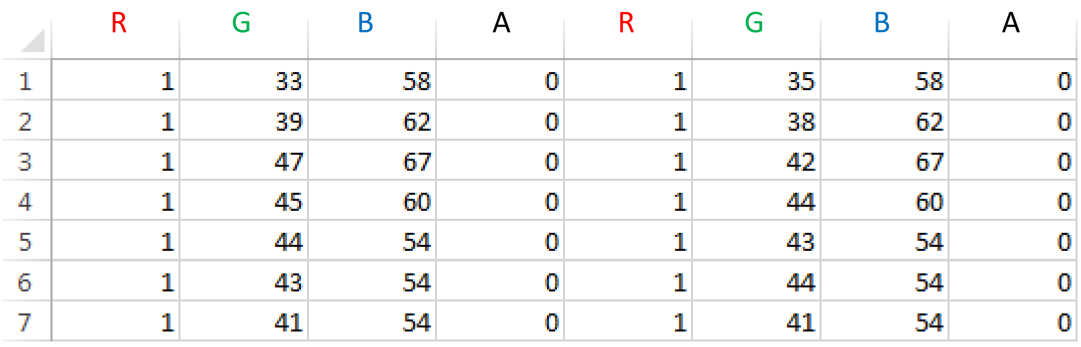


Figure 40 Screen capture of displayed colour and depth frames

7.2.4 PBI – Save Raw Colour and Depth Data to CSV Format

The raw data returned from the sensor was converted to bitmap images for display purposes. A background removal algorithm will be implemented later in this section and will process the raw data prior to conversion to a bitmap image. For this purpose an input/output class was developed that wrote the raw data for both a colour frame and a depth frame (ideally captured simultaneously – i.e. containing data for the same image) to a CSV file. This data will be used during the design of the aforementioned algorithm.



	R	G	B	A	R	G	B	A
1	1	33	58	0	1	35	58	0
2	1	39	62	0	1	38	62	0
3	1	47	67	0	1	42	67	0
4	1	45	60	0	1	44	60	0
5	1	44	54	0	1	43	54	0
6	1	43	54	0	1	44	54	0
7	1	41	54	0	1	41	54	0

Figure 41 Sample of colour frame RGB data, from a CSV file, showing 14 pixels

7.2.5 PBI – Implement Depth Reduction Algorithm

A depth reduction algorithm was implemented which allowed data outside a maximum distance threshold to be removed from a captured colour image. In this context, it allowed for background faces which were not desired to be removed. A colour image and a depth image were captured almost simultaneously from the Kinect sensor. The images were of the same resolution and represented the exact same data, though in a different format.

```
Source File: SensorDataProcessor.cs
for (var i = 0; i < depthData.Length; i++)
{
    if (depthData[i].Depth > maxDepth)
    {
        int startIndexRGBA = i * 4;
        int rIndex = startIndexRGBA;
        int gIndex = startIndexRGBA + 1;
        int bIndex = startIndexRGBA + 2;
        int aIndex = startIndexRGBA + 3;

        colorImageData[rIndex] = 0;
        colorImageData[gIndex] = 0;
        colorImageData[bIndex] = 0;
        colorImageData[aIndex] = 0;
    }
}
```

Snippet 4 The core of the depth reduction algorithm

The depth image pixel data was extracted from the raw depth frame and converted to a *DepthImagePixel* array using a method from the Kinect API. Each item in the array represented the distance of a pixel (starting from the top-left corner, or point 0, 0) from the camera in millimetres. The array was traversed using a loop. For any pixels in the depth array which were outside the specified maximum distance, the corresponding pixels in the colour image were removed. This algorithm is shown in *Snippet 4* above.

Processing the depth data presented a challenge. Each depth pixel was represented by one value but each colour pixel, as they were in RBG format, had four values. To correctly remove a colour pixel, the index of the current depth pixel was multiplied by four to acquire the start index of the related RGB pixel. Then, using this new index four contiguous RGB pixels were removed from the colour image. In this case, to "remove" an RBG pixel its colour value was changed to black. In RBG colour space, black is represented using zeroes.

7.3 Sprint 2 – Facial Recognition

Note: the terms eigenvectors and eigenfaces are used interchangeably, as discussed in Section 2.4.2.

The core functionality required for this project was facial recognition. To achieve this an algorithm was required which, when trained, determined whether an unknown facial image was that of a person known to the system.

As discussed during the Project Scope in Section 5.1, it was decided to implement the photometric eigenfaces algorithm, as described by Turk and Pentland. GNU Octave was chosen as the development environment. It is a high-level interpreted language whose syntax is very close to that of MATLAB. Octave was chosen for this project as it is designed for numerical computations. Data types such as matrices are preferred and highly suited to the specified area when working with images.

Using the eigenface algorithm, a set of known facial images can be classified using a much smaller set of eigenfaces. By classifying an unknown face using this set of eigenfaces, it can be determined if a facial match exists within the set of known faces.

7.3.1 PBI – Load Yale Training Database

Prior to implementing the facial recognition algorithm it was important to acquire a training set of data. This set was used as the initial seed data to train the algorithm. There are a number of facial training sets available. The Yale facial database was chosen as the initial training set for this project. More specifically, the cropped version of the *Extended Yale Face Database B* was used. This database contains images of 38 different subjects. For each subject, there are up to 65 images captured under variations in both pose and lighting. The images are normalised – they are all grayscale images using the .pgm image format and they have all been resized to 168x192 pixels. A sample image is shown in *Figure 42*.

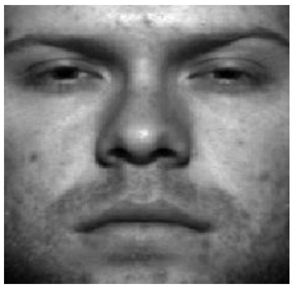


Figure 42 Image of subject yaleB01 from the cropped Extended Yale Face Database B

The cropped images were first reported on by Lee, Ho and Kriegman in 2005. (Lee, et al., 2005) The *Extended Yale Face Database B* was first used for research purposes by Georghiades, Belhumeur and Kriegman in 2001. (Georghiades, et al., 2001)

An Octave function was developed to load the Yale facial database from disk into memory, a section of which is shown in *Snippet 5*. Octave has native support for reading images stored in the .pgm format. Each image was loaded as a 168x192 matrix where each cell represented an individual pixel.

Each image was converted to a column vector of size 32256x1 and then appended to a matrix M. When the entire training set was loaded, M was of size 32256x2432. In essence, this means M contained the entire training set of 2,432 images. The matrix below provides a visual representation of this.

$$M = \begin{bmatrix} Image1Pixel1 & \cdots & ImageMPixel1 \\ \vdots & & \vdots \\ Image1PixelN & & ImageMPixelN \end{bmatrix}$$

Where M the number of images was in the training set, and N was the number of pixels in each image.

```
Source File: LoadYaleTrainingDatabaseFromImageFiles.m

img = double(imread(currentImagePath));
imgColVector = reshape(img, rows(img) * columns(img), 1);

data = [data, imgColVector];
```

Snippet 5 Reading an image and appending it to a matrix in Octave

Along with the pixel data for each image, a label was also stored. In this case the label corresponded to the subject's identifier as selected by the creators of the training set. For example, all images of the first subject were labelled as *yaleB01*. The image labels were stored in a corresponding vector of length M.

7.3.2 PBI – Reduce Training Set Using the Average Face

With the training set of images stored in the matrix M, the eigenface algorithm first specified that the images were further normalised by removing the average image. The average image was acquired by finding the mean of the data in each row of M. It is important to remember that each row of M represents a corresponding pixel for each image. This produced a column vector representing the average image (shown in *Figure 43*) which was then subtracted from each column in M to reduce the training set.

$$M_{normalised} = \begin{bmatrix} Image1Pixel1 & \cdots & ImageMPixel1 \\ \vdots & & \vdots \\ Image1PixelN & & ImageMPixelN \end{bmatrix} - \begin{bmatrix} AveragePixel1 \\ \vdots \\ AveragePixelN \end{bmatrix}$$

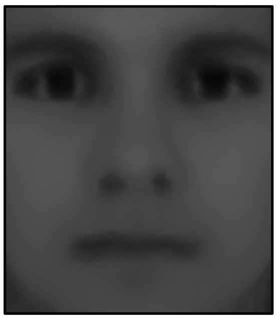


Figure 43 Representation of the calculated average face

7.3.3 PBI – Calculate Eigenfaces

Using the normalised data, the eigenfaces could be calculated. The eigenfaces are eigenvectors which represent the principal components of the data. These eigenfaces, which have a ghost-like appearance, are important to both the reconstruction and recognition of facial images.

7.3.3.1 – Eigenfaces with PCA

The eigenfaces were first calculated using Principal Component Analysis which was discussed in Sections 2.4.1 and 2.4.2. Firstly, the covariance matrix of M was calculated by multiplying M by its transpose. However, due to the size of M (recall, 32256x2432) this would result in a huge matrix of size 32256x32256. This was not computationally feasible on the primary development machine. Instead, the transpose of M was multiplied by M to produce a much more computationally efficient covariance matrix of size 2432x2432.

The **eig** function in Octave was used to calculate both the eigenvectors and eigenvalues of the covariance matrix. The resulting eigenvectors were reorganised to be in descending order of their eigenvalues. The process is shown in *Snippet 6*.

These eigenvectors are in lower dimensional space. To convert the eigenvectors back to higher dimensional space they were multiplied by the matrix of normalised facial images.

This method of calculating eigenfaces, however, proved ineffective. During manual testing of the algorithm – the eigenfaces calculated using this methodology failed in both the reconstruction of images from the original data set and in the recognition of an unknown facial image.

7.3.3.2 – Eigenfaces with SVD

A second method that can be applied to calculate eigenfaces is called Singular Value Decomposition (SVD). According to Philip Wagner, the SVD methodology is closely related to that of PCA. His research showed that SVD can be used as an alternative to the PCA method described above. (Wagner, 2011)

Octave's **svd** function wraps the behaviours described in PCA. Thus, it was not required to calculate the covariance matrix prior to the calculation of eigenvectors. After subjecting the matrix of normalised faces to SVD, an ordered matrix was returned which contained the eigenvectors of the data. The eigenfaces calculated by SVD proved effective in later tests. The SVD method is shown in *Snippet 6*.

```
Source Files: getEigenfacesPCA.m, getEigenfacesSVD.m
% PCA Method
C = Mnorm' * Mnorm;

[V D] = eig(C);

[D,i] = sort(diag(D), 'descend');

V = V(:, i);

V = V(:, 1:k);

% SVD Method
[V S U] = svd(M, "econ");

V = V(:, 1:k);
```

Snippet 6 Calculating eigenfaces (V) using PCA and SVD. K eigenfaces are retained

The number of eigenfaces calculated was equal to the number of faces in the training set. Because the eigenfaces represented the principal components of the data, only a small number needed to be retained. This number was chosen heuristically. At a certain point, the data contained in the eigenfaces became noisy and was not significantly beneficial to facial recognition. During the implementation of this project it was found that the first 250 eigenfaces produced accurate recognition results. The first 9 eigenfaces are shown in *Figure 44*.

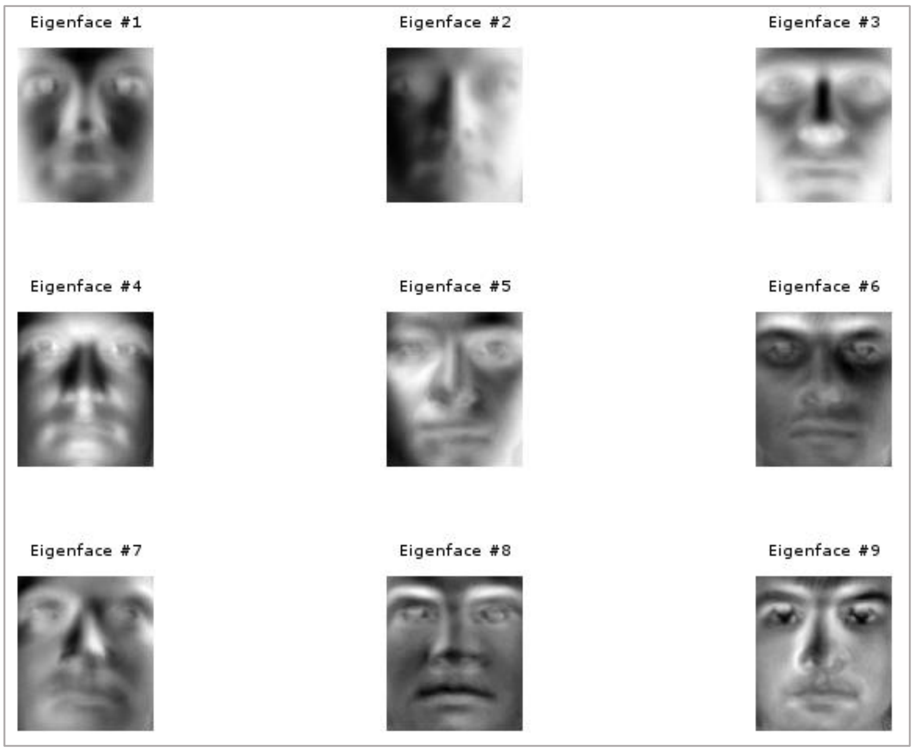


Figure 44 The first 9 eigenfaces calculated using the SVD method

7.3.4 PBI – Calculate Weight Vectors

The final step in preparing the training data was to calculate a weight vector for each normalised image. The weight vector for each of these images was determined by finding the dot product between that image and every calculated eigenface. Thus a set of k values were produced for each image, where the k^{th} value was a weight which corresponded to the k^{th} eigenface. A matrix of weight vectors was produced, as shown below.

$$Weights = \begin{bmatrix} Face1Weight1 & \cdots & FaceMWeight1 \\ \vdots & & \vdots \\ Face1Weightk & & FaceMWeightk \end{bmatrix}$$

At this point, these images were said to be projected into face space.

7.3.5 PBI – Reconstruct a Known Face

Each face in the training set could be reconstructed using the eigenfaces and the images corresponding weight vector. This is an important point – the initial large dataset could be reconstructed using the much smaller set of eigenvectors. For example, the *Extended Yale Face Database B* is 600 megabytes when persisted to disk. Taking the top 250 eigenvectors and persisting them to disk produced a file of size 60 megabytes.

Reconstructing a face from its weight vector was relatively straightforward. It was achieved by multiplying the weight vector (which had been calculated for the face that was being reconstructed) by the eigenfaces, and adding to the result the average face which was removed at the beginning of the process. The result was a column vector which when reshaped to the original dimensions produced an image very close to the original face.

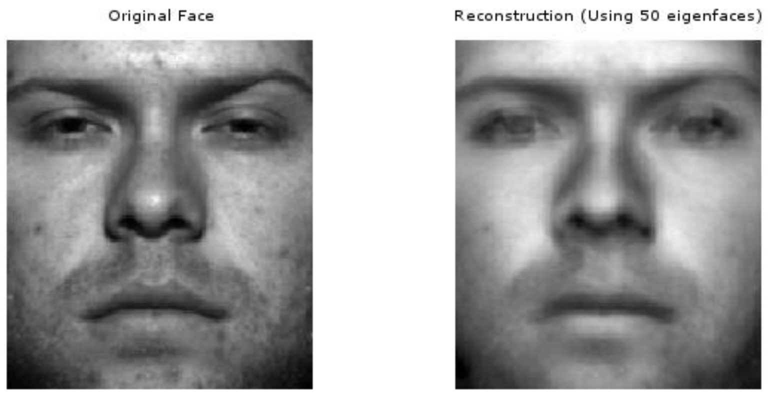


Figure 45 Reconstruction using 50 eigenfaces

This concept was important when developing the recogniser. The visualisation of reconstructed faces allowed one to choose a suitable number of eigenfaces to retain such that the training set was represented appropriately. It lowered the probability of retaining too few eigenfaces for successful recognition. It can be seen from *Figure 45* and *Figure 46* that reasonably good results were obtained when using 250 eigenfaces.

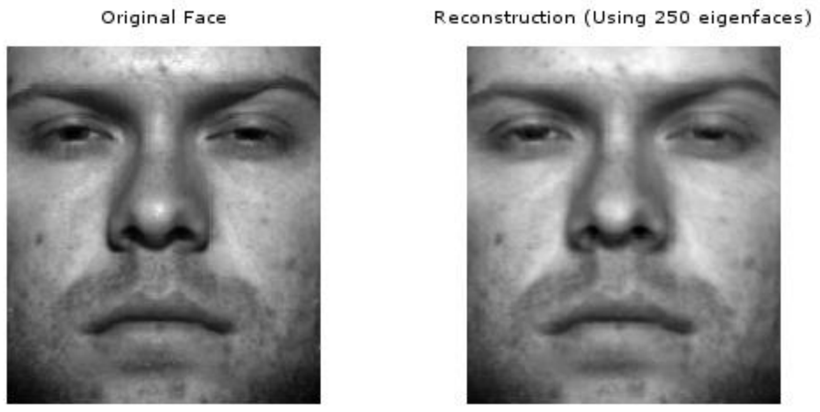


Figure 46 Reconstruction using 250 eigenfaces

7.3.6 PBI – Recognise an Unknown Face

At this point of the sprint, the core functionality of the facial recogniser was completed. The next PBI was to implement functionality in Octave to attempt recognition of an unknown face. To achieve this, the unknown facial image had to be projected to the face space. This was accomplished similar to what has been discussed above.

The unknown face, when loaded into memory in Octave, was reshaped to a column vector. This column vector was then reduced by subtracting the previously calculated average face from it. The normalised column vector of the unknown image was then projected to the face space by calculating its weight vector. This was completed exactly as described in Section 7.3.4 – the dot product between the unknown face and the previously found eigenfaces was calculated.

Once the facial image was projected into face space it could be determined if a match existed. A decision was made using a Euclidean distance metric. The Euclidean distance was calculated between the weight vectors of all known faces and the weight vector of the unknown face. The **norm** function was used to calculate the Euclidean distance in Octave – the Euclidean distance between two vectors is the two-norm of their difference. (StackExchange, 2013)

```
Source File: nearestMatchEuclideanDistance.m
for i=1:columns(weights)
    ED(1, i) = norm(weights(:, i) - weightOfUnknownFace, 2);
endfor

[distance idx] = min(ED);
```

Snippet 7 Calculating the closest facial match by finding the minimum Euclidean distance

The closest known weight vector to the unknown face represented the closest match. Programmatically, this was determined by finding the known weight vector which has the minimum Euclidean distance from the unknown face – shown in *Snippet 7*. A threshold value may be pre-specified to determine if the unknown face belongs to either a person that is completely unknown to the recogniser, or is possibly not a facial image at all, should the distance from the closest match be outside this threshold value.

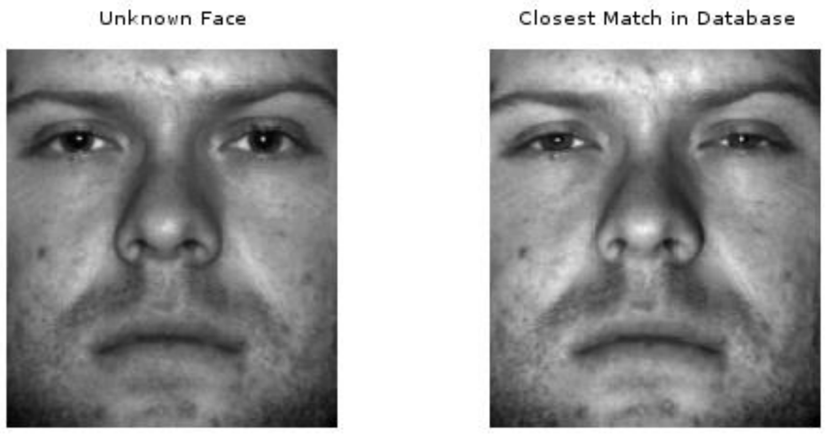


Figure 47 Recognition in Octave - closest match is the same face but not the same image

7.3.7 PBI – Persist Data

The start-up time of the completely implemented process for preparing the facial recognition system, from loading the training dataset to calculating the weight vectors, was observed to take between 15 and 20 minutes. To improve the time efficiency of the system persistence was introduced. Octave scripts were developed that allowed the data that was calculated to be saved to disk and reloaded as required. Namely, the following data was of importance:

- The set of original faces,
- The set of reduced faces,
- The calculated eigenfaces,
- The average face,
- The weights matrix.

The data was persisted to disk using the HDF5 file format as shown in *Snippet 8*. HDF5 was developed by the National Centre for Supercomputing Applications and is designed for storing large volumes of numeric data.

By saving the data to disk, and reloading it when required, the start-up time for the recogniser was reduced to about 10 seconds.

```
Source File: saveMatrixData.m
function saveMatrixData(M, fileName)
  % Save the matrix to disk using the HDF5 format
  % Reset directory when complete
  dir = pwd();

  % Create output directory if it doesn't exist
  if !isdir("C:/FacialRecognition/data")
    mkdir("C:/FacialRecognition/data");
  endif

  cd("C:/FacialRecognition/data");
  save("-hdf5", fileName, "M");

  cd(dir);
endfunction
```

Snippet 8 Writing a matrix to disk using HDF5 in Octave

The Yale training data was also persisted – this data never changes and loading it from raw image files while training the recogniser had been observed to require a number of minutes. The data was persisted using the HDF5 format which, as above, reduced its load time from a number of minutes to a number of seconds.

7.3.8 Issues Encountered

A number of issues were encountered during Sprint 2 while implementing the PBIs that were discussed above. Some of the issues encountered were considered notable and are thus documented.

7.3.8.1 Switching to 64-bit Octave

During development with Octave (32-bit), an issue was encountered while calculating the eigenvalues. During execution the interpreter halted and displayed the message “out of memory or dimension too large for Octave's index type”. This issue became a blocking point for a considerable period of time. At one point considerations were made to implement the recogniser using the R statistical language where experimentation had encountered no issues while calculating the eigenvectors of an equally sized dataset.

The error was ultimately overcome by switching to a 64-bit build of Octave. From a performance perspective the 32-bit build was observed to reach its maximum memory usage around 250mb. In comparison, the 64-bit build was observed to make extensive use of memory during calculations, regularly using up to 2.5GB while also consuming the full capabilities of a single processing core (Octave does not directly support multithreading).

No further issues related to memory consumption or Octave’s index type were encountered during development.

7.3.8.2 PCA vs SVD

As discussed in Section 7.3.3, eigenfaces calculated using the PCA methodology failed in both reconstruction of known facial images and recognition of unknown facial images. The true reasoning for this is not determined. It is speculated by some that the calculation of the covariance matrix introduces additional noise. This, however, is unconfirmed. It was observed that all facial reconstructions produced the exact same image when using the PCA method to calculate eigenfaces. This was despite the eigenfaces appearing, when visualised, to have been calculated correctly.

Again, a considerable time period (a number of days) was spent on this issue. There were a number of bugs within the code which were discovered and rectified but with no positive effect on the overall results of the algorithm.

The SVD method of determining eigenfaces was implemented after reading the work of Philip Wagner, as mentioned. The eigenfaces calculated using this method produced a successful facial reconstruction. As a result, the SVD method was utilised.

7.3.8.3 Efficiency

Issues with efficiency were mentioned is Section 7.3.7 and were the primary basis for using a data persistence model in this application. The recogniser was observed to take a significant time period at start-up to complete all required calculations and reach a state where it was ready to attempt facial recognition. This was not sustainable – it would have been impractical to wait a large time period each on each instance that the application was executed before it became usable.

By persisting the data to disk, start-up times were reduced to a few seconds. This has obvious benefits and means the application is almost immediately usable. The eigenface algorithm can, however, be re-executed should the dataset change. For example, the addition or removal of facial images to or from the training set would necessitate this. Retraining the recogniser became a requirement during later sprints as new users were inserted into the dataset.

7.4 Sprint 3 – Facial Detection

In order to achieve facial recognition a source image had to be captured. The majority of the data captured in the source image was background data or noise which would have inhibited the recogniser. To overcome this a facial detection algorithm was utilised to allow facial images to be extracted from the original source image.

7.4.1 PBI – Implement Facial Detection

The core area of this project was facial recognition for which a full algorithm was implemented. Facial detection, on the other hand, was implemented using a pre-existing library. At the beginning of this sprint two notable libraries were considered. These were:

- OpenCVSharp,
- EmguCV.

Both are .NET wrappers for the OpenCV computer vision library. EmguCV was chosen as a result of its efficient maintenance schedule and more intuitive syntax.

EmguCV is released as an installation package which was installed under a Windows environment. Utilising it within the .NET solution required both referencing a number of DLLs and ensuring that all other required .DLLs were copied to the output folder each time the project was built.

Following successful configuration, the EmguCV API was used to implement facial detection. A pre-defined cascade classifier is bundled with EmguCV and was applied to the solution. A cascade classifier can be applied to detect faces in a given image, as described in Section 2.2.1. The facial detection process is shown in *Snippet 9*. Its results were returned as an array of rectangles wherein the bounds of each rectangle enclosed an area that was occupied by a face in the given source image. Using these rectangles, any facial images which existed within the source image provided to the facial detector were extracted. The extracted facial images were then utilised as either an unknown image when invoking the facial recogniser (see Section 7.7.2) or when adding images of a subject to the facial database (see Section 7.5).

For visualisation purposes, the rectangles returned by the facial detector were drawn onto the source image on the user interface utilising a .NET graphics object, as shown in *Figure 48*.

```
Source File: FacialDetector.cs
public Rectangle[] DetectFaces(Bitmap image)
{
    var emguImage = new Image<Gray, byte>(image);

    var classifier = new CascadeClassifier(this.ClassifierPath);

    var faces = classifier.DetectMultiScale(emguImage,
       this.ScaleFactor,
       this.MinimumNeighbours,
       this.MinimumSize,
       this.MaximumSize);

    return faces;
}
```

Snippet 9 Using EmguCV to perform facial detection

The cascade classifier object was provided with the path to a Haar classifier in its constructor. In this case the default frontal face classifier which is bundled with EmguCV (and provided in XML format) was used. The classifier was invoked using the **DetectMultiScale** method. This method required a numbers of parameters, the values of which had a notable impact on the outcome of the facial detection process. (StackOverflow, 2013) The parameters required were; in order:

1. **Image**

   The source image on which facial detection will be executed.

2. **Scale Factor**

   The factor by which image size is scaled during each step of detection. The face model has a predefined fixed size (specified in the classifiers XML) and this is the size of face that is detected in the image. By rescaling the source image the face sizes can be reduced making them more likely to match those of the face model and be detected by the algorithm. A lower scale factor has a higher chance of detecting faces, though at increased computing costs.

3. **Minimum Neighbours**

   The number of neighbours a candidate rectangle should have for it to be retained and classed as a face. Higher values increase the quality of detected images but may reduce the overall number of detections.

4. **Minimum Size**

   The minimum size of a detected face in pixels. Faces detected with smaller dimensions are ignored.

5. **Maximum Size**

   The maximum size of a detected face in pixels. Faces detected with larger dimensions are ignored.

The final values used were chosen based on trial and error following manual testing of the facial detection functionality. These are the values which produced the most accurate and reliable facial detection results during testing.

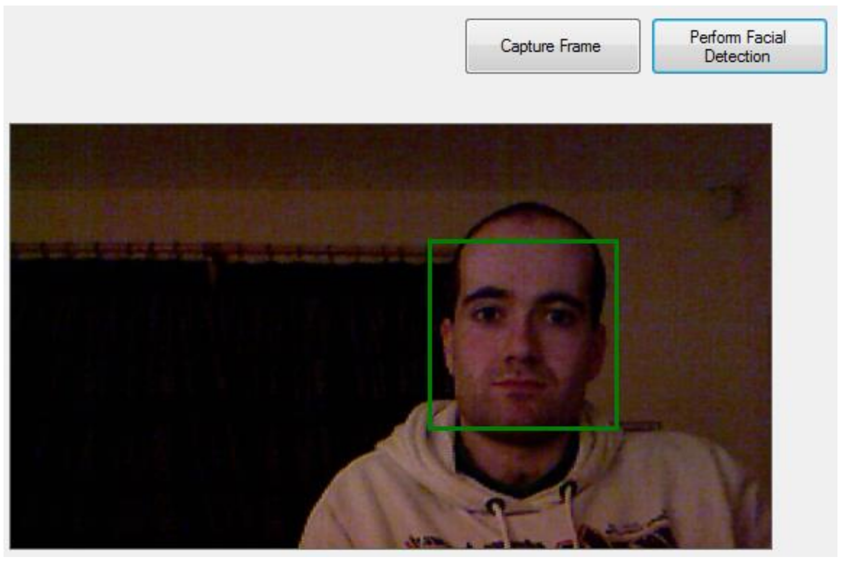


Figure 48 Detection of a single facial image, as shown on the UI

7.5 Sprint 4 – CouchDB

The facial recogniser was trained using the *Extended Yale Face Database B*, as discussed in Section 7.3. This allowed the recogniser to be built without the added burden of manually acquiring training images which would have taken a considerable amount of time. However, to correctly assess the research criteria of this paper, a customised database was required. The purpose of this database was straightforward: to allow one to store identification and facial data for any number of individuals, such that their facial images could be added to the recogniser and subsequently used in the recognition of those persons.

It was decided to use CouchDB as the database provider. It is a modern NoSQL document store that is lightweight and scalable. CouchDB uses JSON formatting for both storage and the communication of query results. It is important to note that ad-hoc querying is not supported – instead CouchDB uses views produced by map-reduce functions to access stored documents. CouchDB documents have two required fields, the ID and the revision number. The revision number is used when updating documents as a means of ensuring data consistency. (Redmond & Wilson, 2012)

The most influential feature for which CouchDB was chosen for this project was its support for attachments. Each document could have any number of attachments which could be of any type of file format. This was a desirable model for this project – any number of facial images of a person could be attached to the database document that identified that person.

7.5.1 PBI – Design and Implement Required Models

Data models had to be developed prior to implementing the database layer. For the purpose of this project only one model was required: that of a person. The model was designed with simplicity in mind, and required only a few fields. It is shown in *Snippet 10*.

```
Source File: Person.cs

public class Person
{
    public String Id { get; set; }
    public String Forename { get; set; }
    public String Surname { get; set; }
    public List<Image> Images { get; set; }

    public Person()
    {
        this.Images = new List<Image>();
    }
}
```

Snippet 10 The Person model

As discussed in the introduction to this section, each CouchDB document, by default, has a revision number field. This field is important when updating. To successfully update an existing document, the data being passed to CouchDB must provide the most recent revision number. In the case that it doesn't, the database will refuse to accept the update.

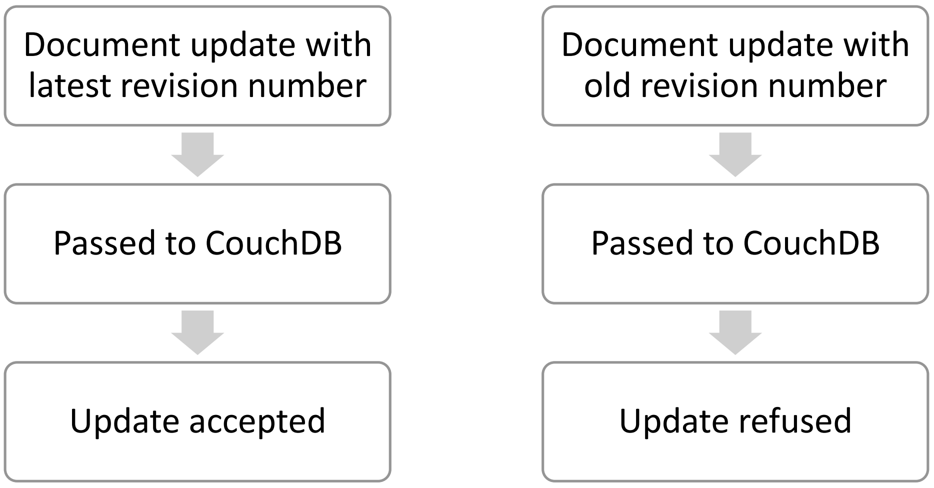

Document update with latest revision number
Passed to CouchDB
Update accepted
Document update with old revision number
Passed to CouchDB
Update refused

Figure 49 CouchDB and Revision Numbers

The original person model did not contain a field for the revision number. To overcome this obstacle, a second model was created that extends the person model: **PersonCouchDB**. This model supplied the extra revision field which was used only by the CouchDB class when updating already existing documents. It is shown in *Snippet 11*.

```
Source File: PersonCouchDB.cs
public class PersonCouchDB : Person, ICouchDocument
{
    public String Rev { get; set; }
}
```

Snippet 11 The PersonCouchDB model

7.5.2 PBI – Determine Generic Database Functionality

It was desirable to implement a database layer that was loosely coupled to the application. To achieve this in an uncomplicated manner it was decided to define the database functionality within an interface, as shown in *Snippet 12*. All calling code was subsequently wrote against this interface as opposed to the concrete implementation. This meant the calling code had no knowledge of the concrete database or its implementation – its only requirement was to honour the contract as defined by the interface.

Defining the database functions generically allowed for the concrete database implementation to be changed, if required, without having to modify any of its calling code.

```
Source File: IDatabase.cs
public interface IDatabase
{
    void CreateDatabase(String database);
    void DeleteDatabase(String database);
    bool Store(Person person);
    bool Update(Person person);
    Person Retrieve(String id);
    List<Person> RetrieveAll();
}
```

Snippet 12 The database interface

7.5.3 PBI – Implement Concrete Database Functionality

To implement the concrete database a package was required to allow the .NET application to interface with CouchDB. The *DreamSeat* package was chosen. It is an open source package that allows for both synchronous and asynchronous communication with the database. *DreamSeat* provides full support for documents and attachments. (GitHub, 2013)

A number of private helper methods were required to fully achieve the desired functionality. These methods ensured that the specified Couch database existed and also verified that the view required to retrieve all documents from that database was created.

```
Source File: CouchDatabase.cs
var designDocument = new DreamSeat.CouchDesignDocument(DesignDocumentName);
var couchView = new DreamSeat.CouchView(AllDocumentsViewDefinition);

designDocument.Views.Add(AllDocumentsViewName, couchView);
this.Database.CreateDocument(designDocument);
```

Snippet 13 Creating a view within a design document

The *DreamSeat* package adds attachments to CouchDB documents using memory streams. This required a private helper method that implemented functionality for the conversion of images from the generic Image type available in .NET to a byte array. This method is shown in *Snippet 14*. A similar method was required when retrieving attachments to reconstruct an image from the returned memory stream.

```
Source File: CouchDatabase.cs
private byte[] ImageToByteArray(Image image)
{
    var stream = new MemoryStream();
    image.Save(stream, System.Drawing.Imaging.ImageFormat.Bmp);

    return stream.ToArray();
}
```

Snippet 14 Converting an image to a byte array

An interesting aspect of this implementation was the use of dynamic types in C#. When retrieving all people from the database a view was required which returned the results of a map-reduce function in JSON format. Instead of being required to provide a model which mapped and allowed access to the data of this result object, one alternatively used a dynamic expression whose definitions were resolved at runtime. This allowed a JSON response from CouchDB to be cleanly de-serialised and converted to a list of person models, shown in *Snippet 15* below. Remember here that the method defined in the interface to retrieve all persons from the database returned a list of people.

```
Source File: CouchDatabase.cs
for (int i = 0; i < viewRows.Count; i++)
{
    var currentDocument = viewRows[i];
    var documentValues = currentDocument.value;

    var curPerson = new PersonCouchDB();
    curPerson.Id = (string)documentValues._id;
    curPerson.Rev = (string)documentValues._rev;
    curPerson.Forename = (string)documentValues.forename;
    curPerson.Surname = (string)documentValues.surname;
    curPerson.Images = this.RetrieveAttachments(curPerson.Id);
    result.Add(curPerson);
}
```

Snippet 15 Traversing view results with dynamic expressions

7.5.4 PBI – Design a User Interface for Database Interaction

A user interface was required for the database section of this project. Once the database functionality was implemented and tested a visual means was provided to allow for both adding new users to the database and updating existing users. Taking this requirement into account, a user interface was created with the following components:

- A data grid which was populated with existing users retrieved from the database,
- Camera functionality to add facial images to a user's record,
- Fields to manually enter user information for both new and existing users,
- A section to view, iterate through and delete the facial images associated with a user.

The database interface is shown below using two separate screenshots (*Figure 50* and *Figure 51*) – otherwise the image would be distorted and unreadable.

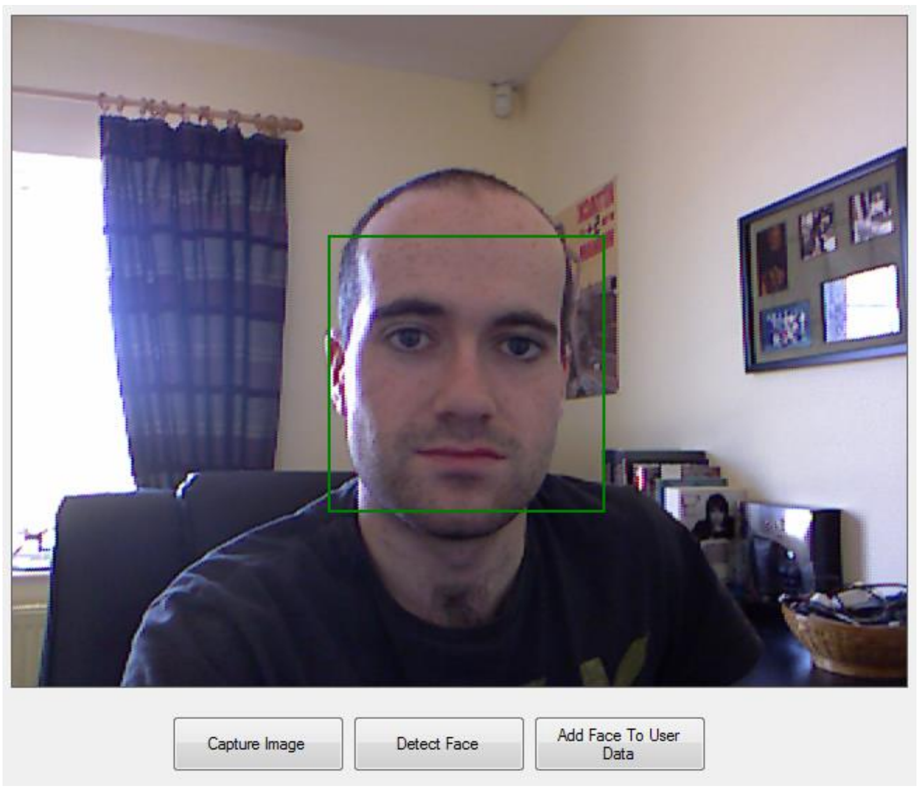
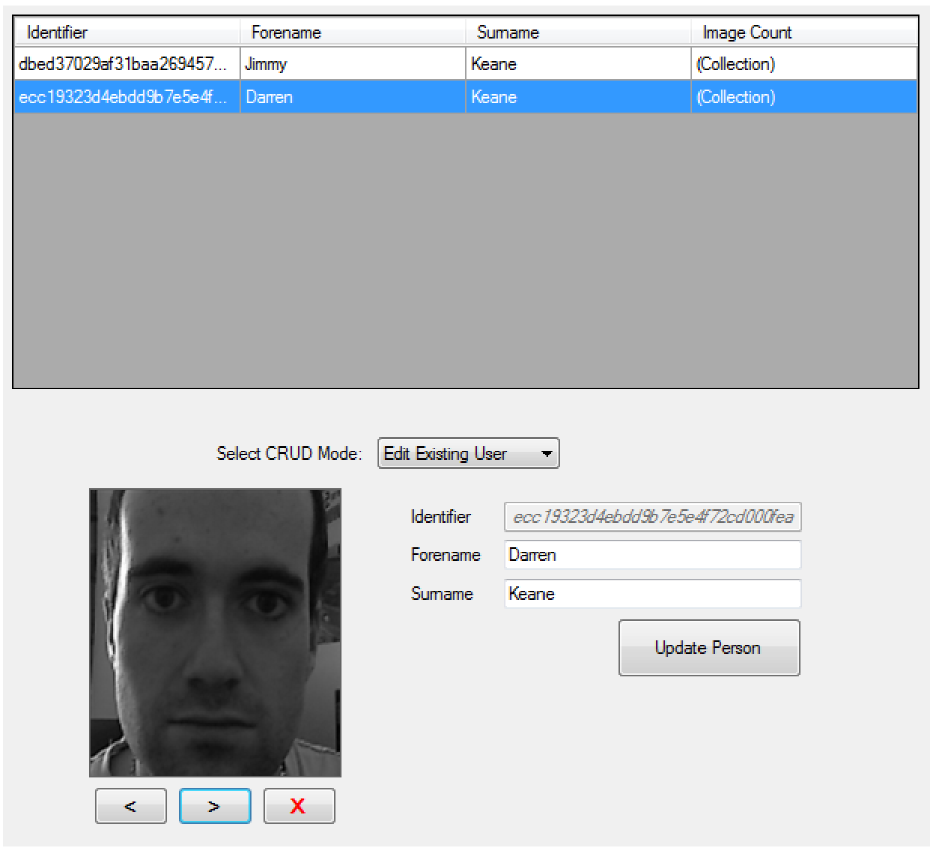


Figure 50 The image capture section of the database UI



Identifier	Forename	Surname	Image Count
dbed37029af31baa269457...	Jimmy	Keane	(Collection)
ecc19323d4ebdd9b7e5e4f...	Darren	Keane	(Collection)

Select CRUD Mode: Edit Existing User

Identifier	ecc19323d4ebdd9b7e5e4f72cd000fea
Forename	Darren
Surname	Keane

Figure 51 The CRUD section of the database UI

7.5.5 – Issues Encountered

Some issues were encountered during this sprint, specifically while developing a concrete CouchDB class using the available packages for .NET. These issues are documented below.

7.5.5.1 CouchDB Packages for .NET

There are a wide number of CouchDB packages available for the .NET platform. The most prominent of these at the time of development was a package called *MyCouch*. This is an asynchronous CouchDB client developed primarily by Daniel Wertheim. (Wertheim, 2014) Due to the asynchronous nature of the client coupled with the authors scarcity of experience with asynchronous programming models, it was determined that a synchronous solution would be better suited to this implementation.

The *LoveSeat* CouchDB package was first considered as it had been utilised by the author during a previous project. *LoveSeat*, however, produced some unexpected issues. It has a dependency on the popular *Newtonsoft.JSON* package. This package is a vital requirement for all CouchDB clients due to the databases prominent use of JSON for both document storage and the communication of view results. *LoveSeat* proved incapable of automatically resolving this dependency and manual attempts to resolve it proved unsuccessful and time consuming.

A fork of *LoveSeat*, *DreamSeat*, was ultimately chosen in its place. *DreamSeat* allows for synchronous communications while providing support for documents, attachments and views, the three most significant attributes of CouchDB that were required by this implementation. (GitHub, 2013) No issues were encountered with JSON formatting throughout the usage of this package.

7.6 Sprint 5 – Redis Integration

The facial recogniser, as discussed in Section 7.3, was implemented using GNU Octave. The Octave application handles the training of the facial recogniser and the determination of the closest facial match for an unknown facial image. The front end components of the overall project were implemented using C#, which will be discussed in Section 7.7, and the database was implemented using CouchDB as described in Section 7.5. Communication with the database was conducted exclusively using the C# application. Therefore, a means was required to allow the C# application to communicate with the recogniser which was implemented in Octave.

This problem was solved using Redis. Redis is an open source key-value data store. It was chosen primarily as it is amongst the few data stores for which an Octave driver is available. Redis has a number of properties that enhanced its suitability to provide the integration between Octave and C#. It is exceptionally fast – it can handle up to 100,000 set operations per second. While classified as a key-value store at its most rudimentary level, Redis supports many advanced data structures. (Redmond & Wilson, 2012) Amongst these structures are lists which will be utilised in this implementation.

Following the design of the communication protocol to be used – there were two distinct stages required in its implementation. The first of these was in Octave which listens for requests sent via Redis, executes them and sends a response. The second was in C# where requests were sent via Redis and a response was awaited.

7.6.1 PBI – Redis Protocol

A Redis messaging protocol was required. It defined the Redis keys and their corresponding values that were necessitated to permit communication. The protocol that was designed allowed request and response messages to be sent between Octave and C# to invoke pre-defined functionality. The functionality required was as follows:

- Redis keys to identify the requests being sent to the recogniser and to hold their response data, if required,
- Redis keys to hold the image data and their corresponding labels as stored in the Couch database. These keys were used when retraining the recogniser,
- A Redis key to hold the current state of the recogniser.

Redis Integration: Request and Response Keys

Redis Key	Redis Value	Function
		Request
facial.request.code	50	No available request
	100	Request facial recognition
	200	Request reload recogniser data from disk
	300	Request save recogniser data to disk
	400	Request retrain recogniser
facial.request.data	Varies	Hold data associated with the request
		Response
facial.response.code	50	No available response
	100	Response indicating request succeeded
	200	Response indicating request failed
facial.response.data	Varies	Hold data associated with the response

Table 12 Recogniser request and response keys along with their functions

The Redis request keys, shown in *Table 12*, allowed the functionality of the recogniser to be invoked from the C# application. For example, when one wished to perform facial recognition then the value of **facial.request.code** key was set to 100. For this particular function the Octave application had to be supplied with an unknown facial image on which to attempt recognition. The unknown image had to be provided as the value of the **facial.request.data** key. When Octave completed the execution of the request it set the values of the response code and response data keys as appropriate.

Redis Integration: Database Keys

Redis Key	Redis Value	Function
facial.database.labels	The Couch identifier values of all people in the database	A list that held the labels of all individuals stored in the Couch database
facial.database.data	All images in the Couch database	A list that held the images of all individuals in the Couch database. The images were stored in comma delimited string format

Table 13 Recogniser database keys and their functions

The Redis database keys which are shown in *Table 13* were utilised in conjunction with a request to retrain the recogniser. Each of the database keys represented a Redis list. All images in the facial database were pushed onto the list held by the **facial.database.data** key. The corresponding database identifier for each image was pushed onto the list held by the **facial.database.labels** key. When Octave received a request to retrain the recogniser it popped the values from these lists, as required, until all the available data has been consumed. It was decided to store each database image in the Redis list as a string where the individual pixel values were separated by a delimiter. This solution allowed both Octave and C# to access the image data and efficiently marshal it to any format as required.

Redis Integration: Recogniser Status Key

Redis Key	Redis Value	Function
facial.recogniser.status	100	Indicates that the recogniser is available to execute requests
	200	Indicates that the recogniser is busy executing a request

Table 14 Recogniser status key and its functions

The final Redis key utilised was **facial.recogniser.status**. It is outlined in *Table 14*. This key was controlled by the Octave application. The value of the key indicated the state of the recogniser. It allowed the C# application to determine whether or not the recogniser was available to service a request.

7.6.2 PBI – Implement Redis in Octave

An appropriate package was required to provide Redis functionality in Octave. An open source package named *go-redis* has been developed and distributed via GitHub by Markus Bergholz. It provides support for some of the most common Redis commands such as *get* and *set*. (Bergholz, 2015) The version of the *go-redis* package that was utilised has a dependency upon an Octave package called *instrument-control* which provides low level networking functionality.

An Octave function was developed to listen for Redis requests, as shown in *Snippet 16*. It continuously polled the **facial.request.code** key until a request was received. Upon receipt of a request it was passed to a dedicated handler function. Here, the request was parsed and executed. Upon successful execution of a request a positive response was indicated using the response code key and any required data was stored in the response data key.

Otherwise a negative response was indicated using the response code key and details of any errors which occurred were stored in the response data key.

```
Source File: redisListener.m
while(!done)
       requestCode = redisGet(R, RequestCodeKey);

     if(requestCode != NoData)
       redisSet(R, RecogniserStatusKey, RecogniserBusy);
       printf("Request Received: %d", requestCode);

       % Store the Redis request code and data
       recogniserData.requestCode = requestCode;
       recogniserData.requestData = redisGet(R, RequestDataKey);

       % Pass the request to the handler.
       % It will be executed and a response will be sent
       recogniserData = redisRequestHandler(R, recogniserData);

       % Prepare Redis to receive new requests
       redisSet(R, RequestCodeKey, NoData);
       redisSet(R, RecogniserStatusKey, RecogniserAvailable);
     endif
endwhile
```

Snippet 16 The Redis listener checked for requests and sent them to the handler

As described in Section 7.6.1, Redis lists were desired when sending the contents of the Couch database to Octave. However, the *go-redis* package utilised did not provide functionality to pop an item from a list. It was decided that the appropriate solution would be to extend the Redis package to provide this functionality. A function was provided by the package to execute any command against Redis and retrieve its response in raw format. Utilising this function an LPOP command was executed against Redis. The raw response received from the command was parsed with reference to the bulk strings section of the

```
Source File: redisListLPOP.m
responseType = substr(response, 1, 1);

if (responseType == '$')
    responseComponents = strsplit(response, "\r\n");

    % responseComponents format:
    % (1, 1) = $length
    % (1, 2) = string value to be retrieved
    % (1, 3) = CRLF

    value = responseComponents(1, 2);
    value = cell2mat(value);
else
    error("Unexpected response type from LPOP command");
endif
```

Snippet 17 Parsing and processing a response from the Redis LPOP command

Redis protocol. (Redis, 2015) The protocol indicated that a successful response began with a *$* character on its first line which was followed by the number of bytes composing the response string. The second line of a successful response held its data. *Snippet 17* shows how a response from the LPOP command was processed.

A facial recognition request sent via Redis must include the facial image data in the **facial.request.data** key. Upon detection of the recognition request by Octave it retrieved the unknown facial data from Redis and classified it as described in Section 7.3.6. Assuming that no errors occurred a successful response code was indicated by Octave and the label of the closest facial match was stored using the **facial.response.data** key. The labels used were identical to the database identifier of the person and allowed them to be retrieved from the Couch database.

7.6.3 PBI – Implement Redis in C#

Similar to Octave, an appropriate Redis package was required prior to implementing the Redis integration in C#. The *StackExchange.Redis* package was chosen to provide access to Redis. It is a high performance Redis client for .NET that is developed by Stack Exchange and is currently being utilised by websites such as Stack Overflow. Amongst its features are a dual programming model that allows it to be used in either a synchronous or an asynchronous manner. (StackExchange, 2015)

```
Source File: RecogniserCode.cs
public enum RecogniserCode
{
    // Represents a key which contains no data
    NoData = 50,

    // The requests that can be sent to Octave
    RequestRecognition = 100,
    RequestReload = 200,
    RequestSave = 300,
    RequestRetrain = 400,

    // A request will either succeed or fail
    ResponseOk = 100,
    ResponseFail = 200
}
```

Snippet 18 The enum developed to hold the Redis integration codes

An enum, shown in *Snippet 18*, was implemented to hold the message codes defined in the integration protocol. Utilising an enum allowed for a high level of abstraction. The named constants prevented the appearance of magic numbers, increased type safety and produced much more readable code that had a higher level of maintainability.

All communications that were sent via Redis had a specific structure – each message required at a minimum a code, and some also required specific data. This was modelled using a class in C#, shown in *Snippet 19*. An overloaded constructor allowed for the insanitation of message objects that best suited the requirements of the calling code.

```
Source File: RedisMessage.cs
public class RedisMessage
{
    public int Code { set; get; }
    public String Data { set; get; }

    public RedisMessage()
    {
        this.Code = (int)RecogniserCode.NoData;
        this.Data = ((int)RecogniserCode.NoData).ToString();
    }

    public RedisMessage(int code)
    {
        this.Code = code;
        this.Data = ((int)RecogniserCode.NoData).ToString();
    }

    public RedisMessage(int code, String data)
    {
        this.Code = code;
        this.Data = data;
    }
}
```

Snippet 19 The RedisMessage class

Finally, a Redis connection class was developed. This class implemented all the functions that were required to allow communication with Octave via Redis, including:

- Initialising a connection to a Redis server using a specified hostname and port number,
- Sending the code and data of a request to Redis,
- Retrieving response data from Redis, parsing it and returning it to the calling function as an instance of RedisMessage.

Snippet 20 shows a section of the C# method which is utilised when sending a request to Redis.

```
Source File: RedisConnection.cs
public bool SendRequest(RedisMessage message)
{
    var transaction = this.RedisDatabase.CreateTransaction();

    transaction.StringSetAsync(this.FacialRequestCodeKey, message.Code);
    transaction.StringSetAsync(this.FacialRequestDataKey, message.Data);

    return transaction.Execute();
}
```

Snippet 20 A section of the method that send a request to Redis

Upon sending a request it was detected by the Redis listener in Octave, as discussed in Section 7.6.2, and executed appropriately. The response was awaited in C#. A method tasked with receiving the response message continuously polled the **facial.response.code** key until either a response was received, or a pre-specified timeout interval was reached. The timeout value was indicated when instructing C# to wait for a response. This value must be suitable to the situation as, for instance, a retrain recogniser request took up to thirty minutes to complete. *Snippet 21* shows a section of the C# method that was implemented to await a response from Redis.

```
Source File: RedisConnection.cs
while (watch.ElapsedMilliseconds <= timeout && !responseReceived)
{
    var responseCodeString = this.RedisDatabase.Get(this.FacialResponseCodeKey);

    if (responseCodeString != null)
    {
        var responseCode = int.Parse(responseCodeString);

        if (responseCode != (int)RecogniserCode.NoData)
        {
            var responseData = this.RedisDatabase.Get(this.FacialResponseDataKey);
            response = new RedisMessage(responseCode, responseData);
            responseReceived = true;
        }
    }
}
```

Snippet 21 A section of the method that awaits a response from Redis

7.6.4 Issues Encountered

7.6.4.1 Octave and Redis

The package utilised to communicate with the cache from Octave, *go-redis*, was not free of issues. Though no significant blocking issues were encountered, some adjustments had to be made. Firstly, as discussed in Section 7.6.2, a function had to be developed as an extension of the package to allow popping items from a list.

Another issue was encountered when keys do not exist or have a null value in Redis. Though Redis itself returns a null value if a key doesn't exist, the Octave package utilised was unable to handle this and an infinite loop would occur. To overcome this issue a *no data* value was specified, as seen in the protocol. It was programmatically ensured that the request and response keys, when not being utilised, were set to hold this value. This approach prevented infinite loops in Octave.

Also, the *go-redis* package did not provide support for transactions. An attempt to extend the package to allow transactions proved futile as, similar to above, infinite looping issues occurred. This created a problem – when a response was sent from Octave to Redis, both the response code and response data keys were updated. Because a transaction could not be used the C# application was receiving the response code, and then reading the data values, before Octave had completed updating them.

A workaround was used to overcome this issue. As C# listened for the response key to be updated, it was decided to first update the data key when sending a response. Then, the response code key was updated. Using this specific order to update the keys ensured that the both the code and data keys were always set despite the lack of support for Redis transactions in Octave.

7.6.4.2 Redis Packages for C#

It was initially intended to utilise the *ServiceStack.Redis* package for integration purposes in C#. However, with the integration partially completed, it was discovered during testing that the latest version of the package enforces a usage limit of 6000 requests per hour, as detailed on the ServiceStack website. (ServiceStack, 2015) Though a previous release of the package, which does not enforce a usage limit, could have been used for the project it was decided to instead migrate the code-base to the *StackExchange.Redis* package. Both packages are fully featured – the only difference from an implementation perspective being the syntax of their API's.

7.7 Sprint 6 – C# Functionality

Functionality was required in the C# application to bring the overall architecture of this project together. These functions included implementing both facial image normalisation and a class that provided access to the recogniser's functionality.

7.7.1 PBI – Normalise Image Data

As discussed in Section 7.3, normalised images were used when implementing facial recognition in Octave. It is a requirement of photometric recognition that the same image format must be provided when invoking all functions of the recogniser. In this case all images were normalised to the following format:

- A width of 168 pixels,
- A height of 192 pixels,
- A colour format of 8-bit grayscale.

These image dimensions were consistent with those that were used in the *Extended Yale Face Database B*. They were neither too small to degrade detail in a facial image nor too large to have a notable performance impact on the recogniser. Grayscale images of size 8 bits were chosen as they could be processed must faster than colour images which contain multiple channels. An 8 bit grayscale image has pixel intensity values in the range 0 to 255. Each pixel was stored using one numeric value which represented its intensity level.

```
Source File: FacialImageNormaliser.cs
public abstract class FacialImageNormaliser
{
    public Image NormaliseImage(Image sourceImage, int width, int height)
    {
        Image normalisedImage;

        normalisedImage = this.Resize(sourceImage, width, height);
        normalisedImage = this.SetColormap(normalisedImage);

        return normalisedImage;
    }

    public abstract Image Resize(Image source, int width, int height);
    public abstract Image SetColormap(Image source);
}
```

Snippet 22 Template for image normalisation

The process of image normalisation was implemented in C# using a template method, shown in *Snippet 22* above. The template method defined the steps of the normalisation algorithm and allowed subclasses to implement these steps as required. A concrete photometric normaliser was developed which provided the implementation of the steps of this algorithm to normalise facial images in accordance with the specifications outlined above.

The image was first resized using the graphics class provided by C# - the source image was redrawn into a destination bitmap of size specified by parameters to the method. This process in shown in *Snippet 23*.

```
Source File: PhotometricFacialImageNormaliser.cs
public override Image Resize(Image source, int width, int height)
{
    var resizedImage = new Bitmap(width, height);
    var graphics = Graphics.FromImage(resizedImage);

    graphics.InterpolationMode = InterpolationMode.HighQualityBicubic;
    graphics.DrawImage(source, 0, 0, width, height);
    graphics.Dispose();

    return resizedImage;
}
```

Snippet 23 Concrete resize method

Then, the colour-map of the image was converted to grayscale. There are three methods of achieving this for each pixel in a source RGB image, as outlined by Cook. (Cook, 2009) The first is the **average** method which calculates the average of the red, green and blue components. The second is the **lightness** method which determines the average of the most prominent and least prominent colours. The final method, the **luminosity** method, also calculates an average of the red, green and blue components. This differs from the above average method by weighting each of the red, green and blue components to account for human perception. According to Cook, "we're more sensitive to green than other colours, so

```
Source File: PhotometricFacialImageNormaliser.cs
for (int i = 0; i < source.Width; i++)
{
    for (int j = 0; j < source.Height; j++)
    {
        var pixel = originalImage.GetPixel(i, j);

        var grayscaleValueOfPixel = 0.21*pixel.R + 0.72*pixel.G + 0.07*pixel.B;

        var pixelGrayscale = Color.FromArgb(grayscaleValueOfPixel,
        grayscaleValueOfPixel, grayscaleValueOfPixel);

        result.SetPixel(i, j, pixelGrayscale);
    }
}
```

Snippet 24 Changing the colour-map using the luminosity method green is weighted most heavily". The luminosity method was chosen based on Cook's recommendation that it produces the best overall results. Its implementation is shown in *Snippet 24* above. *Figure 52* shows an image before and after normalisation.

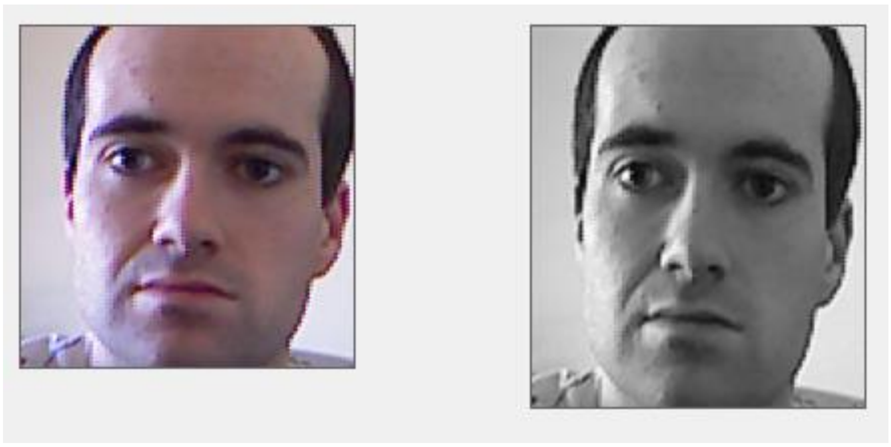


Figure 52 Captured facial image, before and after normalisation

7.7.2 PBI – Implement Recogniser

The operations provided by the facial recogniser included not only classifying a facial image, but also functions to persist, reload and retrain the recogniser. These functions are common to any facial recognition system and were defined within an interface which is shown in *Snippet 25*.

```
Source File: IFacialRecogniser.cs
public interface IFacialRecogniser
{
    Person ClassifyFace(Image facialImage);
    bool SaveSession();
    bool ReloadSession();
    bool RetrainRecogniser(List<Person> people);
}
```

Snippet 25 Recogniser functionality as defined by its interface

A concrete implementation of this interface was provided to allow the invocation of the functions of the photometric Octave recogniser. It implemented the four methods that were defined by the interface along with a number of private helper methods which were unique functions required only by the implemented photometric recogniser. Amongst these methods were a function to marshal facial images to a comma separated format prior to being sent to the recogniser via Redis.

The concrete photometric recogniser class which was implemented, in terms of architecture, is positioned a layer above the **RedisConnection** whose implementation was described in Section 7.6.3. The methods of the recogniser class, when invoked, each carried out their instructions in a specific order which was required to correctly utilise the recogniser.

Firstly, any data which had to be passed to Octave was processed and sent to the cache using the **RedisConnection** class. The recogniser class then invoked a request by constructing an instance of **RedisMessage** and dispatching its contents to the cache, again using the **RedisConnection**. Finally, the recogniser class instructed the **RedisConnection** to wait for a response, specifying the interval for which a response should be awaited before a timeout occurred. The use of the **RedisConnection** was transparent. Any calling code which utilised the functions of the recogniser was not aware of the existence of this connection class.

The concrete recogniser read the response and returned either the response data, if appropriate, or an indication of its success. Exceptions were thrown, as required, if the requests sent to the recogniser were unsuccessful or resulted in errors.

The implementation of the **ClassifyFace** method within the concrete photometric recogniser class is shown in *Snippet 26*.

```
Source File: PhotometricFacialRecogniser.cs
public Person ClassifyFace(Image facialImage)
{
    var imageAsString = this.MarshalFacialImage(facialImage);

    var recogniserRequest = new
              RedisMessage((int)RecogniserCode.RequestRecognition,
                           imageAsString);

    this.RedisConnection.SendRequest(recogniserRequest);

    var timeout = 30000;
    var response = this.RedisConnection.ReceiveResponse(timeout);

    if (response.Code == (int)RecogniserCode.ResponseOk)
    {
        var result = new Person();
        result.Id = response.Data;
        return result;
    }
    else
    {
        throw new Exception(response.Data);
    }
}
```

Snippet 26 Concrete implementation of the ClassifyFace method

7.8 Sprint 7 – Refactoring

The final sprint that was undertaken during the implementation of this project involved refactoring and fine-tuning the code base. This was a requirement – the code was disorganised in places and in the case of the recogniser, suffered from performance inefficiencies. It is important to note that all code snippets shown throughout the implementation chapter are representative of the final product. The author felt that including snippets of the code structure prior to refactoring would have been counter-productive.

7.8.1 PBI – Add Configuration Options to C# Application

The C# application communicated with two data stores – the Couch database and the Redis cache. Prior to refactoring, the connection parameters for these data stores were hard coded within each class that required them. This was refactored – the connection parameters were moved to a central storage location within a settings file from which all calling code retrieved them, as shown in *Snippet 27*. A configuration section that allowed modification of these settings was added to the user interface.

```
Source File: StartupController.cs
private void ReadApplicationSettings()
{
    this.CouchDBHost = Properties.Settings.Default.CouchDBHost;
    this.CouchDBPort = Properties.Settings.Default.CouchDBPort;
    this.CouchDatabaseName = Properties.Settings.Default.CouchDatabaseName;
    this.RedisHost = Properties.Settings.Default.RedisHost;
    this.RedisPort = Properties.Settings.Default.RedisPort;
}
```

Snippet 27 Reading the default settings from a configuration file

7.8.2 PBI – Improve Recogniser Efficiency

It was noted in Section 7.3 that the recogniser suffered from efficiency issues during training. These issues were reduced using persistence. A modification was made which further improved efficiency and reduced training times to a minimum. The *Extended Yale Face Database B* was previously fully read from disk, image by image, each time it was required. Instead, it was read once and then persisted using the HDF5 algorithm. This change notably increased the pace at which training the recogniser completed – reading the Yale database from a persisted file that used HDF5 took a matter of seconds. Previously, it took a number of minutes. The training times observed for the completed recogniser averaged about twenty minutes.

7.8.3 PBI – Refactor C# Application

A full refactor of the C# application was completed. This was undertaken for a number of reasons. Firstly, the author felt that the code required refactoring to consistent coding standards. Secondly, a number of observations of the code base were discussed during code reviews which indicated that both the initial coding standards used and the naming of some classes caused confusion. It was emphasised, in particular, that the code base was difficult to understand when read by those who were unfamiliar with it.

During refactoring appropriate class names were used. The **RedisConnection** class, for example, was refactored. This class was initially, and confusingly, named **OctaveInterface**. Coding standards were followed. For C#, the application code was modelled upon the standards and practices implemented in the .NET Framework. (dofactory, 2015) Finally, documentation comments were created for the core library used in the application. Again, the commenting format which was followed is that utilised in the .NET Framework.

7.8.4 PBI – Refactor Octave Application

Similar to C#, a full review of the Octave application was completed. Again, refactoring to coding standards was required. In this case the author decided to follow C# conventions for the majority of the code base – this was due to Octave not providing any documented coding standards.

Where possible, the standards utilised in pre-defined library functions provided by Octave were followed. For example, function names start with a lowercase letter – this was a convention which was observed during the implementation of the Octave application.

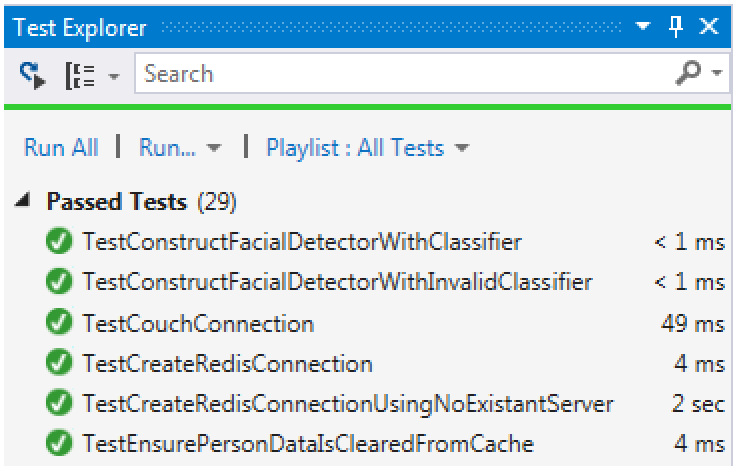
It was a goal to utilise meaningful variable names. In one instance, a structure named **recogniserData** was utilised. This contained the data required for the recogniser to function. It was initially called **sessionData** – a name which was not descriptive and particularly misleading.

Finally, all functions which were developed were updated to include documentation comments. The commenting format applied was standardised for use in MATLAB applications and was that defined by Denis Gilbert. (Gilbert, 2004)

7.9 Implemented Tests

Tests were implemented for both the C# and Octave applications. The tests which were implemented were based upon the projects Test Plan which was described prior to the implementation phase and can be viewed in Section 6.5.

In the case of C#, the tests were implemented using the Microsoft Unit Test Framework. Tests classes and methods are defined using attributes, for instance *[TestMethod]*, and support is provided for both test initialisation and clean-up methods. Test results can be verified using a variety of methods available in the *Assert* class. The tests, in this case, were run from directly within the Visual Studio 2013 IDE. They could also have been run using *Microsoft Test Manage*r, if so desired. (Microsoft, 2015)



```
Passed Tests (29)
  TestConstructFacialDetectorWithClassifier                < 1 ms
  TestConstructFacialDetectorWithInvalidClassifier         < 1 ms
  TestCouchConnection                                       49 ms
  TestCreateRedisConnection                                  4 ms
  TestCreateRedisConnectionUsingNoExistantServer             2 sec
  TestEnsurePersonDataIsClearedFromCache                     4 ms
```

Figure 53 Test Explorer in Microsoft Visual Studio 2013

In the case of Octave there is no pre-defined framework for writing or executing unit tests. There is, however, documentation that describes the format required for unit tests. In an Octave application, test blocks begin with the keyword *test*. Python style indentation is used – any code following the *test* keyword which is part of the same test block must be indented by at least one whitespace. Along with indentation, the characters **$!** must be placed at the beginning of each line of a test block. Similar to C#, *assert* functions are used to verify test results and *fail* conditions may also be specified. (Octave, 2015) Test scripts were implemented for each specific area of the application – for example a single script was developed for testing all eigenface functions. A single test execution script was defined that automatically ran all the test scripts and displayed their results.

Table 15 below shows a list of all tests which were implemented for both the C# and Octave applications.

No.	Test Name	Class Name
	Visual Studio Application	
1	TestFacialDetection	FacialDetector
2	TestConstructFacialDetectorWithClassifier	
3	TestConstructFacialDetectorWithInvalidClassifier	
4	TestMarshalFacialImage	FacialRecogniser
5	TestSendDataToCache	
6	TestSetRecogniserInterface	
7	TestNormaliseAnImageUsingTemplateMethod	FacialImageNormaliser
8	TestResizeAnImage	
9	TestSetColormapOfImage	
10	TestCouchConnection	CouchDatabase
11	TestRetrieveAll	
12	TestRetrieveAllWithAttachments	
13	TestRetrieveAPerson	
14	TestRetrieveNonExistingPersonCausesException	
15	TestRetrievePersonWithAttachment	
16	TestStoreAPerson	
17	TestStorePersonWithAttachment	
18	TestUpdateAPerson	
19	TestUpdateNonExisitingPersonCreatesANewDocument	
20	TestUpdatePersonWithAttachment	
21	TestCreateRedisConnection	RedisConnection
22	TestCreateRedisConnectionUsingNonExistantServer	
23	TestEnsurePersonDataIsClearedFromCache	
24	TestIsRecogniserAvailableSucceeds	
25	TestIsRecogniserAvailableThrowsException	
26	TestReceiveResponseUsingRedisConnection	
27	TestRedisConnectionReceiveResponseTimeout	
28	TestSendPersonDataToCache	
29	TestSendRequestUsingRedisConnection	

	Octave Application	
30	TestCalculateAverageFace	Eigenfaces
31	TestCalculateEigenfaces	
32	TestCalculateWeights	
33	TestClassifyAFace	
34	TestReduceFacesUsingAverageFace	
35	TestRedisConnectionFails	Redis
36	TestRedisConnectionSuccessful	
37	TestRedisGet	
38	TestRedisSet	
39	TestLoadMatrixDataFailsForNonExistingFile	IO
40	TestLoadYaleTrainingDatabase	
41	TestNormalise	Util

Table 15 Implemented Unit Tests

Many of the C# methods which were tested were declared with a scope of *private*. Reflection was used to both access and test these methods. A test which utilised reflection is shown in *Snippet 28*.

```
Source File: RedisConnection_Test.cs
[TestMethod]
public void TestIsRecogniserAvailableSucceeds()
{
    this.RedisDatabase.StringSet(RecogniserStatusKey,
                  (int)RecogniserStatus.Available);

    var isRecogniserAvailableMethod =
                  this.RedisConnection.GetType().GetMethod(
                           "IsRecogniserAvailable",
                            System.Reflection.BindingFlags.NonPublic |
                            System.Reflection.BindingFlags.Instance);

    var result =
                 (bool)isRecogniserAvailableMethod.Invoke(
                            this.RedisConnection,
                            new Object[] { });

    Assert.IsTrue(result);
}
```

Snippet 28 Testing Redis status key using reflection

As described at the beginning of this section, Octave tests are written in blocks. *Snippet 29* below shows an example of testing Redis connectivity from Octave and asserting that the correct result is received upon execution of a *ping* command. Note the use of indentation and the **%!** characters.

```
Source File: testRedis.m
% ___Test Redis Connection Successful___

%!test
%! host = "127.0.0.1";
%! port = 6379;
%! conn = redisConnection(host, port);
%!
%! pong = redisPing(conn);
%!
%! % Expected ping result is a 'pong' in Redis protocol format
%! expectedResult = "+PONG\r\n";
%! assert(pong, expectedResult);
```

Snippet 29 Testing Redis connectivity in Octave

7.10 Recogniser Training Methodology

During the implementation phase of this project many manual tests of the recogniser were conducted with varying levels of successful facial recognition results. It was decided to define a process for image capture in order to achieve the best possible recognition results. The recogniser which was implemented is based upon the eigenface algorithm and thus is photometric – recall that it functions by directly analysing the pixel data in the given images, as discussed throughout Chapter 2 and in Section 7.3.

The results obtained from the recogniser were influenced most noticeably by variations in both **lighting conditions** and **pose**.

Therefore, the recogniser was trained as follows. Training images for each subject were captured under a variety of lighting conditions. For each separate lighting condition it was decided to capture **nine** images of each subject. Each of these nine images were of the subject in a different facial pose, as demonstrated in *Table 16*. This method accounted for variations in pose and provided the recogniser with a diversity of training data.

Facial Poses Used During Training

Looking Up	To the Left	Directly Ahead	To the Right
In-line with Camera	To the Left	Directly Ahead	To the Right
Looking Down	To the Left	Directly Ahead	To the Right

Table 16 Facial poses used when training the recogniser

Figure 54 on the following page demonstrates a set of images which were captured for a single subject. The set of images shown represent a single lighting conditions – that which the author considered to be optimal, and displays each of the nine poses that were captured. Note that the facial poses which were captured correlate with those defined in *Table 16* above.

In terms of architecture, the capture methodology for each facial image was:

1. Capture source colour image from the Kinect sensor,
2. Detect the area containing the face,
3. Extract the facial image,
4. Store the facial image in CouchDB.

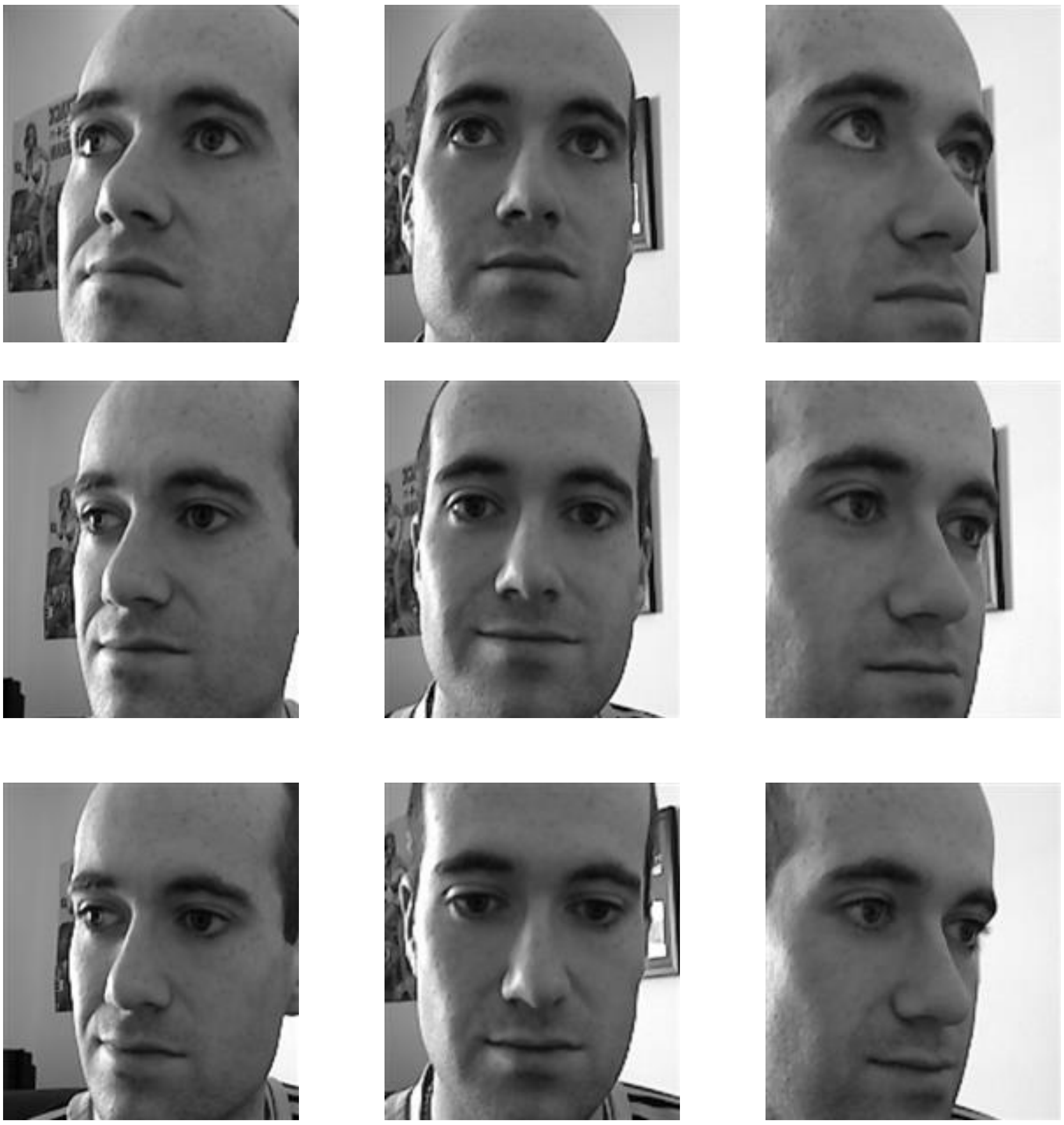


Figure 54 Recogniser training images for one lighting condition showing all nine poses

Chapter 8: Experimental Results

8.1 Experimental Methodology

This section will outline the results of a number of experiments which were conducted to analyse the accuracy of the facial recognition system which was implemented. Each experiment was conducted under different lighting conditions and used varying sets of data that were captured for the participating subjects. For each experiment, the following constants were maintained:

- Two subjects were analysed,
- Training images were captured as described in Section 7.10,
- Thirty tests were conducted for each subject,
- Training and test images were captured within a distance of one metre from the camera.

The experiments which follow investigated the abilities of the implemented recogniser under constant lighting conditions with a small but suitable training set, under poor lighting conditions with an unsuitable training set and under constant lighting conditions with a training set which was both sizeable and suitable.

8.2 Recognition Results under Optimal Lighting Conditions

Optimal lighting conditions were defined as: afternoon daylight, with all training and test imagery captured within an indoor environment. There was no glare. All training and test images were captured with an internal wall as the background and with the predominant light source coming from behind and to the left.

This experiment used a total of eighteen training images, nine for each subject.

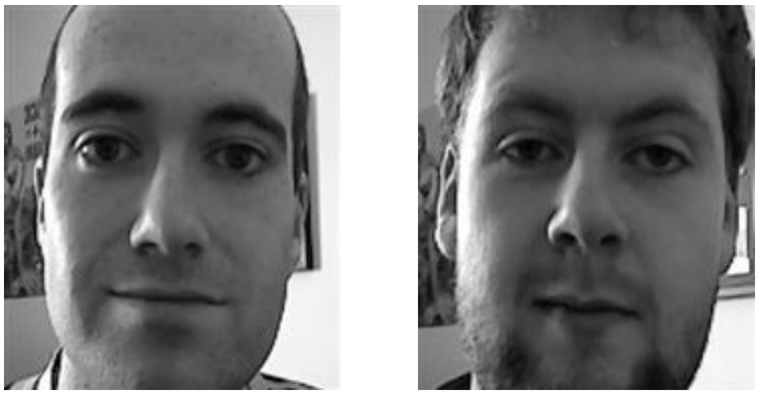


Figure 55 Sample of the training images captured under optimal lighting conditions

The results determined for each subject varied: successful recognition rates were between sixty and eighty percent.

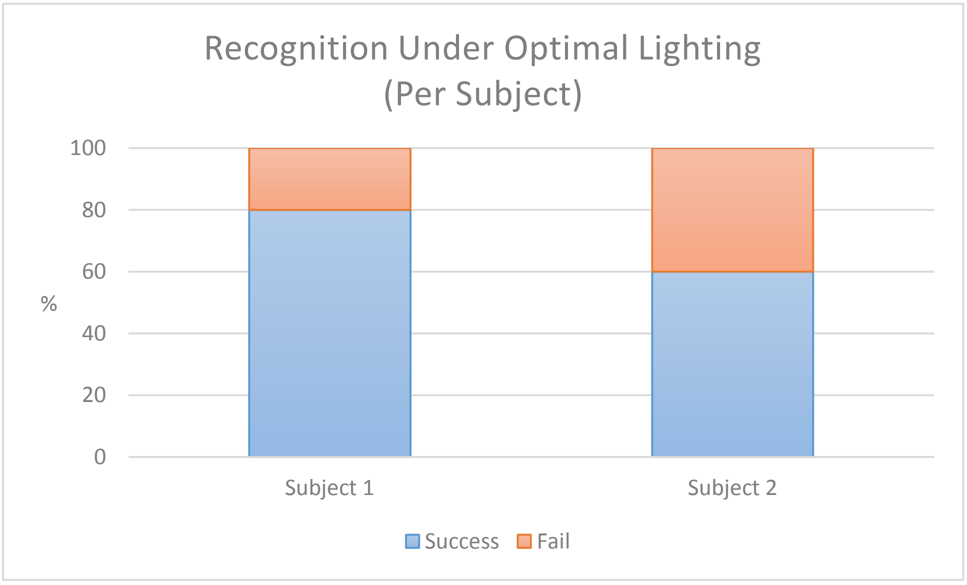


Figure 56 Recognition results per subject under optimal lighting

An overall successful recognition rate of seventy percent was observed. These results show that the recogniser is effective under optimal lighting conditions. Particularly, successful results are evident when both the training and test imagery are captured under similar conditions.

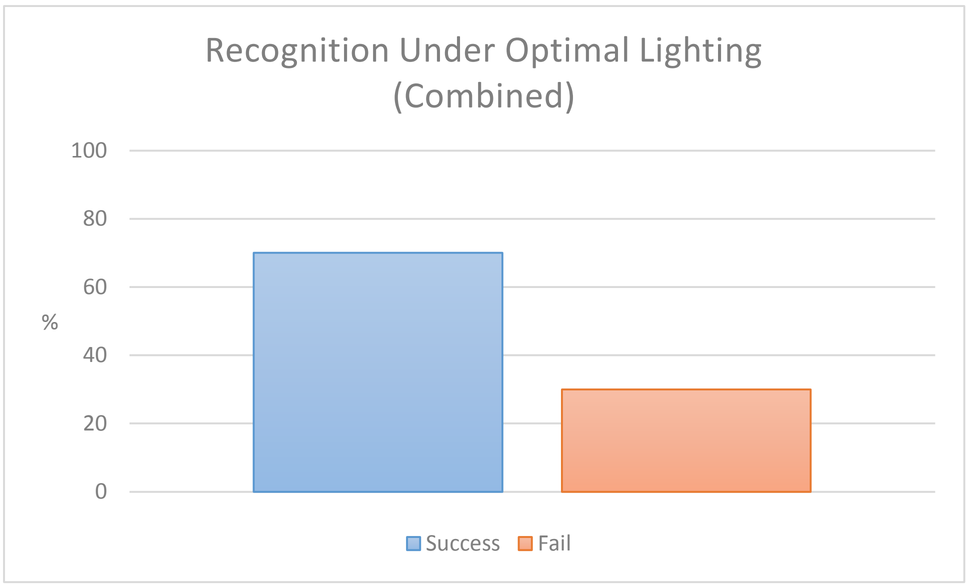


Figure 57 Combined recognition results under optimal lighting

8.3 Recognition Results under Poor Lighting Conditions

Poor lighting conditions were defined as fading daylight. Again, all training and test imagery was captured within an indoor environment. For this experiment, the training images which were previously captured under optimal lighting conditions were used. Therefore, this experiment also used a total of eighteen images for training, nine per subject. All test images were captured under poor lighting conditions.

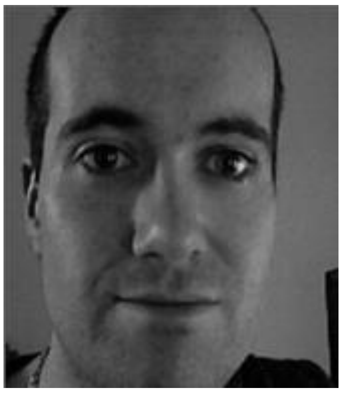


Figure 58 Test image captured in poor lighting

It is important to note the difference in intensity levels between the test image shown in *Figure 58* above and the training images which were utilised as shown in *Figure 55* under Section 8.2.

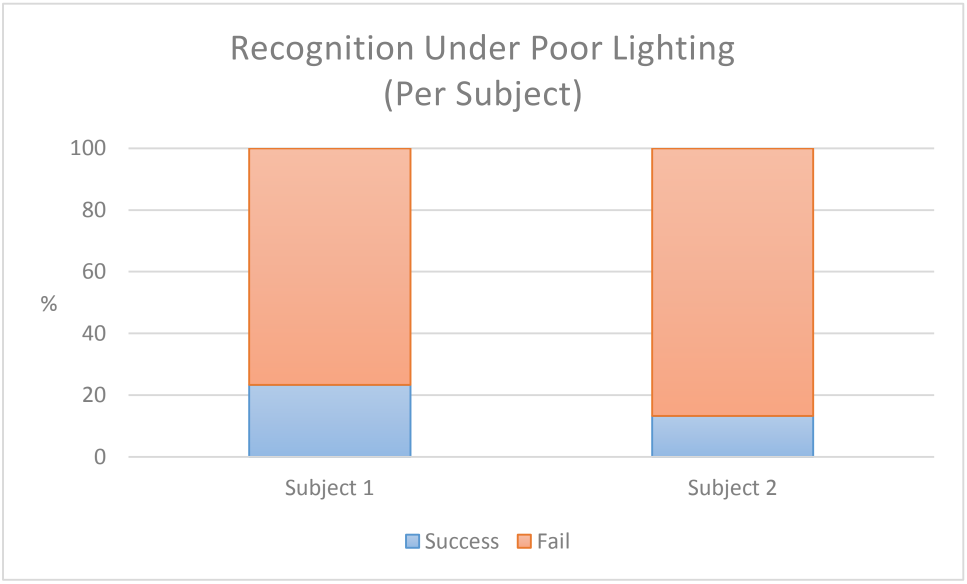


Figure 59 Recognition results per subject under poor lighting

It is clear from the results shown in *Figure 59* that lighting has a significant impact on recognition results. Testing the recogniser under poor lighting conditions dropped the successful recognition rates observed in the previous experiment down to between thirteen and twenty-three percent. The overall successful recognition rates for this experiment were just above eighteen percent.

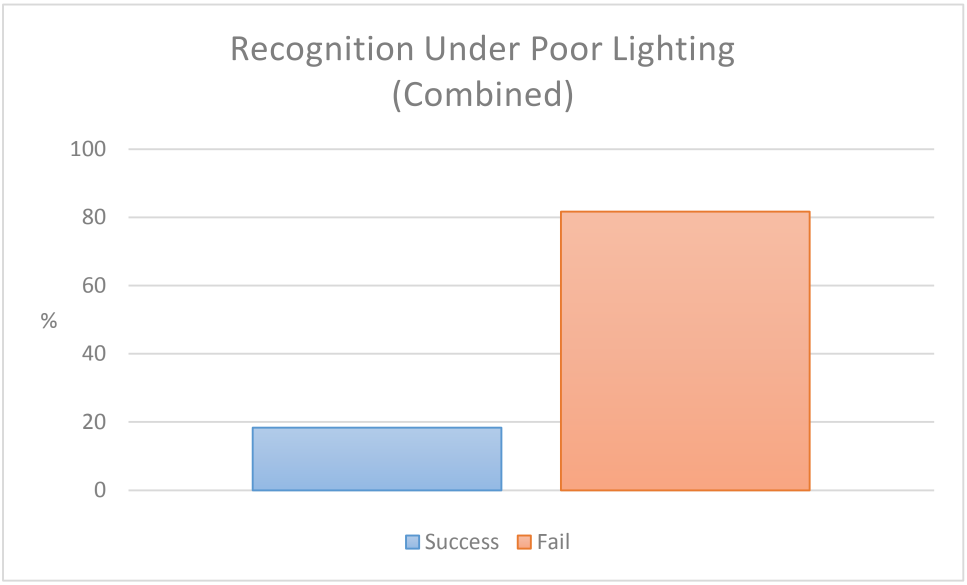


Figure 60 Combined recognition results under poor lighting

8.4 Recognition Results under Artificial Lighting Conditions

In this case, artificial lighting meant that imagery was captured in an indoor environment under standard domestic lighting. For this experiment a larger set of training imagery was employed. The nine images of each subject as captured under optimal lighting in Section 8.2 were retained and nine more images of each subject were captured under artificial lighting. Therefore, a total of thirty six training images were used here, eighteen per subject.

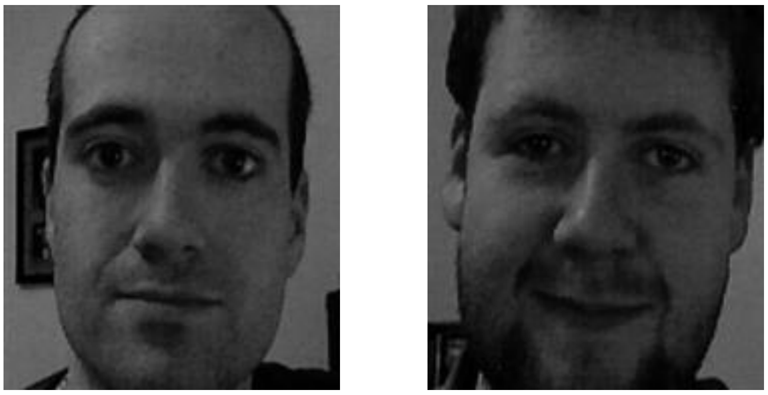


Figure 61 Sample of the training images captured under artificial lighting

Two artificial light sources were used to illuminate the interior room where the training and test images were captured. The first was above and slightly to the left of the camera and the second was behind the camera at a distance of roughly two metres. Two light sources were used to ensure an even lighting distribution across each facial image and to prevent shadows.

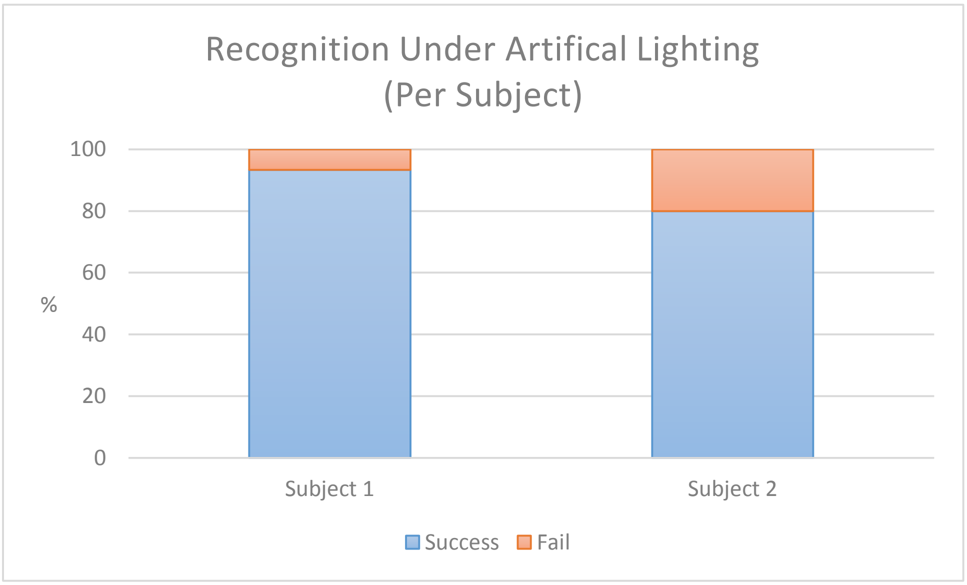


Figure 62 Recognition results per subject under artificial lighting

This experiment used the largest number of training images per subject and its tests were performed under lighting conditions for which corresponding training images had been captured. The results, as shown in *Figure 63*, were notable. Successful recognition rates for the individual subjects ranged from eighty to ninety three percent.

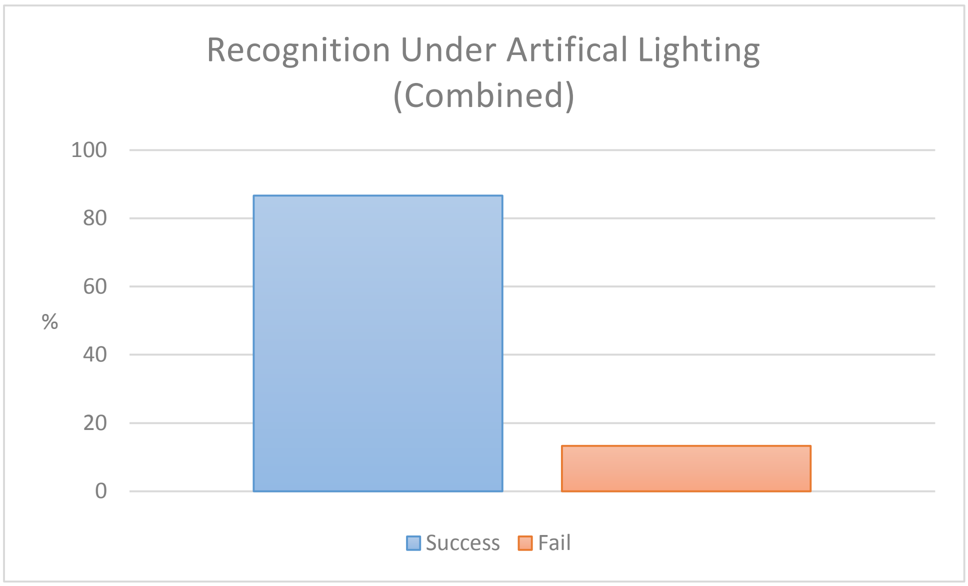


Figure 63 Combined recognition results under artificial lighting

Combined, this experiment achieved recognition results of almost eighty seven percent. These results again demonstrate that the recogniser is effective under constant lighting conditions. The results also show that recognition rates are positively affected by capturing a larger set of training data for each individual.

Chapter 9: Analysis, Conclusions and Recommendations

9.1 Analysis of Experimental Results

The experimental results of this project, as detailed in Chapter 8, show the following:

- The eigenfaces algorithm can produce successful facial recognition results of seventy percent and above when both training and testing imagery are captured under consistent lighting conditions for a variety of facial poses,
- The eigenfaces algorithm is very susceptible to changes in lighting conditions. Successful recognition results drop to below twenty-five percent when the training and testing imagery are captured under different lighting conditions,
- Using a large set of training images, captured under multiple lighting conditions and for a variety of poses, successful recognition results of eighty percent and above are produced.

These results are consistent with, and quantify, the research findings on both the photometric eigenfaces algorithm and the challenges to facial recognition which were discussed in Chapter 2.

One of the most significant findings of this projects research suggested that lighting conditions have a noteworthy impact on the probability of achieving successful facial recognition when utilising a photometric algorithm. This was confirmed by the experimental results. It was found that the impact of lighting variations upon facial recognition results could be partially mitigated by capturing training imagery under many lighting conditions. As stated, recognition results above seventy percent were achieved. However, even under consistent lighting it was not possible to achieve one hundred percent successful recognition results.

A second concern of photometric algorithms encountered during research was the effect of facial pose. This was mitigated by capturing training imagery of each subject under many facial poses, as defined in Section 7.10. As a result, variations of facial pose within the testing imagery were found to not have an impact upon successful recognition results.

Finally, the background conditions of captured imagery were predicted to have an effect on recognition results. This was prevented, through depth reduction (Section 7.2.5) and facial detection (Section 7.3), and thus not found to effect the efficiency of the facial recogniser.

9.2 Conclusions

This project was conducted to answer the research question, *could facial recognition software be a viable identification and security measure at point-of-sale (POS) in payments processing systems?*

A software system was implemented to answer this question. A photometric facial recogniser, based upon the eigenfaces algorithm, was at the core of the developed architecture. The implemented recogniser was tested to determine the rate of successful recognition that could be achieved. This section will discuss the results of this project in terms of both facial recognition accuracy and efficiency.

Following the completion of the implementation and from the analysis of the experimental results, it was found that the photometric eigenfaces algorithm can produce viable identification results under ideal environmental conditions. However, the results found indicate that this algorithm, and photometric algorithms as a whole, would not be a sustainable identification and security measure within point-of-sale payment systems.

Successful recognition results are unstable and vary – particularly due to variations in external factors. Success rates of seventy percent were observed using a small training set which accounted for variations in pose. The success rates of the recogniser improved as larger sets of training imagery were captured – results above ninety percent were observed. However, changes to external factors, primarily lighting conditions, had a notable impact on the performance of the recogniser – results as low as twenty percent were observed. These results concur with prior research on photometric algorithms that encountered success rates of up to ninety percent, but were also affected by external factors. This research was discussed in Chapter 2. The observed results are outside the boundaries of those which would be acceptable in any payments processing system.

In current point-of-sale payments systems, verification is principally conducted when the end user enters a personal identification number, or PIN. This method is one hundred percent successful assuming that the correct PIN number has been entered. However, it is also time consuming process. In a real world implementation, the facial recognition system which was implemented would capture user facial images automatically and be capable of processing them in an efficient manner. Techniques such as background removal and facial detection were effective and ensured that a facial image of the correct individual was captured. User verification was observed to take less than one second when executed on the primary development machine. This process included normalising a facial image, passing it to the recogniser (implemented in Octave) using the Redis cache, awaiting a response and displaying the results.

Performance issues were encountered during development when launching the recogniser. Upon start-up, the recogniser was retrained and this process was observed to take twenty minutes to complete. A persistence model was implemented which stored the recognisers required data on disk in HDF5 format. This data was reloaded on start-up and observed to take ten seconds. This represented a notable improvement in performance. It is important to note that when new facial images were added to the recogniser it had to be completely retrained, which as mentioned above, took twenty minutes to complete.

Architectural decisions had an impact on this project. Firstly the use of C# to develop both the user interface and core functionality which allowed the architecture to function was found to be a suitable design choice. It had excellent API support for the Kinect sensor, the Redis cache and CouchDB. Previous experience with the language meant C# provided an instinctual development environment.

The choice of GNU Octave to implement the recogniser produced some issues. Notably, it does not have wide support for external packages, including databases. This necessitated the use of Redis as an intermediary to pass image data from the both the C# application and the facial database to the Octave application. Though one of Redis' primary benefits is its speed, performance of the recogniser would have been improved if it were able to directly communicate with the database.

Implementing the facial database using CouchDB was advantageous. CouchDB was observed to be very efficient when executing queries and, as discussed in Section 7.5, the application of its attachments feature to store facial images was very suitable for this project. The use of JSON formatting by the database when returning query results, and the prominent availability of CouchDB drivers, meant it was a valuable database provider to interact with.

The usage of the Kinect sensor was successful. Both the colour and depth data streams were observed to produce images of an acceptable quality. The depth reductions techniques developed were effective – they allowed for unwanted background noise, particularly unwanted facial images, to be accurately removed from source imagery.

Briefly, it was found that the photometric eigenfaces algorithm is efficient and produces successful results. However, due to its unfavourable recognition accuracy, it is not recommended as a viable verification and security method in point-of-sale payment processing systems.

9.3 Recommendations

Despite the conclusion that photometric facial recognition algorithms are not viable in payments systems, there are a number of recommendations which must be highlighted based upon the outcome of this project. Biometric verification is a growing field. In the payments industry alone both Apple and MasterCard are making advancements in this area, as discussed in Section 4.5.

It is suggested that further research is undertaken in the biometric field of facial recognition with regards to its viability within payments systems. Three dimensional geometric techniques should be considered. These methodologies, some of which were briefly discussed in Chapter 2, are independent of factors such as lighting and pose which were problematic to the eigenfaces algorithm.

The investigation of geometric facial recognition methodologies could be combined with the utilisation of machine learning techniques such as artificial neural networks. This, along with the decreasing costs of big data processing could allow a facial recognition system to be developed and trained using a dataset of considerable size. It is probable that such a system could produce highly successful and increasingly accurate recognition results.

The R programming language is of particular interest. It is a functional language which has widespread use in mathematics and statistics. R is extensively supported, much more so than Octave was. A considerable number of plugins have been developed for R which provide functionality such as access to a number of databases (both SQL and NoSQL) and the implementation of many types of artificial neural networks.

In terms of imaging hardware, the Kinect version 1 sensor which was utilised produced reliable results. It was used effectively for both image capture and depth reduction. Utilising a Kinect version 2 in future work is recommended to ensure continued successful results are achieved – the application of its improved depth sensor could be applied to aid the capture 3D images.

Works Cited

Axis Communications, 2014. *History.* [Online]

Available at: http://www.axis.com/corporate/about/history.htm

[Accessed 30 November 2014].

Belheumer, P. N., Hespanha, J. P. & Kriegman, D. J., 1997. Eigenfaces vs Fisherfaces: Recogniton Using Class Specific Linear Projection. *IEE Transactions On Pattern Analysis And Machine Intelligence,* 19(7), pp. 711-720.

Belhumeur, P. N., 2006. *Ongoing Challenges in Face Recognition,* New York: Columbia University.

Belicove, M. E., 2013. *The Benefits of Cloud-Based Point-of-Sale Systems.* [Online]

Available at: http://www.entrepreneur.com/article/224961

[Accessed 08 November 2014].

Bergholz, M., 2015. *go-redis.* [Online]

Available at: https://github.com/markuman/go-redis

[Accessed 3 March 2015].

Braintree, 2014. *Braintree Payments.* [Online]

Available at: https://www.braintreepayments.com/

[Accessed 08 November 2014].

Brignall, M., 2014. *Mastercard launches first thumbprint biometric card.* [Online]

Available at: http://www.theguardian.com/money/2014/oct/17/mastercard-thumbprint-biometric-card

[Accessed 30 November 2014].

Cook, J. D., 2009. *Three Algorithms for Converting Color to Grayscale.* [Online]

Available at: http://www.johndcook.com/blog/2009/08/24/algorithms-convert-color-grayscale/

[Accessed 7 April 2015].

Devaney, E., 2014. *The History of the Webcam.* [Online]

Available at: http://www.ehow.com/info_8626014_history-webcam.html

[Accessed 30 November 2014].

dofactory, 2015. *C# Coding Standards and Naming Conventions.* [Online]
Available at: http://www.dofactory.com/reference/csharp-coding-standards
[Accessed 19 April 2015].

Entrepeneur.com, 2014. *Point of Sale (POS) Systems.* [Online]
Available at: http://www.entrepreneur.com/encyclopedia/point-of-sale-pos-system
[Accessed 08 November 2014].

FaceDetection, 2015. *Face Detection Techniques.* [Online]
Available at: http://facedetection.com/techniques/
[Accessed 25 April 2015].

Georghiades, A., Belhumeur, P. & Kriegman, D., 2001. From Few to Many: Illumination Cone Models for Face Recognition under Variable Lighting and Pose. *IEEE Transactions On Pattern Analysis And Machine Intelligence,* 23(6), pp. 643-660.

Gilbert, D., 2004. *M-file Header Template.* [Online]
Available at: http://www.mathworks.com/matlabcentral/fileexchange/4908-m-file-header-template
[Accessed 19 April 2015].

GitHub, 2013. *DreamSeat.* [Online]
Available at: https://github.com/vdaron/DreamSeat/commits/master
[Accessed 7 March 2015].

GitHub, 2013. *DreamSeat.* [Online]
Available at: https://github.com/vdaron/DreamSeat
[Accessed 8 March 2015].

Halleck, T., 2014. *PayPal Accounts Hacked With A Click: Engineer Uncovers Potential Security Breach.* [Online]
Available at: http://www.ibtimes.com/paypal-accounts-hacked-click-engineer-uncovers-potential-security-breach-1735158
[Accessed 9 April 2015].

Ha, P., 2014. *The Best Wireless IP Camera.* [Online]
Available at: http://thewirecutter.com/reviews/best-wireless-ip-camera/#
[Accessed 30 November 2014].

Haughey, D., 2014. *MoSCoW Method.* [Online]
Available at: http://cdn.projectsmart.co.uk/pdf/moscow-method.pdf

Henry, A., 2012. *Five Best Webcams.* [Online]
Available at: http://lifehacker.com/5961369/five-best-webcams
[Accessed 30 November 2014].

H. R. S., 2014. *MFlow.* [Online]
Available at: https://www.hrsid.com/product-mflow
[Accessed 02 November 2014].

Hyung-Ji Lee, W.-S. L. J.-H. C., 2001. *Face Recognition Using Fisherface Algorithm And Elastic Graph Matching.* Thessaloniki, IEEE.

Jana, A., 2012. *Kinect for Windows SDK Programming Guide.* Birmingham: Packt Publishing.

Jeffries, A., 2014. *Apple Pay allows you to pay at the counter with your iPhone 6.* [Online]
Available at: http://www.theverge.com/2014/9/9/6084211/apple-pay-iphone-6-nfc-mobile-payment
[Accessed 30 Nobember 2014].

Keen, P., 2014. *The Life of a Stripe Charge.* [Online]
Available at: https://www.petekeen.net/life-of-a-stripe-charge
[Accessed 08 November 2014].

Kennedy, J., 2015. *Fraud comes to Apple Pay as criminals exploit weakness in verification process.* [Online]
Available at: http://www.siliconrepublic.com/enterprise/item/40993-fraud-comes-to-apple-pay-as
[Accessed 9 April 2015].

KeyLemon, 2014. *KeyLemon.* [Online]
Available at: https://www.keylemon.com/
[Accessed 02 November 2014].

Krebson Security, 2015. *How Was Your Credit Card Stolen?.* [Online]
Available at: http://krebsonsecurity.com/2015/01/how-was-your-credit-card-stolen/
[Accessed 9 April 2015].

Lally, C., 2015. *Gardaí introduce facial recognition to identify suspects.* [Online]
Available at: http://www.irishtimes.com/news/crime-and-law/garda%C3%AD-introduce-facial-recognition-to-identify-suspects-1.2094072
[Accessed 8 March 2015].

Lee, K., Ho, J. & Kriegman, D., 2005. Acquiring Linear Subspaces for Face Recognition under Variable Lighting. *IEEE Transactions On Pattern Analysis And Machine Intelligence,* 27(5), pp. 684-698.

Lienhart, R., Kuranov, A. & Pisarevsky, V., 2002. *Empirical Analysis of Detection Cascades of Boosted Classifiers for Rapid Object Detection,* Santa Clara: Intel Corporation.

Lin, S.-H., 2000. An Introduction to Face Recognition Technology. *Informaing Science - Special Issue on Multimedia Informing Technologies,* 3(1), pp. 1-7.

MasterCard, 2015. *Chip & Pin FAQ.* [Online]
Available at: http://www.mastercard.com/ie/consumer/chip-and-pin-faq.html
[Accessed 9 April 2015].

Microsoft, 2014. *Color Stream.* [Online]
Available at: http://msdn.microsoft.com/en-us/library/jj131027.aspx
[Accessed 15 November 2014].

Microsoft, 2014. *Depth Stream.* [Online]
Available at: http://msdn.microsoft.com/en-us/library/jj131028.aspx
[Accessed 15 November 2014].

Microsoft, 2014. *Face Tracking.* [Online]
Available at: http://msdn.microsoft.com/en-us/library/jj130970.aspx
[Accessed 15 November 2014].

Microsoft, 2014. *Kinect for Windows.* [Online]
Available at: http://www.microsoft.com/en-us/kinectforwindows/develop/default.aspx
[Accessed 22 November 2014].

Microsoft, 2014. *Kinect for Windows features.* [Online]
Available at: http://www.microsoft.com/en-us/kinectforwindows/meetkinect/features.aspx
[Accessed 22 November 2014].

Microsoft, 2015. *Writing Unit Tests for the .NET Framework with the Microsoft Unit Test Framework for Managed Code.* [Online]
Available at: https://msdn.microsoft.com/en-us/library/hh598960.aspx
[Accessed 25 April 2015].

Octave, 2015. *Test Functions.* [Online]
Available at: https://www.gnu.org/software/octave/doc/interpreter/Test-Functions.html
[Accessed 28 April 2015].

O'Rourke, N. & Hatcher, L., 2013. Principal Component Analysis. In: 2nd, ed. *A Step-by-Step Approach to Using SAS for Factor Analysis and Structural Equation Modeling.* Cary: SAS Institute Inc., pp. 2-56.

PayPal, 2014. *History.* [Online]
Available at: https://www.paypal-media.com/about
[Accessed 08 November 2014].

PayPal, 2014. *PayPal Developer Documentation.* [Online]
Available at: https://developer.paypal.com/docs/
[Accessed 08 November 2014].

Ramchandra, A. & Kumar, R., 2013. Overview of Face Recognition System Challenges. *Internation Journal of Scientific & Technology Research,* 2(8), pp. 234-236.

Realex Payments, 2013. *PayPal Remote API.* [Online]
Available at: https://resourcecentre.realexpayments.com/documents/pdf.html?id=171
[Accessed 08 November 2014].

Redis, 2015. *Redis Protocol Specification.* [Online]
Available at: http://redis.io/topics/protocol
[Accessed 20 March 2015].

Redmond, E. & Wilson, J. R., 2012. CouchDB. In: J. Carter, ed. *Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement.* s.l.:Pragmatic Bookshelf, pp. 177-217.

Redmond, E. & Wilson, J. R., 2012. Redis. In: J. Carter, ed. *Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement.* s.l.:Pragmatic Bookshelf, pp. 261-304.

Scholarpedia, 2014. *Scholarpedia.* [Online]
Available at: http://images.scholarpedia.org/w/images/b/b0/Fisherfaces.jpg
[Accessed 04 December 2014].

ServiceStack, 2015. *ServiceStack.Redis.* [Online]
Available at: https://servicestack.net/download#free-quotas
[Accessed 2 April 2015].

Sims, K., 2014. *Stripe Press Resources.* [Online]
Available at: https://stripe.com/about/press
[Accessed 08 November 2014].

Sirovich, L. & Kirby, M., 1987. Low-dimensional procedure for the characterization of human faces. *Journal of the Optical Society of America A,* 4(3), pp. 519-524.

Smith, L. I., 2002. *A tutorial on Principal Component Analysis,* s.l.: s.n.

StackExchange, 2013. *Euclidean Distance in Octave.* [Online]
Available at: http://scicomp.stackexchange.com/questions/8223/euclidean-distance-in-octave
[Accessed 18 February 2015].

StackExchange, 2015. *StackExchange.Redis.* [Online]
Available at: https://github.com/StackExchange/StackExchange.Redis
[Accessed 4 April 2015].

StackOverflow, 2013. *Recommended values for OpenCV detectMultiScale() parameters.* [Online]
Available at: Recommended values for OpenCV detectMultiScale() parameters
[Accessed 15 April 2015].

StatisticBrain, 2014. *Credit Card Fraud Statistics.* [Online]
Available at: http://www.statisticbrain.com/credit-card-fraud-statistics/
[Accessed 9 April 2015].

Stripe, 2014. *Stripe Features.* [Online]
Available at: https://stripe.com/ie/features
[Accessed 08 November 2014].

Taigman, Y., Yang, M., Ranzato, M. & Wolf, L., 2014. *DeepFace: Closing the Gap to Human-Level Performance in Face Verification,* Menlo Park: Facebook.

Tech Radar, 2010. *How face detection works.* [Online]
Available at: http://www.techradar.com/news/software/applications/how-face-detection-works-703173
[Accessed 31 March 2015].

TechCrunch, 2014. *TechCrunch.* [Online]
Available at: http://tctechcrunch2011.files.wordpress.com/2014/06/screen-shot-2014-06-05-at-11-42-14-am.png?w=738
[Accessed 22 November 2014].

Thompson, C., 2014. *How hackers could still get around Apple Pay security.* [Online]
Available at: http://www.cnbc.com/id/101992749
[Accessed 30 November 2014].

Turk, M. & Pentland, A., 1991. Eigenfaces for Recognition. *Journal of Cognitive Neuroscience,* 3(1), pp. 71-86.

Viola, P. & Jones, M., 2001. *Rapid object detection using a boosted cascade of simple features.* Cambridge, IEEE.

Visa, 2015. *Verified by Visa.* [Online]
Available at: http://www.visaeurope.com/making-payments/verified-by-visa/
[Accessed 9 April 2015].

Wagner, P., 2011. *Eigenfaces.* [Online]
Available at: http://www.bytefish.de/blog/eigenfaces/
[Accessed 18 February 2015].

Wagner, P., 2012. *Fisherfaces.* [Online]
Available at: http://www.bytefish.de/blog/fisherfaces/
[Accessed 1 April 2015].

Weckler, A., 2014. *Twitter unveils new Stripe feature created by successful Collison brothers.* [Online]
Available at: http://www.independent.ie/business/technology/news/twitter-unveils-new-stripe-feature-created-by-successful-collison-brothers-30570393.html
[Accessed 08 November 2014].

Wertheim, D., 2014. *The release of v1.0.0 of MyCouch.* [Online]
Available at: http://danielwertheim.se/2014/04/16/the-release-of-v1-0-0-of-mycouch/
[Accessed 8 March 2015].

Wicks, M., 2014. *8 Things to Consider Before Implementing a Cloud Based POS System.* [Online]
Available at: http://www.businessbee.com/resources/profitability/8-things-to-consider-before-implementing-a-cloud-based-pos-system/
[Accessed 08 November 2014].

Wilson, P. I. & Fernandez, D. J., 2006. Facial Feature Detection Using Haar Classifiers. *Journal of Computer Sciences in College,* 21(4), pp. 127-133.

Wiskott, L., Fellous, J.-M., Krüger, N. & von der Malsburg, C., 1997. Face Recognition by Elastic Bunch Graph Matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 19(7), pp. 775-779.

Wiskott, L., Fellous, J.-M., Krüger, N. & von der Malsburg, C., 1999. Face Recognition by Elastic Bunch Graph Matching. In: 1st, ed. *Intelligent Biometric Techniques in Fingerprint and Face Recognition.* s.l.:CRC Press, pp. 355-396.

Woodman, R., 2007. *A Photometric Stereo Approach to Face Recognition,* Bristol: University of the West of England.

Zhang, Z., 2012. Microsoft Kinect Sensor and Its Effect. *IEEE Multimedia,* pp. 4-10.