**Image Processing & Facial Recognition**

A dissertation submitted in partial fulfilment of

the requirements for the degree of

BACHELOR OF *SCIENCE* in Computer Science

in

The Queen's University of Belfast

by

**'Ben Anderson’**

**'Submitted on 27th April 2018'**

Acknowledgements

I would like to give acknowledgement to Professor Ji Ming who has been my project advisor throughout the planning and development of this project.

Abstract

This project is an investigation into image processing and the different image effects that can be simulated using such pixel manipulation calculations. These manipulations are used in later experiments to simulate different environmental factors that may been seen in real life image capture scenarios. For example, an image of a person that is over-exposed, out of focus, or has been captured in a dark environment resulting in an increase of noise.

Following on, an implementation of patch-based facial recognition using the LFW crop dataset is used to experiment with how these environmental factors may have a negative or positive effect on the accuracy of facial recognition. In summary I have found certain environmental factors that have severe negative effects on facial recognition accuracy, and changes to how the recognition algorithm itself to increase the overall accuracy.

All the developed image processing and facial recognition algorithms have been developed for an Android device to be used in an application, to make the features more readily usable.

**Contents**

1. Problem description4

1.1 Solution approach5

2.0 System requirements and specification7

2.1 Hardware and software requirements7

2.2 Project development cycle8

3.0 System design9

3.1 User flow diagram9

3.2 Wireframe diagrams10

3.3 Development design14

4.0 Implementation and testing24

4.1 Implementation24

4.2 Testing40

5.0 Experiments44

6.0 System evaluation48

7.0 Conclusion49

8.0 References50

9.0 Appendices53

Problem description

The purpose of this project is to research, implement and experiment with ways that may improve or effect the accuracy of unconstrained facial recognition [1]. Over previous years lots of work has been carried out in-order to improve unconstrained facial recognition for several reasons. The process of identifying people is vital when implementing access control and for security purposes. An example of this is a case study carried out Joshua C. Klontz and Anil K. Jain of Michigan State University after the Boston Marathon bombings [2]. In this case study a comparison between three commercial state of the art facial recognition systems to determine just how accurate each was, it was noticeable that none of the systems were perfect.

The main problem with facial recognition algorithms is that they are not always 100% accurate. In some cases, it is vital to reach the correct conclusion, for example when we are trying to identify an individual. Accuracy in these algorithms can degrade dramatically depending on a person’s pose or facial expression and can also be dependent on the image lighting or quality at the time. In a case study carried out by A.S. Georghiades, P.N. Belhumeur, and D.J. Kriegman [3], various experiments were carried out to discuss the effect of differing poses and illuminance levels in an image of a person’s face. The results clearly show that depending on a pose or level of illuminance in comparison with the image you are trying to match, the percentage rate of errors can increase dramatically.

These problems do not normally need to be addressed in algorithms which use a 1:1 matching process (one image matched to another), as these are normally used in scenarios where the algorithm is using a satisfactory recognition image [4]. For example, a smart phone using an image of the owner to allow them to unlock their phone. In this case the user will aim to capture an image to save for reference which is of good light level and is guided to fit their face into the capture zone at a good angle, decreasing the opportunity for error rates to rise.

The work of this project will experiment with a specific type of facial recognition known as block-based correlation [5]. In this approach two image patches at a time are used, one of the user’s image and one of a training image. With these two patches we can calculate the correlation between the pixels values to accumulate a score. This process is then carried out for each patch throughout the entirety of each training image in comparison with all possible patches in the main image. It is important to note that each patch investigated can overlap. In doing this, we account for a facial feature being in a slightly different location in comparison to others, which may reduce the effect of differing poses as we are segmenting the image and looking to compare features.

To further investigate the accuracy of patch-based facial recognition it is necessary to consider limiting factors such as mechanisms within the algorithm itself such as patch size, but also factors such as image noise, brightness, contrast, blur and exposure.

In-order to accomplish this it is important to make use of image processing, which will allow for the manipulation of various aspects of the image. Seeing the image as a 2D structure [6] allows us to process the image, learning information with regards to its size, pixel co-ordinate (x, y) values and the colour channel ratio. By learning all of this information from an image, we can begin to manipulate by changing colour values held in a pixel, which would have a direct effect on the image.

Overall, there are various methods of face recognition and have been various investigations into how we could improve face recognition, perhaps the problem does not lie with robustness of an algorithm, but instead external factors such as a person’s pose, or the quality of the image we are trying to recognise.

Solution approach

As a solution to the problem I shall develop an Android application to house my image processing toolkit and facial recognition algorithm. The reason I want to do this is so that the work can be experimented with easily by other people. It provides the possibility to create a user-friendly app which can be multi-purpose, either to simply edit and save images or to also complete facial recognition for experimental or practical use.

For this project I will consider the 2D image structure as a bitmap which is mainly used in android development for storing image data. This makes use of the three colour channels red, green and blue (RGB), ignoring alpha (transparency) [7]. Using this mechanism to store image data I can develop and implement algorithms which manipulate pixel values to simulate various scenarios which may or may not have an effect on the accuracy of the facial recognition process.

The android application will provide various functionality to the user allowing them to alter the image they have chosen to use. The application will enable to user to adjust image properties such as the brightness, contrast and exposure of the image. It will also provide methods of simulating scenarios which can be later used in experiments to investigate the effect of things such as blur or noise on the accuracy of face recognition. To do this I will research image processing topics to develop my own code allowing me to complete these features, rather than using external packages such as OpenCV to complete the process for me. In doing this, I will become much more familiar with how data portrays an image.

As a database of images for my research and experimentation I will use the LFW crop dataset [8]. This is a cropped version of the LFW dataset [9], which keeps only the centre portion of each image. This allows all the focus in each image to be the facial features and omit the background. In the database used, the images are greyscale, however by using image processing I can introduce various levels of light or colour in some cases to see how this effects my experiments.

As previously discussed in the problem section, the methodology behind the facial recognition I will implement is multi-scale block-based correlation [5]. To implement this, I plan to store a subset database of images from the LFW crop dataset [8] and develop a loader class which will load all the data relevant to each training image for each person. The algorithm will calculate the correlation of the pixels in two patches, one of the user’s image and one of the nth training images for a person in the database. After calculating these correlations, the application will provide a meaningful set of results to the current user, showing them how accurate the recognition has been, and a name of the best match found.

System requirements and specification

Below is a list of requirements the developed application should adhere to. The developed application should:

1. Be an application for an android device.
2. Be developed for a suitable API level so that it is compatible with most devices in today’s society.
3. Be developed in a suitable language such as C++ or Java.
4. Make use of a consistent, user-friendly interface.
5. Be user friendly to ensure the user knows what function they are carrying out when using the interface.
6. Support capturing an image for use from the user’s mobile device
7. Support selecting an image from the user’s device storage.
8. Support saving an edited image to the devices storage.
9. Make use of a main menu system to allow the user to easily navigate different options that the application provides.
10. Provide a preview window the user can make use of when making changes to an image through image processing, allowing them to see changes made.
11. Make use of complex algorithms to carry out image processing rather than making use of external packages such as OpenCV.
12. Facilitate the user to alter brightness properties of an image in use.
13. Facilitate the user to alter contrast properties of an image in use.
14. Facilitate the user to alter saturation properties of an image in use.
15. Facilitate the user to alter exposure properties of an image in use.
16. Facilitate the user in simulating noise in their image.
17. Facilitate the user in simulating blur in their image.
18. Provide functionality to allow the user to carry out basic edge detection on their image.
19. Facilitate the user in converting their image to grey scale.
20. Allow the user to carry out facial recognition on the image they have uploaded and obtain results.
21. Make use of the LFW crop dataset for training image when carrying out facial recognition.
22. Have an application name that the end user can identify.
23. Have an application logo that the end user can identify.
24. Be compatible with varying screen sizes so that the application is readily available for use for a wide range of possible users.
25. Be built with a well organised code base which is easily read. This is important in cases where the application is further developed in the future.

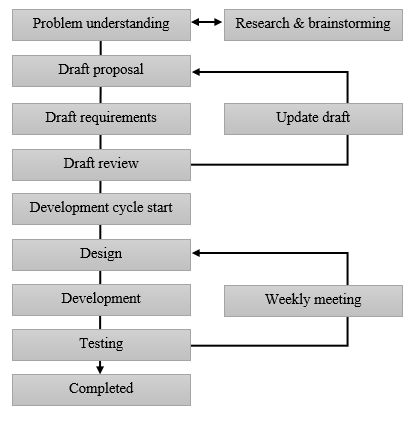
Hardware/software description

The image processing and facial recognition application will be developed using Java on Android Studio. My assigned supervisor and I discussed in a meeting the most suitable possibilities. We discussed using C++ as it is very powerful when it comes to image processing, however I have previous experience using Java and Android Studio and after some research I found this would be more suited for developing an android application.

This application will be developed on an ASUS Zen Book i5-7200U 8GB RAM running Android Studio 3.0 with JRE 1.8. For emulation I will be using a NEXUS 5X emulator and a Samsung Galaxy A5 (2017) phone.

Project development cycle

This section outlines the development cycle I will follow in developing this application.



System design

The purpose of this section is to define a plan for the application being created. In doing this I am able ensure the system satisfies the system objectives and requirements stated in section 2.0.

User flow diagram

The below user flow diagram is a plan of how the end user of this product is going to interact with the application. This diagram covers the process the user will use to complete a certain task that may be listed as a requirement in section 2.0.

**Face recognition screen**

**Start-up screen**

**Main menu screen**

**Face recognition results screen**

**Adjust image screen menu**

**Image filters screen**

**Image colour screen**

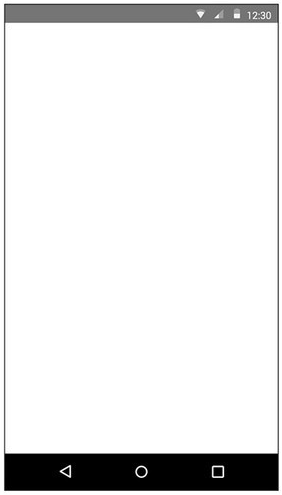
**Image properties screen**

The below table is a brief description of what functionality the user can make use of on each page.

|  |  |
| --- | --- |
| **Application page** | **Description** |
| Start-up screen | The user can choose to upload their image or capture one using the devices camera. |
| Main menu screen | A navigation screen allowing the user to adjust their image using image processing or carry out facial recognition. |
| Adjust Image screen menu | A navigation screen allowing the user to choose specifically what type of image processing to use. |
| Image properties screen | Uses various functionality buttons allowing the user to adjust their images brightness, contrast, exposure and saturation. |
| Image colours screen | Uses functionality buttons allowing the user to adjust the amount of colour in their image in relation to the RGB (Red, Blue, Green) colour channels. |
| Image filters screen | Uses functionality buttons allowing the user to apply various filters to their image such as different types of blur, noise, edge detection and convert the image to greyscale. |
| Face recognition screen | Uses functionality buttons to allow the user to start the facial recognition process. |
| Face recognition results screen | A screen which provides results of the previously carried out facial recognition. |

Wireframe diagrams

The wireframe diagrams in this section are plans of how each screen in the mobile application will look to the end user. They do not take in to account the exact aesthetics of the end product, however they do give a representation of how the requirements listed in section 2.0 are going to be implemented. In developing these screens, the appearance will be designed in XML.

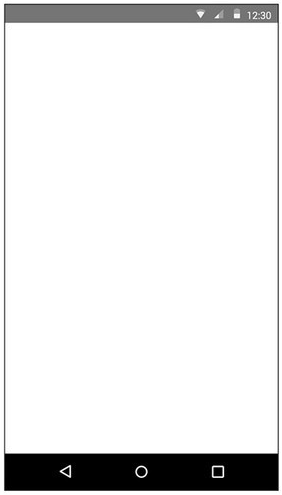
****

Application logo/name (1)

|  |
| --- |
| Name of screen: Start-up screen |
| **Background**: An ImageView used to display a suitable background image to help provide a good user experience. |
| 1. **ImageView**: Displays the application logo. |
| 1. **Button**: Used by the user to allow them to capture an image and navigate to the next screen. |
| 1. **Button**: Used by the user to allow them to upload an image from the device gallery and navigate to the next screen. |

Upload an image (3)

Capture an image (2)

****

Screen name (1)

Image information (2)

Image preview (3)

|  |
| --- |
| Name of screen: Main menu screen |
| **Background**: Displays a suitable background image that does not class or detract from the functionality. |
| 1. **TextView**: Displays the screen name. |
| 1. **TextView**: Displays information about the user’s image. |
| 1. **ImageView**: Provides a preview of the user’s image updated with any changes made. |
| 1. **Button**: Navigates the user to the Adjust image screen. |
| 1. **Button**: Navigates the user to the Facial recognition screen. |
| 1. **Button**: Allows the user to save the current displayed image to their device storage. |
| 1. **Button**: Allows the user to choose or capture a new image. |

Facial recognition (5)

Adjust image (4)

Save image to device (6)

Choose a new image (7)

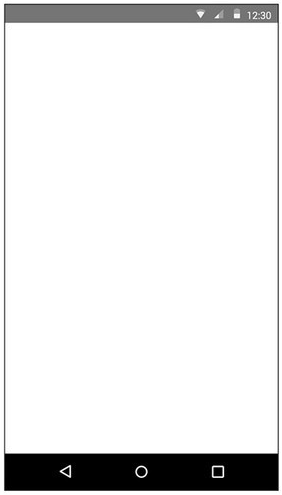
****

Image preview (2)

Filters (5)

Reset image (6)

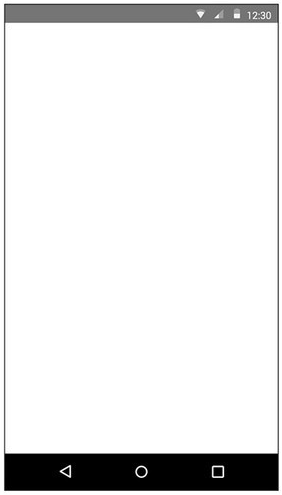
Save changes (7)

Colour ratio (4)

Image properties (3)

Screen name (1)

|  |
| --- |
| Name of screen: Adjust image screen |
| **Background**: Displays a suitable background image that does not class or detract from the functionality. |
| 1. **TextView**: Displays the screen name. |
| 1. **ImageView**: Provides a preview of the user’s image updated with any changes made. |
| 1. **Button**: Navigates the user to the Image properties screen. |
| 1. **Button**: Navigates the user to the Colour ratio screen. |
| 1. **Button**: Navigates the user to the Filters screen. |
| 1. **Button**: Allows the user to reset any changes they have made to their image. |
| 1. **Button**: Saves all changes made and navigates the user to the Main Menu screen. |

****

(3)

(4)

(5)

(6)

Image preview (2)

|  |
| --- |
| Name of screen: Image properties |
| **Background**: Displays a suitable background image that does not class or detract from the functionality. |
| 1. **TextView**: Displays the screen name. |
| 1. **ImageView**: Provides a preview of the user’s image updated with any changes made. |
| 1. **Buttons**: A set of buttons which will use image processing to alter the brightness. |
| 1. **Buttons**: A set of buttons which will use image processing to alter the contrast. |
| 1. **Buttons**: A set of buttons which will use image processing to alter the exposure. |
| 1. **Buttons**: A set of buttons which will use image processing to alter the saturation. |
| 1. **Button**: Navigates the user back to the previous screen. |

Saturation +

Saturation -

Exposure +

Exposure -

Contrast -

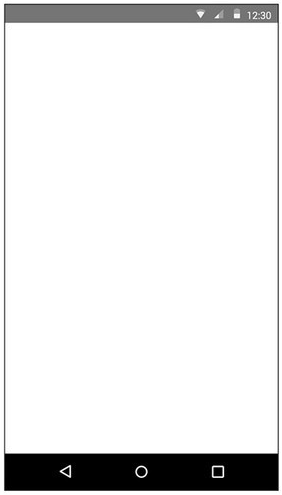
Contrast +

Brightness -

Brightness +

Go back (7)

Screen name (1)

****

(3)

(4)

(5)

Green -

|  |
| --- |
| Name of screen: Image colour screen |
| **Background**: Displays a suitable background image that does not class or detract from the functionality. |
| 1. **TextView**: Displays the screen name. |
| 1. **ImageView**: Provides a preview of the user’s image updated with any changes made. |
| 1. **Buttons**: A set of buttons which will use image processing to alter the amount of red. |
| 1. **Buttons**: A set of buttons which will use image processing to alter the amount of green. |
| 1. **Buttons**: A set of buttons which will use image processing to alter the amount of blue. |
| 1. **Button**: Navigates the user back to the previous screen. |

Image preview (2)

Screen name (1)

Red -

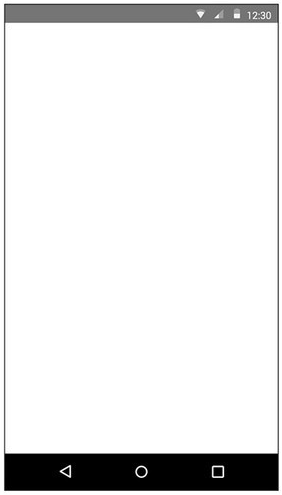
Go back (6)

Red +

Green +

Blue -

Blue +

****

(5)

|  |
| --- |
| Name of screen: Image filters screen |
| **Background**: Displays a suitable background image that does not class or detract from the functionality. |
| 1. **TextView**: Displays the screen name. |
| 1. **ImageView**: Provides a preview of the user’s image updated with any changes made. |
| 1. **Button**: Uses image processing to convert the user’s image to greyscale. |
| 1. **Button**: Uses image processing to simulate adding noise to the user’s image. |
| 1. **Buttons**: A set of buttons which introduce a type of blur to the user’s image. |
| 1. **Button**: Uses image processing to carry out edge detection on the user’s image. |
| 1. **Button**: Navigates the user back to the previous screen. |

Edge detection (6)

Uniform blur

Gaussian blur

Image preview (2)

Screen name (1)

Greyscale (3)

Go back (7)

Noise (4)

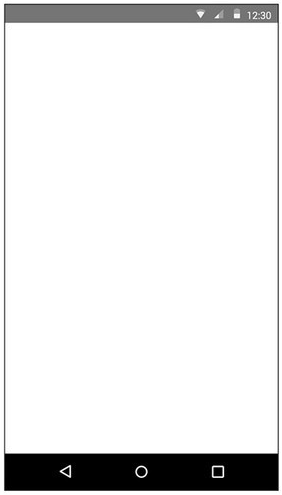
****

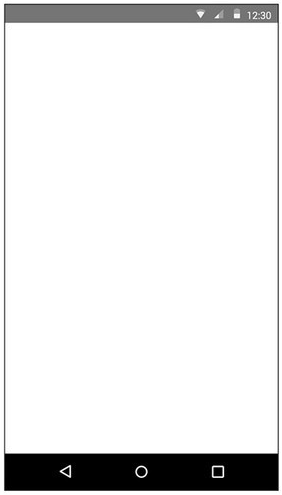
Image preview (2)

|  |
| --- |
| Name of screen: Facial recognition screen |
| **Background**: Displays a suitable background image that does not class or detract from the functionality. |
| 1. **TextView**: Displays the screen name. |
| 1. **ImageView**: Provides a preview of the user’s image updated with any changes made. |
| 1. **Button**: Uses a block-based correlation algorithm to carry out facial recognition on the user’s image. |
| 1. **Button**: Navigates the user back to the previous screen. |

Begin facial recognition (3)

Go back (4)

Screen name (1)

****

(2)

(3)

Matched person: Person name. (4)

Score: score/max score

The matched image

The users image

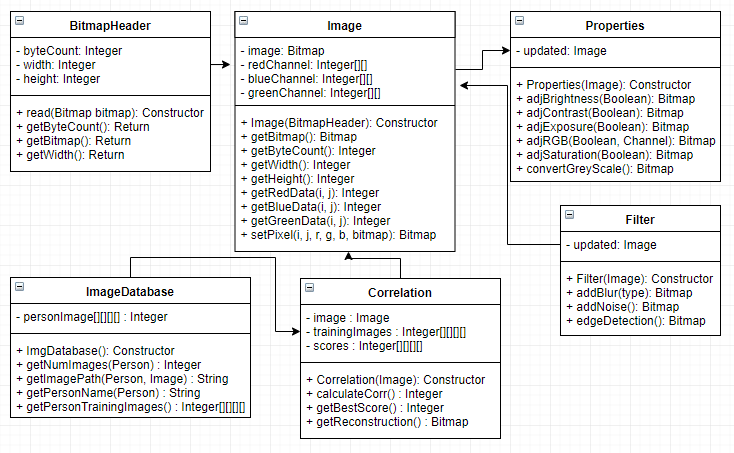
Screen name (1)

Go back (5)

|  |
| --- |
| Name of screen: Recognition results screen |
| **Background**: Displays a suitable background image that does not class or detract from the functionality. |
| 1. **TextView**: Displays the screen name. |
| 1. **ImageView**: The user’s image used in the application. |
| 1. **ImageView**: The image recognised as a match. |
| 1. **TextView**: Displays the results of the face recognition including the person name and the correlation score. |
| 1. **Button**: Navigates the user back to the Main menu screen. |

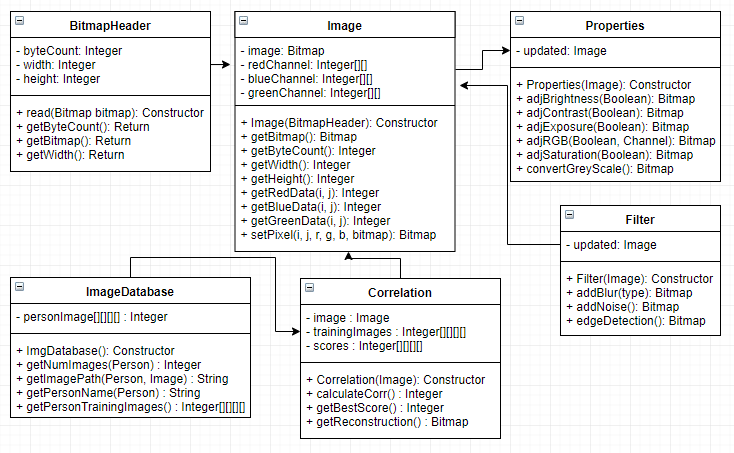
**Development design**

The purpose of this section is to focus on the code structure rather than the aesthetic design of the system. Below is a UML illustrating the design for the main object/processing classes.



Loading and storing the image:

At the core of the functionality the end application will provide is the image itself. Therefore, it is necessary that the code handles the image in a suitable manner and provides a mechanism for extracting and modifying data. As Java is the development language chosen for this project Bitmaps will be used to initialise the image chosen by the user. Bitmaps can consist of numerous parts [10], the main parts to be included in the code are the information header and the image data sections.

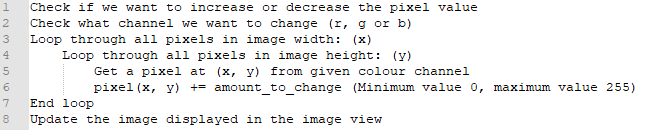
The information header is a section of the image object which stores information such as the file size, width and height of the image. The image data section includes data in relation to the colour channels and pixels in the image. By developing an image object that utilises these two parts of the bitmap loaded, the code will be capable of learning and storing the information required for image processing. The plan of how the image data will be stored and represented in the code can be seen in the UML portion below.

The basics of image processing involve first loading the image itself into the program in such a way that the data associated with the object is meaningful. To do this on creation of the image object the relevant data from the bitmap will be loaded into both the bitmap header and the image object variables. This will involve the population of three two-dimensional arrays, each representing the data contained in each colour channel; red, blue and green [7].

The pixel data is stored in two-dimensional arrays because the code must navigate the picture by use of co-ordinates to find a single pixel value. For example, in a 64x64 resolution image, at co-ordinates (20, 30), we may find a pixel value of [0, 255, 150], each of the three values returned will refer to one of the colour channels. However, I plan to store the data in separate arrays to make the code more readable. Therefore, I will use a getter method to retrieve the pixel at (x, y) for each colour channel.

Adjusting the RGB ratio of the image:

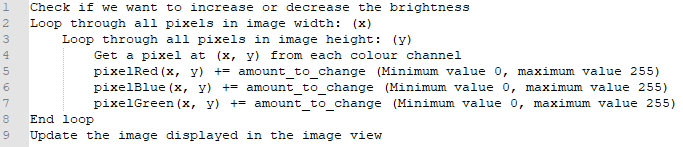
Image processing in the dictionary is defined as “the manipulation or modification of a digitized image” [11], the simplest form of manipulating an image is to change the colour values stored in a pixel at a given location. This functionality will be implemented into the applications code allowing the user to make certain colours more prominent. It will be available for use in the “Image colour screen” and will be usable through buttons which will indicate the user’s intent of either increasing or decreasing the amount of red, blue or green in the image. The below pseudo code is a plan for how this feature will be developed.



Adjusting the brightness of the image:

When increasing or decreasing the overall brightness of the user’s image, it is important to not distort the colour balance, doing this may change how a facial feature is displayed in the image causing it to be unrecognisable during facial recognition (which will be experimented with later). Consequently, the pixel value at (x, y) for each colour channel should be increased or decreased to the same degree. The maximum possible value of a pixel in this format is 255, and the lowest possible value is 0, this should be considered when adjusting the values.

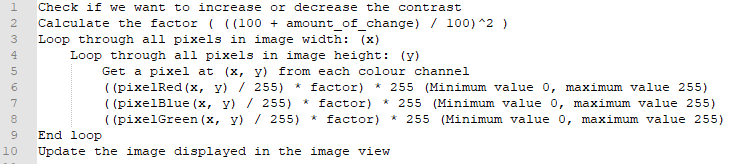
The functionality of being able to increase or decrease the brightness of the user’s image will be available in the “Image properties screen” and will be adjustable using the interface buttons. The below equation and pseudo code give a representation of how this functionality can be achieved.



Adjusting the contrast of the image:

To develop an algorithm which will implement the functionality of adjusting the contrast of an image, the formula which manipulates the pixels at (x, y) will differ depending on the property we want to change. When dealing with contrast we must use a formula which will dramatically adjust the value stored in a pixel in-order to differentiate much more between the image’s colours. The approach taken by Francis G. Loch [12] is similar to the implementation to be used for this solution but with some minor adjustments (See below formula).

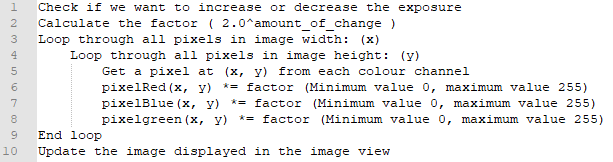
User buttons in the “Image properties screen” will make use of an algorithm which will implement the below formula allowing the user to increase or decrease the contrast of their image. The below pseudo code gives a better idea of how this will be implemented.



Adjusting the exposure of the image:

Exposure levels refer to how light or how dark an image will appear when it has been captured by a person’s camera [13]. However, it is not to be confused with brightness. Exposure is a result of settings on a camera that effect aperture, shutter speed and ISO speed. Depending on the balance of these three settings, the result will be either an over-exposed or under-exposed image.

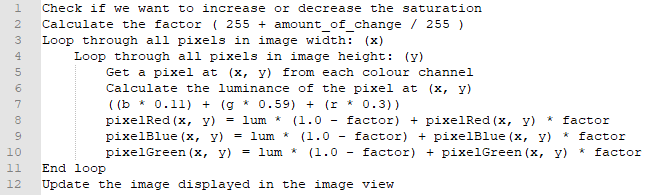
The reason for developing a feature which will enable the user to adjust the image exposure is to give a simulation of an image which may be over-exposed or under-exposed and see what effect this may have on face recognition accuracy. To do this, like contrast, a factor of exposure is used. The factor is multiplied against the current pixel value at position (x, y) accounting for the maximum possible pixel value to give the new exposure-based pixel value. The formula and pseudo code (next page) provide a better understanding of how the code will accomplish this. (See next page for formula and pseudo code design)



Adjusting the saturation of the image:

The saturation of an image can be described as the amount of colour (or the intensity of certain colours) represented in an image [14]. In-order to calculate and modify this the luminance value of a pixel must be considered, this is because if we want to increase the saturation of a pixel, the colour should intensify more if the luminance value of that pixel is greater than others. The calculation to be completed on the pixel values of the user’s image is derived from multiple sources [15, 16] and adjusted slightly.

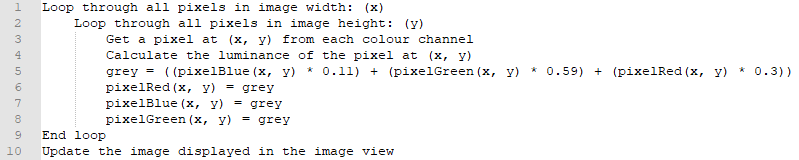
The direction of change for the saturation will be determined by which buttons are being pressed by the user on the “Image properties screen”. These buttons will execute the algorithm represented by the formula and pseudo code below to adjust the saturation of the image.



Converting the user’s image to greyscale:

Converting an image to greyscale should speed up most image processing algorithms. This is because when an image is black and white, the three colour channels (red, green and blue) all hold the same pixel value. It is important that this feature is available in the end application as the face recognition database to be used (LFW crop dataset) [8] is also greyscale. Therefore, if at some point the application were to be expanded to allow the user to use their own images, they would be able to make the image compatible using the application itself.

To accomplish the implementation of this feature, a known calculation will be used [17], which will be applied to the pixel values stored at (x, y) throughout the image. This calculation is the same of that used previously in saturation for determining the luminance, although the calculated value this time will be applied directly to each pixel. The result of this calculation is that the value stored at (x, y) in each colour channel will be the same, resulting in a greyscale (or black and white) image. This functionality will be available by use of a button in the “Filters screen”. Shown below is the calculation that will be applied and pseudo code to represent the process involved in the algorithm to apply this to all pixels.

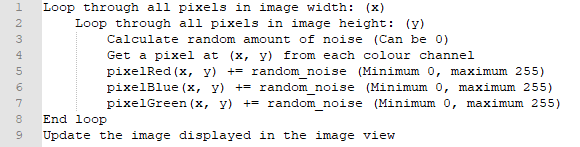


Adding noise to the user’s image:

Image noise is normally a by-product of high ISO levels (used when capturing images at low-light levels) being used in a digital camera and is often referred to as image grain [18]. I aim to implement an algorithm which will manipulate values stored at (x, y) in the user’s image to introduce random noise.

Simulating increased levels of noise can be done using a simple algorithm which randomly chooses a value within a range to add to the current pixel value at (x, y). Inspiration for developing an algorithm to add noise to the image in this manner was taken from a non-working example provided by an online thread [19]. When developed, this should work in both a coloured image, and a greyscale image.

A button in the “Filters screen” will enable the user to add a level of random noise to their image. From this, I can experiment with the effect differing levels of noise in an image can have on the accuracy of facial recognition. The below pseudo code shows the design plan for this algorithm.



Adding blur to the user’s image:

To successfully develop an algorithm which will process the user’s image and result in additional levels of blur, I must make use of image kernels [20, 21]. Image kernels are used for processing an image to produce various outcomes such as modifications to levels of blur, sharpness and complete edge detection. My aim is to give the user the option of experimenting with two different types of image blurring; Gaussian blur and Uniform blur [22].

Gaussian blur is added by convolving a kernel with an image in a circular-like motion to give the effect of smooth blurring, it can be used to reduce image noise and give the effect of an image out of focus. In contrast, Uniform blur, often referred to as box blur convolves a filter with an image directly each time having the same degree of blur. See the example kernels (right) for these blur types. (Note: ‘k’ refers to the given kernel size e.g. 3, 5)

|  |  |  |
| --- | --- | --- |
| **1/k** | **1/k** | **1/k** |
| **1/k** | **1/k** | **1/k** |
| **1/k** | **1/k** | **1/k** |

Uniform blur

|  |  |  |
| --- | --- | --- |
| **1/16** | **1/8** | **1/16** |
| **1/8** | **1/4** | **1/8** |
| **1/16** | **1/8** | **1/16** |

Gaussian blur

Once the desired blurring kernel has been initialised the algorithm must complete a convolution of the image and the given kernel [23]. A convolution is the multiplication of an images colour value at (x, y) and its neighbouring pixels by a value in a matrix (kernel). On completion of this process the result will be the user’s image with an increased amount of blur. The formula by which the convolution process is carried out is described below.

Using edge detection on the user’s image:

As previously mentioned another form of image processing which can make use of kernel convolution to result in a modified image is edge detection. As with blur, there are different types of edge detection to consider. The form of edge detection to be implemented into this application is Sobel edge detection which uses the Sobel-Feldman operator [24].

The Sobel-Feldman operator, known as the kernel, can have varying sizes. To have a similar approach to blur implementation I will a kernel size of 3x3. However, the process of applying this form of edge detection makes use of two kernels, one for horizontal changes and one for vertical changes in the image. Therefore, I will define two kernels, each of size 3x3, which contain the same values as stated in the examples to the right.

|  |  |  |
| --- | --- | --- |
| **+1** | **0** | **+1** |
| **+2** | **0** | **+2** |
| **+1** | **0** | **+1** |

(Horizontal)

|  |  |  |
| --- | --- | --- |
| **+1** | **+2** | **+1** |
| **-** | **0** | **0** |
| **-1** | **-2** | **-1** |

(Vertical)

The formula for convolving these two kernels onto the user’s image will use the same as that of blur. The only difference is that the convolution will be computed twice, once for each kernel (horizontal and vertical), and then a summation will occur at the end before applying the new pixel values to the image. The below equations represent the process by which this process will be computed. Following the completion of each of these calculations the two results themselves will be summed to give the new pixel value. Note: The final summed values must remain within the range of valid pixel values, having a minimum value of 0, and a maximum of 255.

Facial recognition using the user’s image:

I will implement an algorithm which will complete a form of facial recognition that uses patch-based correlation (or block-based correlation) [5]. Initially, as previously described in the solution approach, this implementation will use the LFW crop dataset [8] of training images.

The first stage of implementation is developing a controller which will load all the databases images to a useful location in memory. To accomplish this, the plan is to load the data of each training image, for each person into a 4-dimensional array, allowing me to efficiently iterate each person, and each of their training images to calculate correlations. The image database controller should not need to know any information about a person to load their image, meaning I can grow the database as needed and not need to re-factor the code. Below is a description of how this data will be stored in the 4-dimensional array.

To calculate the correlation score of an image, we must first define the patch. Rather than hard coding a patch size I will initialise patch sizes to be a percentage of the overall image size, which can be influenced by the user’s input for future experiments. Once we have a patch of size ‘N’ from both the user’s image and training image (the nth person, and their nth training image), we can begin to calculate the correlation score for that patch. The process by which the algorithm will iterate the image in patches is described in the diagram below.



User’s image Nth training image

At the point where we have a defined patch from each image, we can begin to calculate the correlation score between the two patches and then accumulate the score for that training image against the user’s image. The calculation used to find a correlation score between two patches is more complex than previous ones used. The below formula [5] describes how this is calculated.

To gain a simplified understanding of this equation we need to break the calculation down into stages. (See next page for more information)

Stage 1:

In stage 1, we consider the total size of the patch and calculate the accumulation of all the pixel data in patch ‘x’, multiplied by the pixel data in patch ‘y’. (Patch ‘x’ is a patch of the user’s image, patch ‘y’ is a patch from person p’s nth training image.)

Stage 2:

In stage 2, slightly differently than stage 1, we calculate the sum of all pixel data in patch ‘x’, calculate the sum of all pixel data in patch ‘y’, and multiply the two values together.

Stage 3:

In stage 3 the calculation becomes a little more complicated. First, we calculate the sum of all pixel data in patch ‘x’ squared. From this we subtract the squared sum of all pixel data in patch ‘x’. We then get the square root of the final value.

Stage 4:

In stage 4, we use the exact same calculation as in stage 3, except this time use patch ‘y’ from person p’s nth training image.

Once the four stages of the calculation have been completed we can then finish calculating the correlation score between the two patches. First, we subtract the value obtained in stage 2 from that of stage 1, and then divide this by the value obtained in stage 3 multiplied by that of stage 4. On completion of this we have a patch score. The patch score will have a score closer to 1 if it has a good correlation, and a score closer to 0 if it does not. Each patch score is accumulated to give the overall image score.

When the correlation algorithm has completed the calculation for every training image per person in the database, we must differentiate between the scores to decipher the closest match. To do this, we simply find what score calculated is the highest, and ensure no scores are tied highest. However, it is important to note that this does not necessarily mean that the matched person is 100% correct, it just means the correlation was the best. Using the score of the best match, I can experiment with the effects of image adjustments such as brightness and blur on the algorithm accuracy.

**Implementation and testing**

The purpose of this section is to discuss how the requirements discussed in section 2.1 have been implemented and what level of testing has been carried out on features throughout development.

**Implementation**

Key implementation decisions:

Before the process of implementing key features began, a few implementation decisions needed to be made to allow the application to meet the requirements previously stated in section 2.1. The first was making the decision to develop and Android application in Java. This allows me to implement image processing algorithms into a platform that can be easily experimented with by a variety of other users (R.1, 3).

The second pre-development decision to be made was to decide on an android API level to support and target (R.2). I discovered that most devices available at the time of development operated on API versions 4.4 through to 7.1. These API’s are otherwise known as KitKat, Lollipop, Marshmallow and Nougat [25].

Therefore, it is vital that the application being developed in this project should support these versions of Android. Taking this information into consideration I decided to support all API versions between 4.0.3 and 8.0 (from Ice-cream Sandwich to Oreo).

User interface implementation:

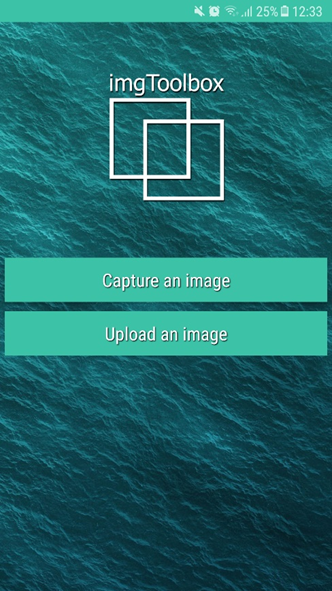
When implementing the user navigation system, I strictly followed the system design illustrated in section 3. The end goal for the user interface and navigation system is to provide the end user with an intuitive experience and facilitate them in using the functionality the app provides. To do this, it was important that I considered various user interface design principles [26] when implementing design elements [27] which would drive the user’s experience.

The implemented user interface and navigation system can be seen in Appendices AA through AH. From these, a simplistic, consistent user interface design has been used along with clearly labelled functionality and a strategic use of colours to direct the user’s attention to important items, resulting in a good and intuitive experience (R. 4, 5, 9). This user interface makes use of various interface elements such as buttons, and user input fields to facilitate navigation and interaction with functionality.

On the relevant application screens, with thought given to the layout, I have included a preview window (R.10) which enables the user to view and better control the changes they are making to their image by use of the image processing functionality. This was an important requirement, as not only does it facilitate the user’s experience when processing the image, it also allows us to compare how accurate the facial recognition results are (this can be seen in Appendix AH).

The user interface was designed in XML and makes use of relative screen layouts [28] to provide the best possible experience for a wide range of devices (R.24). This was an important requirement to meet as there are so many devices in todays society of all different shapes and sizes, so I wanted my application to be suitable for many.

For the purpose of making an application which is more easily identified by the user, I have decided to use the name “imgToolbox”, the logo for which can be seen below. The reason for this name (R.22) is because it portrays what the app is, an application providing a ‘toolbox’ of functionality enabling the use to alter various properties. The logo (R.23) portrays the facial recognition functionality included, with the overlapping squares representing the patches used in calculating correlation scores.



Logo displayed on start -up screen.

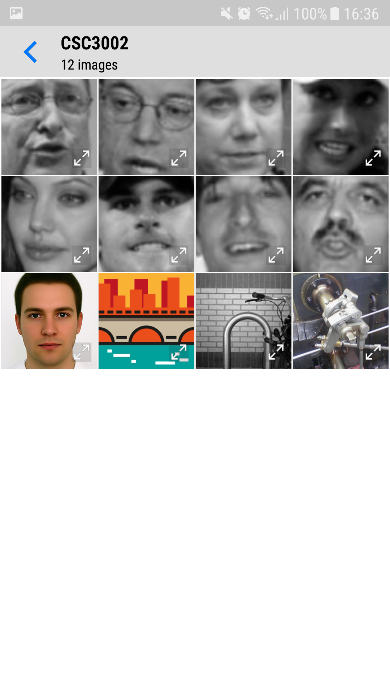
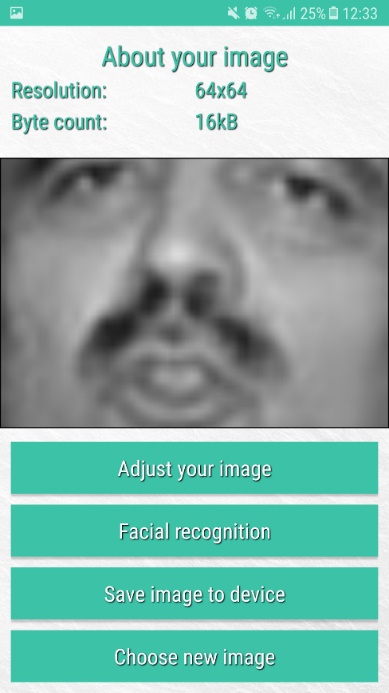
Uploading, capturing and saving the user’s image:

A key fundamental of the user’s experience is their ability to manage the image being used on the app. This involved the implementation of features for uploading, capturing and saving a user’s image (R.6, 7, 8).

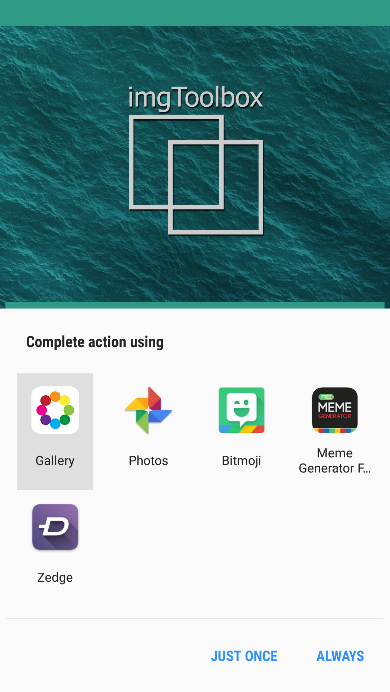
The process of uploading an image to the application is controlled in StartupScreen.java. This is the class which controls all functionality available at start-up (Appendix AA), which include uploading and capturing an image. A button listener method *onClickUploadImage* (Appendix AI) is used to register the choice of the user and begin the process of uploading an image.

This process begins by first retrieving the devices image gallery directory, then using an android intent [29] to start an activity. The new activity will get the activity result code, which is used to decide between uploading or capturing an image. On retrieval a code signifying that the user wants to upload an image, the *onActivityResult* method (Appendix AK) decodes the image by use of input streams and assigns the image to a bitmap object.

Once the user’s image is assigned to a bitmap object, it is then managed by the *getManagedImage* method (Appendix AL) from the same class. This method is used to check the image size, and scale it down if necessary whilst maintaining the image aspect ratio. The reason for this being large images uploaded by the user may result in an application crash caused by lack of allocated memory.

After the image size has been validated, the image is passed to the main screen activity where the user can begin processing their image or carry out facial recognition. The below screenshots show the process of a user uploading an image to the application in action.

1. User chooses to upload image (and chooses to use Gallery).
2. User selects the image from Gallery.
3. The main screen is loaded with the user’s image.



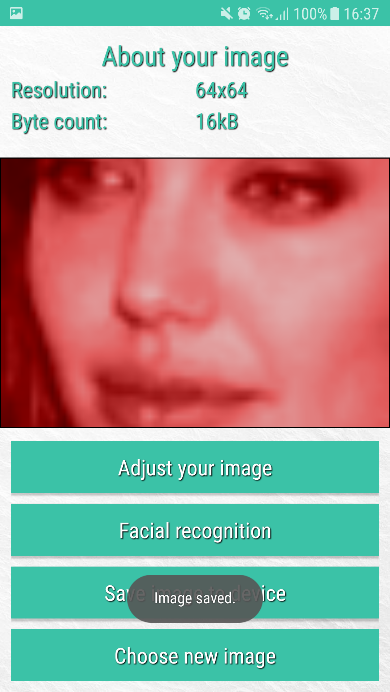
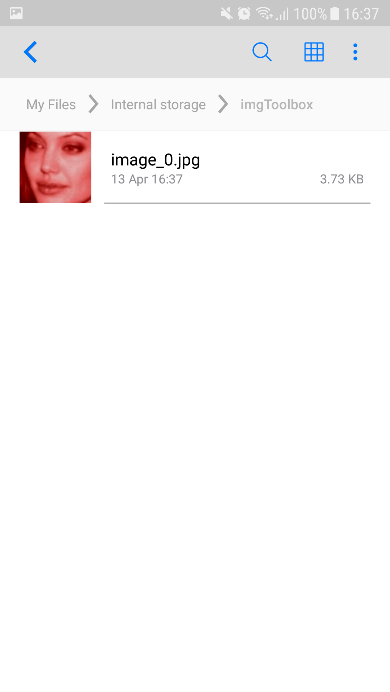
The process of capturing an image from the devices camera for use in the application has been implemented in a similar fashion to uploading an image. A button listener method *onClickCaptureImage* (Appendix AM) registers the choice made by the user and makes use of android intent [29] to start an activity for using the devices camera.

A sub-section of the *onActivityResult* method (Appendix AN) then retrieves the user’s image from the data contained in the request result. Due to a wide range of devices capable of using this application, many of these will be able to capture high definition images. Therefore, the *getManagedImage* method (Appendix AL) is used to ensure the image being used will not result in a crash caused by lack of application memory. The users image is then passed to the main screen activity where they can begin making use of the features the application provides.

The ability of the user to save a latest version of their image, including any changes made by the image toolkit, is provided on the application main screen (Appendix AB) by the *Save image to device* button.

Once this button is clicked, the *onClickSaveToDevice* method from MainScreen.java is executed. This method first requests the necessary permissions required to save to the devices storage. The method then initialises a directory to where the users image will be saved (/imgToolbox/image\_name.jpg). Next the code makes use of a try/catch statement which will use a file output stream object to save the image to the initialised directory, handling any exceptions which may occur. The below screenshots illustrate this process.

1. User makes various changes to their image.
2. User presses *Save image to device* button.
3. Image is saved to “/img/Toolbox” directory.



Storing the image data:

Once the user’s chosen image is passed into the main menu screen, it is initialised as an image object. The Image.java class has been developed to provide a means of storing the user’s image data in a meaningful manner. Once the image object is initialised the constructor method executes the *read* method (Appendix AP).

The purpose of the read method in the image object class is to organise the image data into allocated memory, so that it can be accessed easily for image processing. First the user’s image is loaded into the bitmap header where information regarding its width, height, and byte count are stored. Next three arrays, one for each colour channel, are created. The image colour data is then loaded into each of these arrays for ease of access.

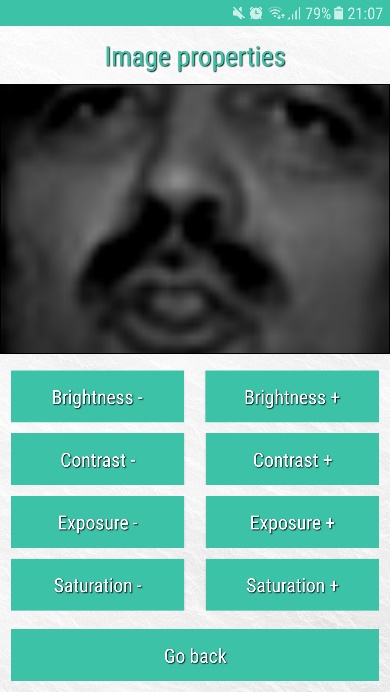
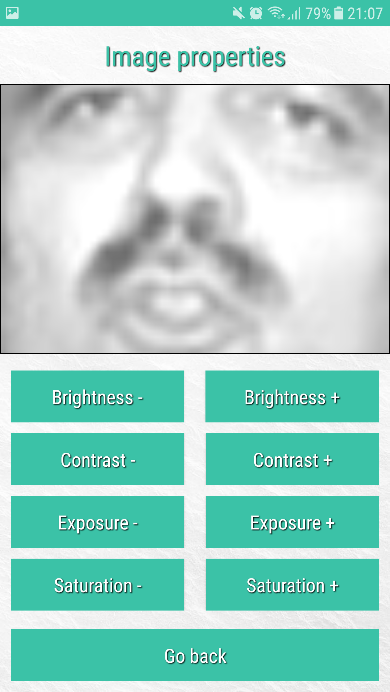
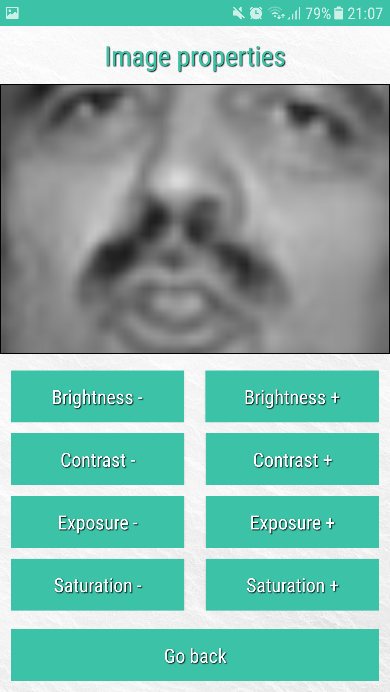
The Image.java object class also provides useful accessor and mutator methods [30]. An example of one implemented is the *setPixel* method (Appendix AQ). This method is provided with various parameters such as the image pixel data for each channel, the co-ordinates of the pixel to change and the user’s image. It then sets a new value for the given pixel in each channel, keeping the value within the bounds of minimum and maximum pixel values, resulting in a change to how the user’s image looks.

Image processing toolkit:

1. Brightness

The first image processing feature implemented was enabling the user to increase or decrease the brightness of their image (R.12). The ability to do this has been provided in the ImagePropertiesScreen.java class. This class represents the image properties screen (Appendix AD) and manages the user interaction with the various buttons. A button listener method is user for each of the increase and decrease brightness buttons (Appendix AR).

In each of the button listeners, an *ImageProperties* object is created with the original image object we have. The *Properties* object class provides a variety of image processing algorithms which can be used to modify the user’s image. In this case, the button listener activated will execute the *adjustBrightness* method (Appendix AS) from the ImageProperties.java class.

The *adjustBrightness* method (Appendix AS) is passed a Boolean, which indicates the change to brightness the user wants. The algorithm then iterates through every pixel in the image using the width and height as a guidance, adjusting the value of the red, blue and green colour values of each pixel by a factor of 10. The result of this algorithm is either a darker, or lighter version of the user’s original image depending on which button was pressed. This change is made apparent to the user by updating the image preview pane with their updated image. An example of the changes made possible by this algorithm is illustrated below.

Decreased brightness. Original brightness. Increased brightness.

1. Contrast

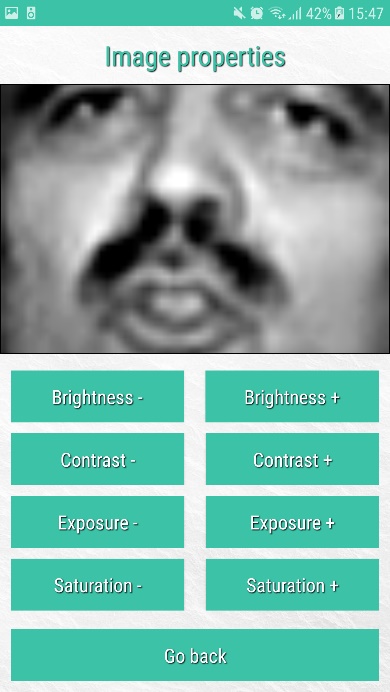
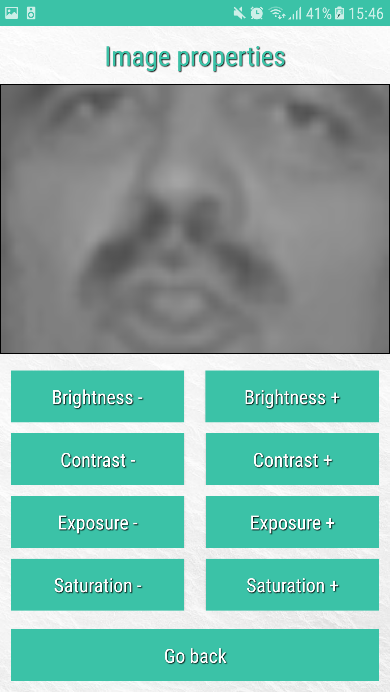
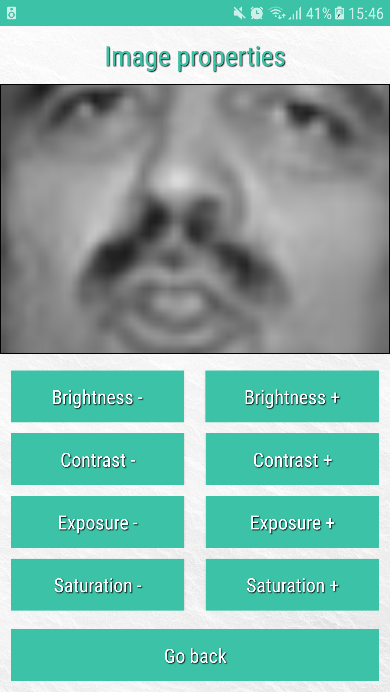
The next image processing feature implemented enables the user to increase or decrease the level of contrast in their image (R.13). Like brightness, this feature is available for use in the image properties screen (Appendix AD) controlled by the code included in the ImagePropertiesScreen.java class.

To begin the process of increasing or decreasing contrast levels, the user must navigate to this screen and interact with one of the two buttons provided for this feature. Once the user has clicked one of the buttons, a button listener method for this element is activated (Appendix AT). Depending on what button listener is activated by the user’s input, the image object representing their image is used to initialise the *Properties* object class which provides the algorithm for manipulating the images pixel data to alter contrast levels.

The button listener uses the *Properties* object to execute the *adjustContrast* method (Appendix AU), passing in the necessary Boolean which indicates either an increase or decrease. The *adjustContrast* method is used to carry out the calculation designed in section 3.3 which has been shown again below.

Once the factor of change ‘*contrast*’ variable is initialised, the algorithm iterates over each pixel co-ordinate in the image using the width and height as a guidance. At each iteration of the nested for loop, the calculation is carried out on the pixel data held in each colour channels array list at that co-ordinate. Then the *setPixel* method available in the image object itself is used to set the updated pixel values, ensuring that the new pixel value does not exceed the boundaries of 0 and 255.

Once the *adjustContrast* method has finished execution, the button listener the updates the image preview located on the user interface so that the user can view the change made to the image. Below is an example of the contrast changes made possible by this implementation.



Decreased contrast. Original contrast. Increased contrast.

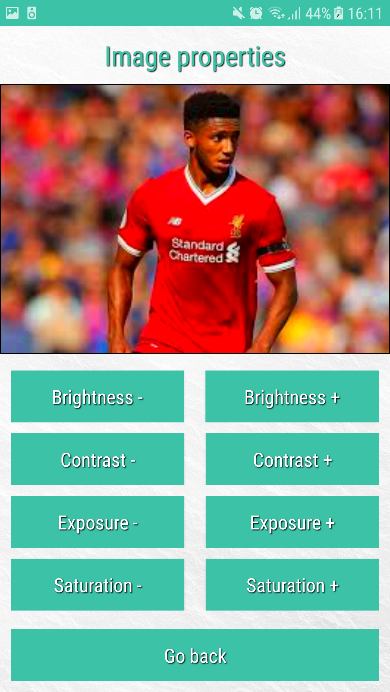
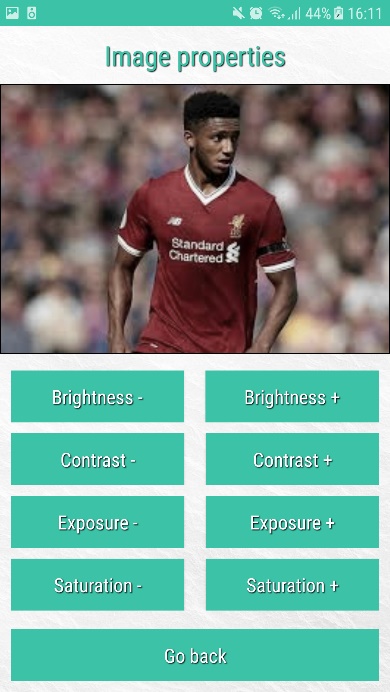
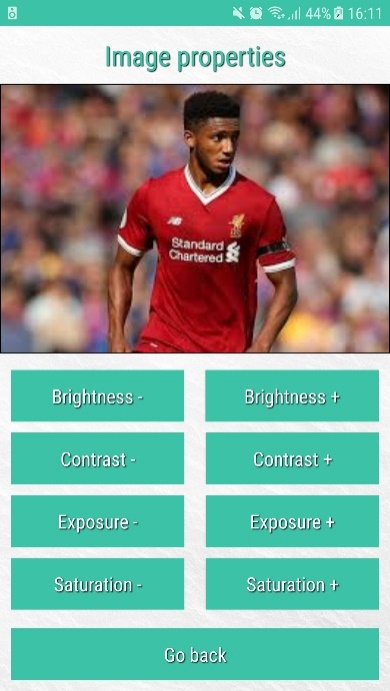
1. Saturation

The image toolkit also provides the ability of the user to increase or decrease the colour saturation in their image (R.14). This refers to the intensity of primary colours in the user’s image. This feature is available for use in the image properties screen (Appendix AD). Once the user has pressed a button to either increased or decrease the levels of saturation in their image, an action listener for that button is then activated (Appendix AV).

The action listener for the button selected initialises the *Properties* object with the user’s image object in-order to access the algorithm for adjusting saturation. The *adjustSaturation* method (Appendix AW) is executed passing in a Boolean as a parameter indicating the user’s choice of either increasing or decreasing saturation levels. The *adjustSaturation* method completes the formula illustrated in section 3.3 (See below).

The *adjustSaturation* method initialises the saturation factor to be applied to the image, which is then used in the main portion of the algorithm. The algorithm iterates through the image pixels using a nested for loop with the image width and height as the boundaries. At each iteration, the luminosity value of the pixel is calculated using the value contained in each colour channel at that index.

Next, the algorithm calculates a new red, blue and green pixel value using the saturation factor and the luminosity value. The new pixel values are saved to their respective colour channel array by using the *setPixel* method provided by the user’s image object. The new updated image is then displayed on the preview panel of the image properties screen to enable the user to view the modifications made. Below is an example showing the effects this algorithm can have on the user’s image. In this case, a coloured image has been used to illustrate the modifications made possible.



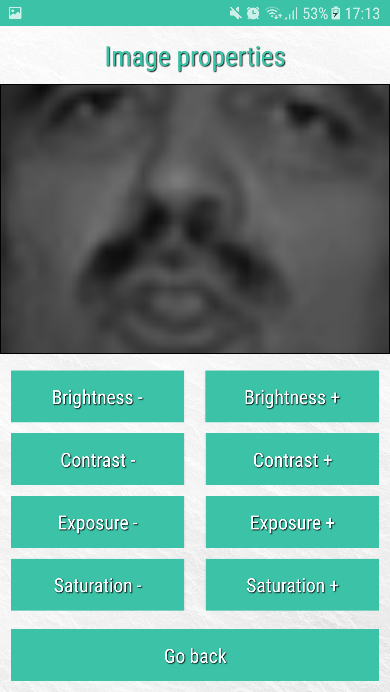
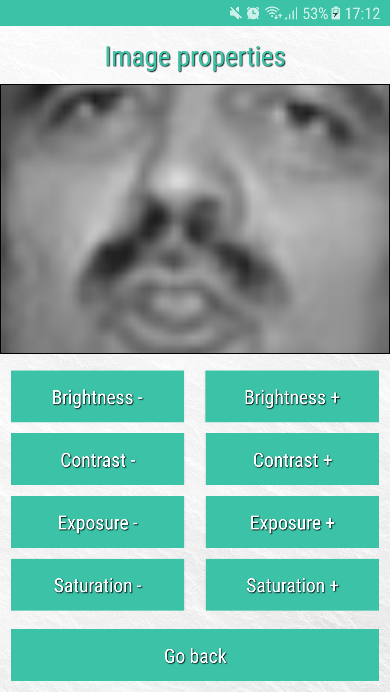
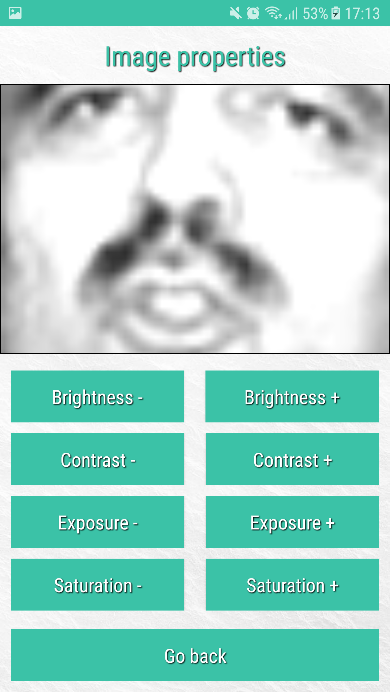
Decreased saturation. Original saturation. Increased saturation.

1. Exposure

Implementing an algorithm which provides the user with the ability to increase or decrease the levels of exposure in their image (R.15) is an important feature for later experiments. This is because in a real-life scenario, an image captured on a bright day may be effected by sunlight, resulting in an image which is over exposed, I aim to investigate what effect this scenario may have on the accuracy of recognition results.

This feature is available for use in the image properties screen (Appendix AD) by using the two buttons provided to either increase or decrease the effect. Once the user selects a button, the corresponding button listener is activated (Appendix AX). Once the button listener is activated it begins completing various functions. The first is to initialise the image object with the user’s image, and the *Properties* object which provides the algorithm for adjusting Exposure. The *adjustExposure* method (Appendix AY) provided by the *Properties* object is executed with a Boolean parameter indicating the user’s desire to either increase or decrease.

The Boolean passed as parameter has an effect on the calculation of the exposure factor used to manipulate the pixel data. The algorithm then iterates through the data in the user’s image, multiplying the exposure factor by the current value held in the pixel. This causes a more dramatic change in the pixel values giving the effect of an over or under exposed image.

The *adjustExposure* method then returns the updated version of the user’s image which is then displayed on the screens preview panel. This gives the user a visual representation of changes made by the algorithm. Below is an example of the features possible outcomes.

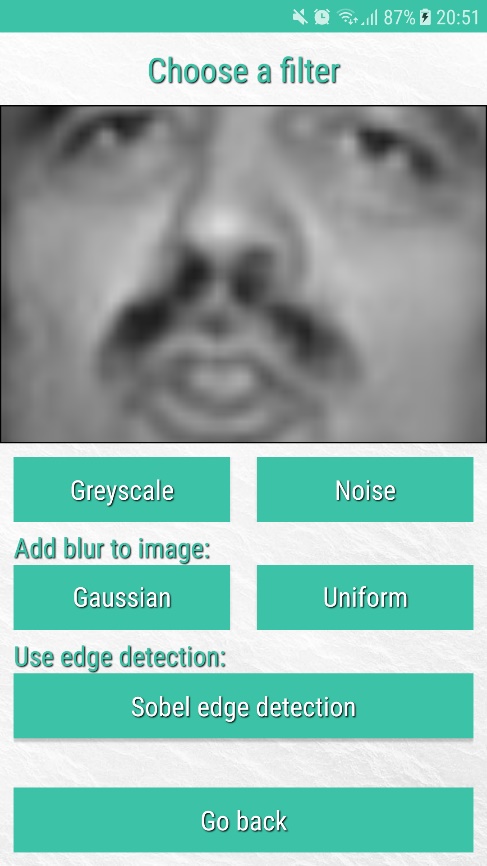
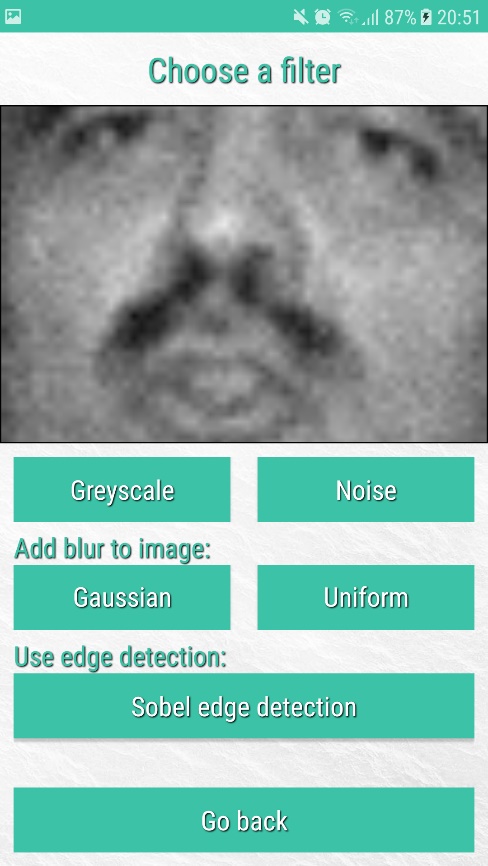
Decreased exposure. Original exposure. Increased exposure.

1. Noise

Noise is generally more prominent in an image that has been taken in a poorly lit or dark environment, which may be the case in the in some surveillance cameras or access systems capturing images of a person’s face for recognition.

This feature is available to use on the applications image filter screen (Appendix AF) (R.16). The noise button activates a button listener (Appendix AZ) which initialises the *Filter* object, this object provides various methods for applying a filter to the user’s image. In this case the method used is *addNoise* (Appendix BA).

The *addNoise* method uses a nested for loop to iterate through the images vertical and horizontal axis. At each iteration, a factor to add to the current pixel value is calculated. In this case it can either be -5, 0 or +5. This gives the impression of a noisy image. The *setPixel* method is used to apply the new pixel value. Once all pixels have been iterated the updated image is returned and then displayed on the preview pane to display the change to the user. The screenshots displayed on the next page demonstrate the possible effect to the user’s image.



Original image. Increased noise levels.

1. Blur

As mentioned in section 3.3, when implementing blur (R.17) there are two different forms to consider. The first of which is Gaussian blur. Gaussian blur gives a smoothing effect to an image and is normally used to reduce noise and detail in an image. To do this, a kernel is used, which is represented in the code as a 2D array of size 3x3. The values held in this 2D array are fractions calculated depending on the current location in the kernel.

Uniform kernels use consistent fraction values throughout the 2D array to be convoluted against the images pixel values. The below diagrams repeated from section 3.3 give an example of the values used.

|  |  |  |
| --- | --- | --- |
| **1/16** | **1/8** | **1/16** |
| **1/8** | **1/4** | **1/8** |
| **1/16** | **1/8** | **1/16** |

Gaussian blur

|  |  |  |
| --- | --- | --- |
| **1/k** | **1/k** | **1/k** |
| **1/k** | **1/k** | **1/k** |
| **1/k** | **1/k** | **1/k** |

Uniform blur

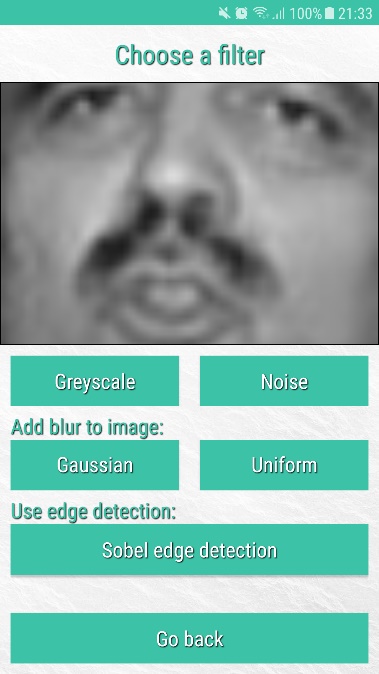
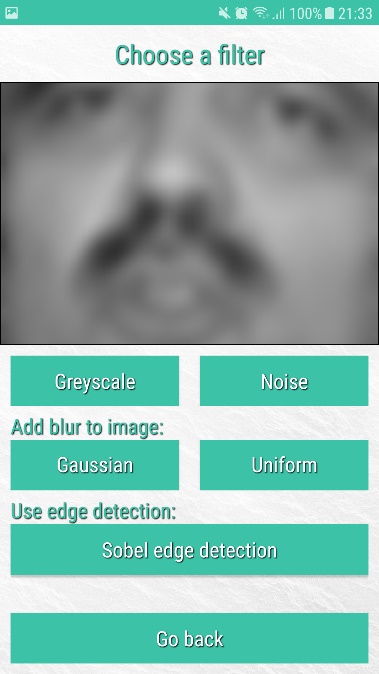
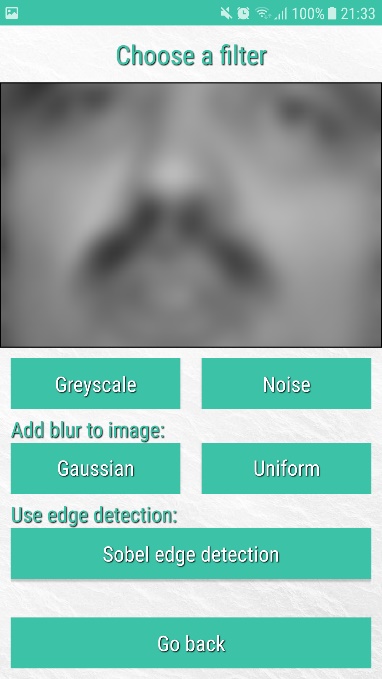
The ability to add blur to an image is available on the *Filter* screen (Appendix AF). By choosing which form of blur to add by using one of the labelled buttons, a different kernel is generated. Depending on the selected button, a button listener method is executed (Appendix BB). This method initialises the *Filter* object with the user’s image and calls the *addBlur* method (Appendix BE) passing in a Boolean which indicates the type of blur: Gaussian or Uniform.

The *addBlur* method (Appendix BE) initialises the kernel size to be 3x3 and deciphers the type of blur to be used from the Boolean. From within the *addBlur* method, a 2D array representing the blurring kernel is retrieved by calling either the *gaussianKernel* (Appendix BC) or *uniformKernel* (Appendix BD) method.

Once the kernel is retrieved, the algorithm first creates a copy of the colour channel data for the user’s image. This is needed to safely convolute the kernel onto the image without disrupting surrounding pixel values. The algorithm then iterates through the image data using a nested loop, this time excluding a small boundary of the image to avoid iterations of the algorithm going out of bounds. At each iteration the kernel is convoluted onto the pixel data values, which gives the resulting effect of a blurred image. The below formula describes this calculation.

The summed pixel values calculated during the convolution process are then assigned to the pixel and are set using the *setPixel* method of the image object which ensures the value doesn’t not exceed that minimum and maximum pixel values. The updated image generated by the *addBlur* method is then displayed to the user, showing their image with added blur. It is important to note that it does not matter which form of blur the user chooses, the convolution process is the same, only the kernel differs.

Having the ability to simulate blur in an image is an advantage when investigating different factors which may improve or weaken the accuracy of the facial recognition algorithm. Adding blur to the user’s image can help to reduce noise or give the effect of an out of focus image gathered in a real-life scenario.

Below are some examples that illustrate the effect this algorithm can have on the user’s image.

Original image. Added Gaussian blur. Further Gaussian blur.

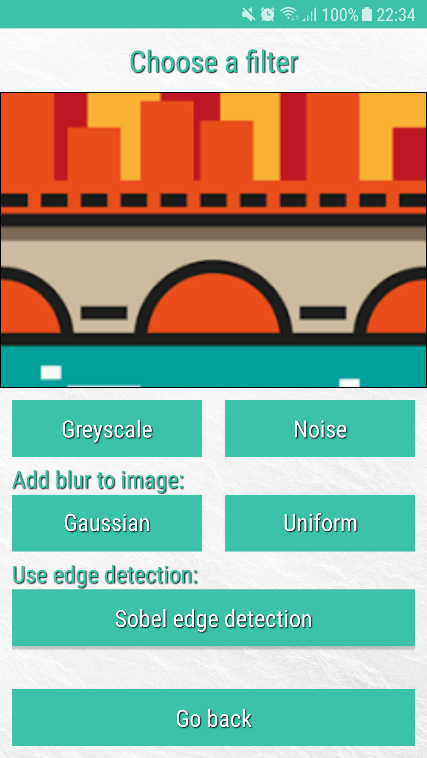
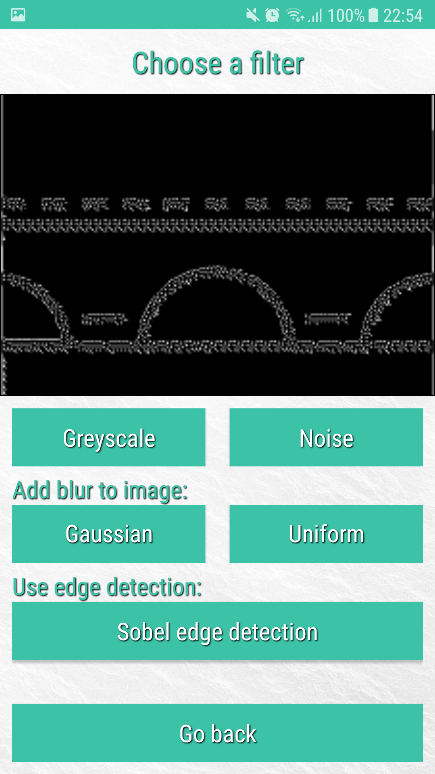
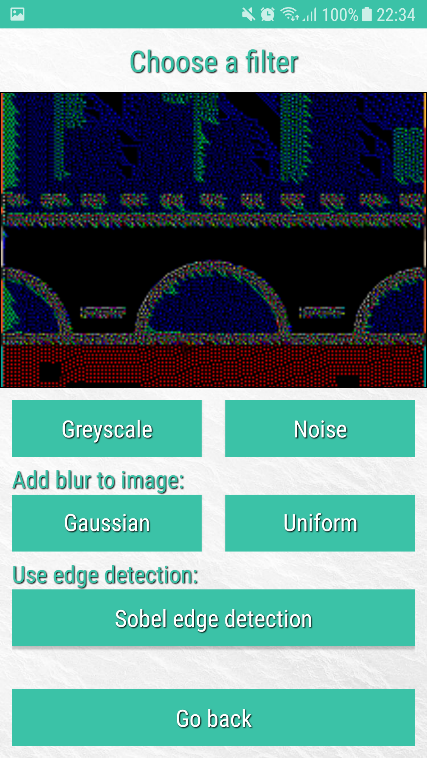
1. Edge detection

The next implemented feature readily available from the image processing toolkit is the ability of the user to carry out edge detection on their image (R.18). This feature is available on the filters screen (Appendix AF) by using the edge detection button. As described in section 3.3, the type of edge detection to be implemented is known as Sobel edge detection. This is because it makes use of Sobel kernels to complete the convolution.

Once the user selects the Sobel edge detection button on the filter screen, an action listener is activated (Appendix BF). This action listener initialises the *Filter* object which provides the necessary algorithm for edge detection. When the *edgeDetection* method (Appendix BH) is executed, it first initialises the two required kernels of size 3x3. The kernels are retrieved by using the *sobelKernelGx* and *sobelKernelGy* methods (Appendix BG). These methods populate the kernels represented by two 2D arrays with the values described in section 3.3. One of the kernels is used to detect vertical edges and the other is to detect horizontal edges.

The algorithm then generates a copy of the pixel data from the user’s image and stores it in a temporary 2D array. This is necessary when carrying out convolutions to minimise risk of mis-analysing pixel values.

The algorithm then iterates over the pixel data in the image using a nested for loop and carries out the convolution calculation described in the design of this feature. This calculation is used on the current pixel value in each of the three colour channels, in a simplified black and white image it would only be necessary to convolute the kernel with one channel.

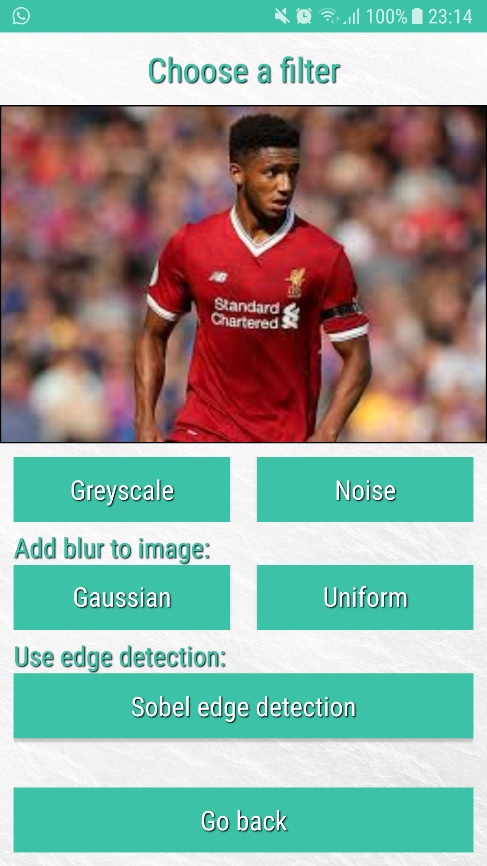
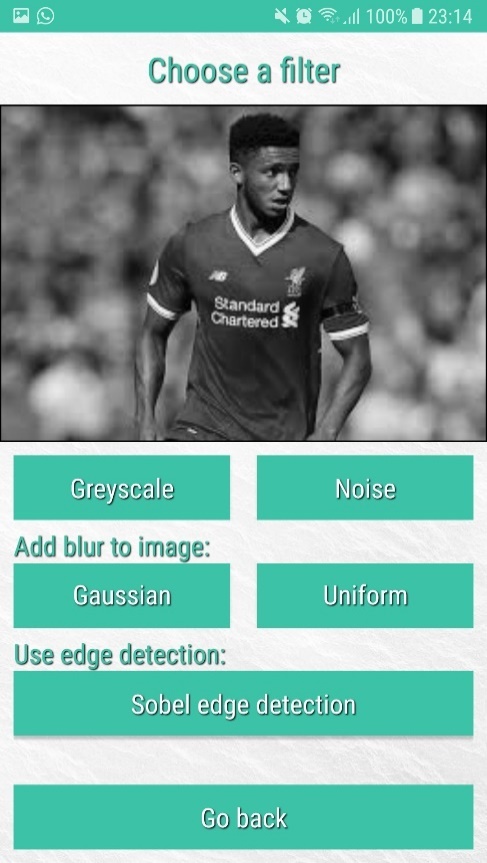
The resulting summed data from each kernel is then added together to give the new pixel value. This data is then applied to the user’s image, ensuring the pixel data remains within the necessary bounds, and then displayed on the filter screen preview pane. This feature is not directly used in the form of facial recognition implemented, however it represents how edges or ‘facial features’ may be able to be recognised in another implementation. The below screenshots so the results produced by this algorithm.

1. Converting to greyscale

The final feature specified in the requirements to be included in the toolkit is the ability of the user to convert their image to greyscale (R.19). This feature was implemented for the filter screen (Appendix AF) for ease of use however uses the *Properties* object class as it does not make use of kernel filters, it simply uses pixel data manipulation.

On the filter screen, once the user selects the greyscale button the corresponding action listener (Appendix BI) activates and initialises the *Properties* object to provide access to the *convertToGreyscale* method (Appendix BJ). The algorithm used to convert the image to greyscale begins by iterating through the image data using a nested for loop as before. At each iteration a simple calculation is carried out on the current iterations data. The below formula describes the calculation carried out.

This calculation is used to gather a luminosity value for the current pixel, and results in the value of each colour channel to be the same. The *setPixel* method located in the image object class is then used to apply these values to the user’s image. The image being displayed on the filter screen is then updated to show the converted image. The below screenshots show the result of this algorithm when used on a coloured image, no change is apparent when used on an already greyscale image.



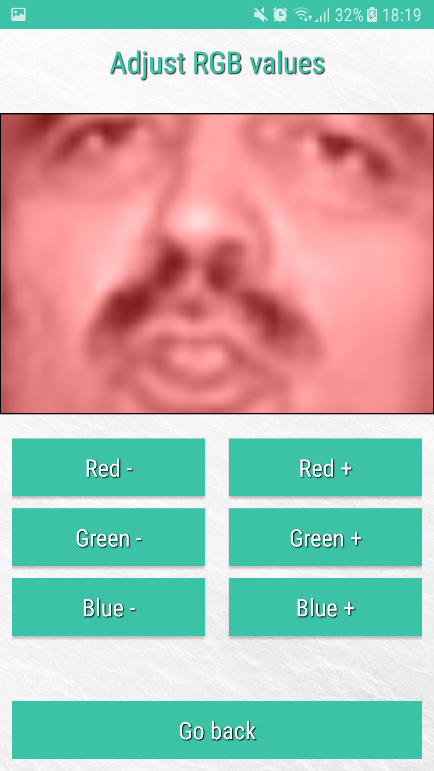
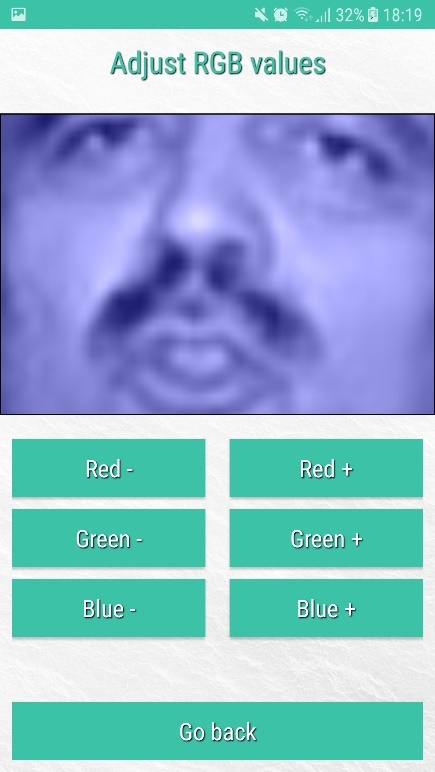
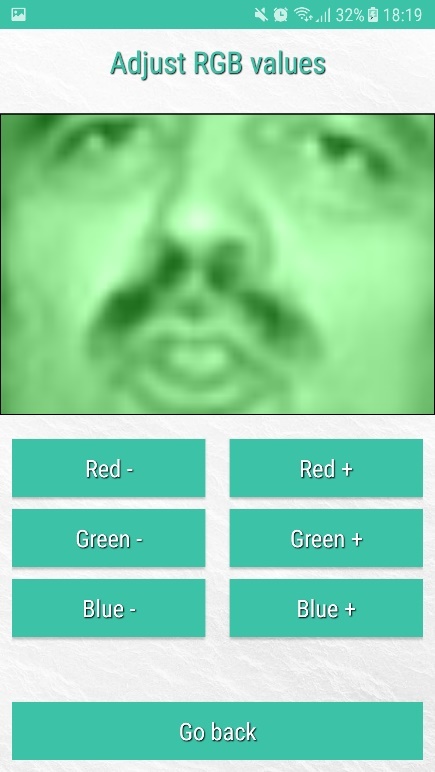
1. Image colour ratio

The ability to manipulate the ratio of colours in the user’s image is not a requirement itself, however is another image processing feature that has been implemented. In comparison with the other features provided in the toolkit this is perhaps one of the more simplistic.

On the *RGB screen* (Appendix AE), a series of buttons allow the user to adjust the colour ratio. Essentially what this feature does is either add or remove the amount of red, blue or green colours from an image. Action listener methods (Appendix BK, BL, BM) are activated depending on the selection made by the user. A Boolean indicating the users choice of increasing or decreasing is used, along with a char (‘b’, ‘g’ or ‘r’) when the action listener executed the *adjustRGB* method (Appendix BN).

The *adjustRGB* method deciphers from the users input whether the user wants to increase or decrease as well as the colour channel to be manipulated. Once this information is gathered, the method adds a pre-calculated factor to the current value held in each pixel.

The result of this is the users image becoming more red, more green, or more blue in colour. The reason this feature is useful is because the user can mix the colours being adjusted and completely change the overall colour of your image. The effects of this algorithm are illustrated on the preview window. The screenshots provided below show this feature working.



Database management:

In-order to implement an algorithm which will provide the feature of facial recognition, I first need to have a database of training images to compare the user’s chosen image to. To do this, as stated in the design of section 3.3, I have implemented the LFW crop dataset [8] (R.21). This is a database of cropped face images for use in unconstrained facial recognition.

To implement the database, I have first downloaded and stored a variety of images per person as assets in the android studio project library (Appendix BO). Every person stored in the database has the same number of training images each, 8, meaning the facial recognition algorithm will maintain a level of fairness when calculating the person correlation scores.

The main code usage for this feature, is the *ImageCorrelation*.java object class. To use the database an *ImageDatabase* object must be created. On construction, the *loadDatabase* method (Appendix BP) is executed which uses the current activity context’s asset manager to access the regarding assets folder. The *loadDatabase* method uses a try/catch statement to get a list of folder names which represent a person. An inner for loop is then used to populate a 2D array of persons to image names, the necessary IO Exceptions are also handled. In loading the folder information in this way, the code does not need to know any information about any person in the database, or their images. This means we can readily grow the database by manually adding more people, or eventually be making the code learn from images uploaded by the user.

Now that we have a 2D array of person information, and their image names, we must load the data held in each image (i.e. pixel values). To do this a 4D array (see explanation on next page) is populated with the pixel values of each image per person. The method that carries out this population is the *getPersonTrainingImages* method (Appendix BQ).

On execution of this method, the 4D array is created and initialised to null, the following for loop iterates through the 2D *persons* array (previously created) to allocate the necessary memory to hold the image data from the images in the database. A nested for loop containing a try/catch statement is then used to iterate through the folder data and load the images.

Within the try/catch, the asset manager is again used, this time loading the image information into a bitmap object rather than simply retrieving information about the folder contents. The bitmap is loaded from the current image path, and then initialised with the *Image.java* object class, where it is converted to greyscale. The reason for this is because the LFW dataset is greyscale also. A final nested for loop is used to load the pixel data of the current image, into the 4D array. This process is repeated for each person and their nth training image. The explanation below gives a better understanding of the data held in the 4D array.

The 4D array is then returned from the method, meaning it can be used in the *ImageCorrelation*.java class for facial recognition purposes.

Unconstrained facial recognition:

The implementation of an algorithm that computes patch-based correlation for unconstrained facial recognition [1, 5] (R.20) is the final stage of image processing in this application. This algorithm uses the previously described database of person training images from the LFW crop dataset [8]. On the facial recognition screen (Appendix AG), the user must input some required information to effect how the algorithm will work. The patch size, is a given percentage size of the overall image, the range refers to a margin of adjustment that can be made to the pixels searched. So, the user can search pixels slightly outside the patch size described, which at times can result in more accurate facial recognition.

Once the user entered the patch-size and range, they click the *Begin facial recognition* button which activates a button listener *onClickCorrelation* (Appendix BS). This button listener makes use of input validation methods (Appendix AG) to ensure the user’s input is of suitable criteria to be used in the algorithm. This method then verifies the user’s own image is of suitable size in comparison to the images in the database (64x64) and initialises the *ImageDatabase* object and the *ImageCorrelation* object (Appendix BT), which initialises various variables used in the correlation algorithm. The *calculateCorrelation* method (Appendix BU) is then executed which returns the matched persons index, from which we can determine this persons name, gather a reconstruction image of highest scoring patches and move forward to the results screen (Appendix AH) were the user’s image and the reconstruction are illustrated, along with the overall correlation score which can be used to decipher the accuracy of the algorithm in this case.

The *calculateCorrelation* method (Appendix BU) implements the patch-based correlation algorithm. This algorithm makes use of the 4D array of person training image data initialised by the database object to iterate through patches of every image. First the algorithm sets the first patch location iteration, which is used across all training images to calculate scores before moving to the next location. Next, we load the pixel data from the user’s image at this patch location and calculate the necessary pre-calculations (sum of all pixels, sum of all pixels squared, the sigma).

Now that we have loaded the user’s image data, we iterate over each person, and each of their training images using for loops. For the nth person, and their nth training, we load the pixel data from the current patch location of that image. Now we have two 2D arrays, one for the patch data of the user’s image, and one for person p’s nth training image. Similar to before, we carry out the necessary pre-calculations (Appendix BV) on the training patch this time (sum of all pixels in the patch, sum of all pixels squared, the sigma).

We now calculate the correlation score (Appendix BY) between the current training image patch and the user image’s current patch data. The score is saved if it is the highest recorded for this patch location, across each person’s training images. Once we have calculated all scores across all images, we move to the next patch location, and repeat the process until we have patch scores for all locations. Below is the formula for calculating the correlation score as previously described in the design section 3.3.

Once we have gathered the scores of all patch locations, for each person and their training images in relation to the user’s image, we can generate a reconstruction (Appendix BW) which gives a visual representation of what patches scored the highest. The image reconstruction generates a new image, which is made up of the highest scoring patch, from each patch location across all database persons and their training images. It is an image which can be made up of patches from multiple images, in some cases from different people if a patch is more alike.

Finally, we carry out a differentiation on the results gathered (Appendix BX). This method uses the array of scores. This is an array which holds the overall score for each person in the database. From this array, we can decipher the highest score, and ensure there is not a tie in results. On completion of this we return the index of the matched person, which can be used along with the *ImageDatabase* object to find the name of the matched person.

The below image shows the process of iterating patch locations for one person. Once the correlation has been calculated between the persons training image patch and the user’s image patch, we move to the next location on every image and repeat the process until finished.



Throughout the development process for this application, I have made use of image processing algorithms (R.11) which have been developed by myself or influenced by research before development. As a result, I have avoided using external packages such as OpenCV, which provide a variety of features.

Code architecture: UML & Javadoc

As designed, the application code has been developed in a manner which easily facilitates future development of new features (R.25). The addition of new features will have minimal to no impact and previously implemented features. To accomplish this, I felt it was important to ensure that the current application features have minimum dependency on other methods.

Throughout development I made use of official Javadoc [31] standards when commenting code. On completion of the development cycle, this allowed me to generate an in-depth analysis of every method in the codebase. Not only does this ensure clarity in code, it also gives a good understanding of current code in case it is refactored or enhanced in the future. The HTML files generated can be found in the Git repository for this project [32].

**Testing**

The purpose of this section is to discuss how the developed application has been tested, and what test cases have been used to ensure the robustness of the code.

To test the final application, I have carried out a variety of both Android automated unit tests for both navigation and toolkit algorithms [33, 34], and a variety of manual tests in-order to achieve a high percentage of test coverage in terms of the developed features. See below table for more information with regards to the automated unit tests carried out on the application navigation and features.

|  |  |  |
| --- | --- | --- |
| # | Automated test case | Result |
| 1 | Verify the image object has been initialised with the correct data by validating the actual image size. | Pass |
| 2 | Verify the image object has loaded the correct red colour channel array. | Pass |
| 3 | Verify the image object has loaded the correct green colour channel array. | Pass |
| 4 | Verify the image object has loaded the correct blue colour channel array. | Pass |
| 5 | Validate expected image data outcome as a result of the brightness increase algorithm from the toolkit. | Pass |
| 6 | Validate expected image data outcome as a result of the brightness decrease algorithm from the toolkit. | Pass |
| 7 | Validate expected image data outcome as a result of the contrast increase algorithm from the toolkit. | Pass |
| 8 | Validate expected image data outcome as a result of the contrast decrease algorithm from the toolkit. | Pass |
| 9 | Validate expected image data outcome as a result of the exposure increase algorithm from the toolkit. | Pass |
| 10 | Validate expected image data outcome as a result of the exposure decrease algorithm from the toolkit. | Pass |
| 11 | Validate expected image data outcome as a result of the saturation increase algorithm from the toolkit. | Pass |
| 12 | Validate expected image data outcome as a result of the saturation decrease algorithm from the toolkit. | Pass |
| 13 | Validate expected image data outcome as a result of the greyscale conversion algorithm from the toolkit. | Pass |
| 14 | Validate expected image data outcome as a result of the noise simulation algorithm from the toolkit. | Pass |
| 15 | Validate expected image data outcome as a result of the gaussian blur simulation algorithm from the toolkit. | Pass |
| 16 | Validate expected image data outcome as a result of the uniform blur simulation algorithm from the toolkit. | Pass |
| 17 | Validate expected image data outcome as a result of the Sobel edge detection algorithm from the toolkit. | Pass |
| 18 | Verify on creation of the ImageDatabase object, the 4D array has been populated with each person’s image data from the assets. | Pass |
| 19 | Using specific input data (patch-size, range), ensure the facial recognition algorithm obtains the correct result as a match. | Pass |
| 20 | By checking the presence of UI elements, verify the Start-up screen loads. | Pass |
| 21 | By checking the presence of UI elements, verify the Start-up screen loads and can navigate to the Main screen. | Pass |
| 22 | By checking the presence of UI elements, verify the Start-up screen loads and can navigate through the Main screen to the Adjust Image screen. | Pass |
| 23 | By checking the presence of UI elements, verify the Start-up screen loads and can navigate through the Main screen and Adjust Image screen to the RGB ratio screen. | Pass |
| 24 | By checking the presence of UI elements, verify the Start-up screen loads and can navigate through the Main screen and Adjust Image screen to the Image Properties screen. | Pass |
| 25 | By checking the presence of UI elements, verify the Start-up screen loads and can navigate through the Main screen and Adjust Image screen to the Image Filter screen. | Pass |
| 26 | By checking the presence of UI elements, verify the Start-up screen loads and can navigate through the Main screen to the Facial recognition screen. | Pass |

This test suite has been developed within Android Studio using ActivityTestRules [36] to load the necessary application screens in-order execute the variety of tests. The test suite [35] is executed in Android Studio using a suitable emulator device or hardware device running a supported Android OS.

As previously mentioned, I have also carried out various manual tests to stress test the application, verify physical results of algorithms used on an image and validate user input. This is a form of backup for the unit tests, but also allows me to try to cause the application to crash, or encounter scenarios and actual user may present. See below table for a description of manual tests carried out.

|  |  |  |
| --- | --- | --- |
| # | Manual test case | Result |
| 1 | Attempt to load an image of large resolution. Expected: Application should down size the image maintaining the aspect ratio to prevent a crash. | Occasional crashing |
| 2 | Upload a new image to the application 10 times to test the application ability to “clean up” previously loaded images. Expected: Application should allow this without crashes. | Occasional crashing |
| 3 | Using the application upload an image and adjust the brightness up and down, verify the changes to the image are correct, and reset the image. Expected: The brightness should increase and decrease, once the reset button is clicked the image should revert to its original state. | As expected |
| 4 | Using the application upload an image and adjust the contrast up and down, verify the changes to the image are correct, and reset the image. Expected: The contrast should increase and decrease, once the reset button is clicked the image should revert to its original state. | As expected |
| 5 | Using the application upload a coloured image and adjust the saturation up and down, verify the changes to the image are correct, and reset the image. Expected: The saturation should increase and decrease, once the reset button is clicked the image should revert to its original state. | As expected |
| 6 | Using the application upload an image and adjust the exposure up and down, verify the changes to the image are correct, and reset the image. Expected: The exposure should increase and decrease, once the reset button is clicked the image should revert to its original state. | As expected |
| 7 | Using the application upload an image and adjust the RGB ratio, verify the changes to the image are correct, and reset the image. Expected: The RGB ratio should adjust as expected, once the reset button is clicked the image should revert to its original state. | As expected |
| 8 | Using the application upload an image and introduce Gaussian blur, reset the image, introduce Uniform blur, reset the image. Expected: Blur should be introduced to the image and return to its original state when the reset button is selected. | As expected |
| 9 | Using the application upload an image with defined edges and carry out Sobel edge detection. Expected: The edges in the image should be highlighted and appear more prominent than before. | As expected |
| 10 | When choosing to save an image to the devices gallery, a pop-up message should appear when the process is finished. Expected: Expected pop-up appears. | As expected |
| 11 | Using whitebox, and blackbox testing methods, validate the input sanitization for image correlation patch-size works as expected. Expected: Pop-up should inform the user only numbers within the suggested range can be used. | As expected |
| 12 | Using whitebox, and blackbox testing methods, validate the input sanitization for image correlation range works as expected. Expected: Pop-up should inform the user only numbers within the suggested range can be used. | As expected |
| 13 | Attempt to use a large image for facial recognition. Expected: a pop-up message should inform the user the image must be 64x64. | As expected |

Because of the testing methods used, I have been able to verify that all the image processing features implemented work as expected, and also that the application navigation runs smoothly. Also, by stressing the application using large images, and reuploading multiple images allowed me to discover crashes caused by the allocated application memory overflowing.

As a result, I was able to implement code to scale down an image to a usable size and clear the application cache when user navigates to a new screen, meaning unnecessary code objects and image data would not be using unnecessary amounts of memory.

**Experiments**

This section outlines and discusses the various experiments carried out using the facial recognition algorithm implemented along with simulated environmental scenarios using features provided by the toolkit.

In experiments 1-5 the focus is on environmental factors that can be introduced into the user’s image to simulate a real-life scenario. The image below was used throughout for each experiment with the same 8x8 database.



1. The effect of increasing image brightness on the accuracy of correlation calculations.

Using the increase and decrease brightness features provided on the *ImageProperties* screen, I have set the user’s image to incremental brightness levels and carried out facial recognition on each setting. It should be noted that the patch-size was set to 10% and range was set to 3 pixels throughout. This is a basic real-life scenario where an image may be captured in low to high light levels, before carrying out facial recognition on a subject.

As seen from the results in Appendix BZ, as we increment through the increasing brightness levels the percentage accuracy of image correlation calculations increases until we reach the original brightness factor of 0, where the percentage accuracy reaches a plateau. I believe this occurs because at low light the correlation algorithm does not have a great deal of pixel value differentiation, therefore the levels of accuracy decrease at low light levels. It can also be seen in the image preview that at extremely low light levels the image is close to being unrecognisable to the human eye.

At each stage of this experiment the algorithm took approx. 9 seconds to execute, therefore the changes in light did influence the execution speed.

1. The effect of increasing image contrast on the accuracy of correlation calculations.

Using the increase and decrease contrast features provided on the *ImageProperties* screen, I have set the user’s image to incremental contrast levels and carried out facial recognition on each setting. It should be noted that the patch-size was set to 10% and range was set to 3 pixels throughout. This real-life scenario may be seen when a resulting image does not contain dark shadows which may or may not influence the recognisability of the subject.

As seen from the results in Appendix CA, from contrast factors -8 to 0 there is little to no negative or positive effect on the percentage accuracy of the correlation calculation. However, from contrast factor 0 onwards we see a consistent decrease in the percentage accuracy. This could be because as contrast levels increase more and more shadows are highlighted in the image than before, causing more pixel value differentiation as dark portions of the image become very dark.

At each stage of this experiment the algorithm took approx. 9 seconds to execute, therefore the changes in light did influence the execution speed.

1. The effect of increasing image exposure on the accuracy of correlation calculations.

Using the increase and decrease exposure features provided on the *ImageProperties* screen, I have set the user’s image to incremental exposure levels and carried out facial recognition on each setting. It should be noted that the patch-size was set to 10% and range was set to 3 pixels throughout. This real-life scenario may be seen when a resulting image is under or over-exposed.

As seen from the results in Appendix CB, from exposure factors -8 to 6 there is little to no negative or positive effect on the percentage accuracy of the correlation calculation. However, once we reach exposure factor 8 there is a drastic decrease in the percentage accuracy of the correlation calculation. This is because as we reach such exposure levels the image becomes over exposed causing the lightly shaded portions of the user’s image to be very white. Resulting in the image looking very different than the original state making facial features hard to match in comparison images.

At each stage of this experiment the algorithm took approx. 9 seconds to execute, therefore the changes in light did influence the execution speed.

1. The effect of increasing image noise on the accuracy of correlation calculations.

Using the add noise feature provided on the *Filter* screen, I have set the user’s image to incremental noise levels and carried out facial recognition on each setting. It should be noted that the patch-size was set to 10% and range was set to 3 pixels throughout. This real-life scenario may be seen when a resulting image is captured in a dark environment causing increased levels of noise.

As seen from the results in Appendix CC, as we increase the noise factor from 0 to 10, there is a steady decrease in the percentage accuracy of correlation calculations. This is because the effect of noise added a speckled effect to the user’s image, with a direct effect on the pixel values themselves which are used in the algorithm calculations. However, in a more general explanation increased noise makes key aspects such as facial features slightly more recognisable at each increment. It may be necessary to consider methods of de-noising an image in this environmental scenario before carrying out facial recognition, as you may experience less of a decrease in accuracy.

At each stage of this experiment the algorithm took approx. 9 seconds to execute, therefore the changes in light did influence the execution speed.

1. The effect of increasing image blur on the accuracy of correlation calculations.

Using the add Gaussian blur feature provided on the *Filter* screen, I have set the user’s image to incremental blur levels and carried out facial recognition on each setting. It should be noted that the patch-size was set to 10% and range was set to 3 pixels throughout. This real-life scenario may be seen when a resulting image is out of focus at the time of being captured, possibly a moving person.

As seen from the results in Appendix CD, as we increase the blur factor from 0 to 1, there is a slight increase in percentage accuracy. Perhaps this is because the blur feature smooths the image, making the image more comparable to those in the database. However, from blur factor 1 to 11 there is only a very marginal decrease in the percentage accuracy because of the image becoming more and more out of focus. From blur factor 12 onwards the accuracy percentage remains similar however it is no longer finding the correct match as at this point the image is very blurred an unrecognisable.

At each stage of this experiment the algorithm took approx. 9 seconds to execute, therefore the changes in light did influence the execution speed.

1. The effect of increasing patch-sizes on the accuracy of correlation calculations.

Using the user input field on the *Facial recognition* screen, I have set the patch-size (%) to different incrementing sizes and carried out facial recognition on each setting. It should be noted that the range was set to 3 pixels throughout. This experiment uses the same user image from all previous experiments with no image processing carried out pre-recognition.

From the results in Appendix CE, as the patch-size decreases from 9% to 3% there is a steady increase in the percentage accuracy of the correlation calculation, with patch-size 3% having the highest accuracy level of 90%.

However, it can also be seen that as the patch-size gets smaller the time to execute the algorithm increases, with a possible anomaly or processing bug at patch-size 5%. The reason for the increase in time to execute is because with smaller patches, there is an increase in the number of correlation calculations to cover the whole of the user’s image. Therefore, it is important to weigh up what factors are more important when carrying out facial recognition, the speed of the process, or the accuracy of the results. I would suggest the accuracy is more important, however it is necessary to find of means of keeping processing time low also.

1. The effect of increasing range sizes on the accuracy of correlation calculations.

Using the user input field on the *Facial recognition* screen, I have set the range of pixels to search to different incrementing amounts and carried out facial recognition on each setting. It should be noted that the patch-size was set to 10% throughout. This experiment uses the same user image from all previous experiments with no image processing carried out pre-recognition

From the results in Appendix CF, as the range of additional pixels searched in addition to the patch increases, there is an increase in the percentage accuracy of correlation calculations, with range value 11 reaching a percentage accuracy of 85%. However, at this range value it can also be seen that the time to execute is also much more at 45 seconds. This is because as we increase the range parameter used by the correlation algorithm, we are asking the algorithm to search through more pixel values and calculate the correlation between extended patches.

1. Finding the optimal patch-size and range combination to use.

It can be seen from experiments 6 and 7, that to find accurate results the image correlation algorithm should be used with a smaller patch-size, although not too small, and a larger range value. An accuracy level of 88.13% [1] was previously recorded, however during this experiment I have been able to achieve an accuracy of 92% as per the results shown in Appendix CG. Finding an optimal setting requires finding a small patch-size and a larger range, which together does not take considerably long to execute and also more importantly finds the correct result. The most accurate result of 92%, shows it took 290 seconds to compute. However, it may be more suitable to sacrifice 1% of accuracy and using a patch-size of 5% and a range of 10 to obtain an accuracy of 91% in 150 seconds.

System evaluation

Did I accomplish what I set out to do?

* An android application
* Investigate and learn basic image processing
* More advanced image processing
* Implementing facial recognition
* Experimental approach

Improvements and future advancements.

The process of setting goals, planning work and developing code. (Weekly meetings)

Project management

* Git
* Solution work plan

The quality of code

* Is it built for future development?
* Using Javadoc
* Consistent coding standards (camel case)
* Re-using variables/methods to avoid duplication

Conclusion

Briefly go over how you feel the project went, was it a success, reiterate your findings, what might improve the project further. Do not introduce additional content.

References

[0]"Project code repository", GitLab, 2018. [Online]. Available: <https://gitlab.eeecs.qub.ac.uk/40126559/CSC3002-Project.git>.

[1]"Unconstrained Face Recognition", Wjscheirer.com, 2018. [Online]. Available: <https://www.wjscheirer.com/projects/unconstrained-face/>.

[2] J. Klontz and A. Jain, A Case Study on Unconstrained Facial Recognition Using the Boston Marathon Bombings Suspects. Michigan State University, 2013.

[3] A.S. Georghiades, P.N. Belhumeur, and D.J. Kriegman, From few to many: illumination cone models for face recognition under variable lighting and pose, 2001.

[4]"Facing challenges in face recognition: one-to-one vs. one-to-many", Asmag.com, 2016. [Online]. Available: <https://www.asmag.com/showpost/21158.aspx>.

[5] Jack Gaston, Ji Ming and Danny Crookes, Unconstrained face identification with multi-scale block-based correlation. Queen’s University Belfast, 2017.

[6] Ian T. Young, Jan J. Gerbrands and Lucas J. van Vliet, Fundamentals of Image Processing, 2007.

[7] Android development using Bitmap objects, Developer.Android.com. [Online]. Available: <https://developer.android.com/reference/android/graphics/Bitmap.html>.

[8] C. Anderson, LFW crop dataset. [Online]. Available: <http://conradsanderson.id.au/lfwcrop/>

[9] UMASS, LFW dataset. [Online]. Available: <http://vis-www.cs.umass.edu/lfw/>

[10]"BMP files", Paulbourke.net, 2018. [Online]. Available: <http://paulbourke.net/dataformats/bmp/>.

[11]"the definition of image processing", Dictionary.com, 2018. [Online]. Available: <http://www.dictionary.com/browse/image-processing?s=t>.

[12]"Image Processing Algorithms Part 5: Contrast Adjustment - Dreamland Fantasy Studios", Dreamland Fantasy Studios, 2018. [Online]. Available: <http://www.dfstudios.co.uk/articles/programming/image-programming-algorithms/image-processing-algorithms-part-5-contrast-adjustment/>.

[13]"Camera Exposure: Aperture, ISO & Shutter Speed", Cambridgeincolour.com, 2018. [Online]. Available: <https://www.cambridgeincolour.com/tutorials/camera-exposure.htm>.

[14]"Math behind colorspace conversions, RGB-HSL – Niwa", Niwa.nu, 2018. [Online]. Available: <http://www.niwa.nu/2013/05/math-behind-colorspace-conversions-rgb-hsl/>.

[15]"Matrix Operations for Image Processing", Graficaobscura.com, 2018. [Online]. Available: <http://www.graficaobscura.com/matrix/index.html>.

[16]"Image Processing By Interp and Extrapolation", Graficaobscura.com, 2018. [Online]. Available: <http://www.graficaobscura.com/interp/index.html>.

[17]"Java DIP - GrayScale Conversion", www.tutorialspoint.com, 2018. [Online]. Available: <https://www.tutorialspoint.com/java_dip/grayscale_conversion.htm>.

[18]"ISO and Image Noise", Digital SLR Guide, 2018. [Online]. Available: <http://www.digital-slr-guide.com/iso-and-image-noise.html>.

[19]"Adding random noise to an image. - CodeProject", Codeproject.com, 2018. [Online]. Available: <https://www.codeproject.com/Questions/98491/Adding-random-noise-to-an-image>.

[20]"Image Kernels explained visually", Explained Visually, 2018. [Online]. Available: <http://setosa.io/ev/image-kernels/>.

[21]"Kernel (image processing)", En.wikipedia.org, 2018. [Online]. Available: <https://en.wikipedia.org/wiki/Kernel_(image_processing)>.

[22]"Gaussian blur", En.wikipedia.org, 2018. [Online]. Available: <https://en.wikipedia.org/wiki/Gaussian_blur>.

[23] Web.pdx.edu, 2018. [Online]. Available: <http://web.pdx.edu/~jduh/courses/Archive/geog481w07/Students/Ludwig_ImageConvolution.pdf>.

[24]"Sobel operator", En.wikipedia.org, 2018. [Online]. Available: <https://en.wikipedia.org/wiki/Sobel_operator>.

[25]"Dashboards | Android Developers", Developer.android.com, 2018. [Online]. Available: <https://developer.android.com/about/dashboards/index.html>.

[26]"User Interface Design Basics | Usability.gov", Usability.gov, 2018. [Online]. Available: <https://www.usability.gov/what-and-why/user-interface-design.html>.

[27] A. Affairs, "User Interface Elements | Usability.gov", Usability.gov, 2018. [Online]. Available: <https://www.usability.gov/how-to-and-tools/methods/user-interface-elements.html>.

[28]"Layouts | Android Developers", Developer.android.com, 2018. [Online]. Available: <https://developer.android.com/guide/topics/ui/declaring-layout.html>.

[29]"Intents and Intent Filters | Android Developers", Developer.android.com, 2018. [Online]. Available: <https://developer.android.com/guide/components/intents-filters.html>.

[30]"Quick Guide to Using Accessors and Mutators in Java", ThoughtCo, 2018. [Online]. Available: <https://www.thoughtco.com/accessors-and-mutators-2034335>.

[31]"Chapter 10. Documentation with Javadoc", Drjava.org, 2018. [Online]. Available: <http://drjava.org/docs/user/ch10.html>.

[32]"CSC3002 Image Processing Git, Javadocs", GitLab, 2018. [Online]. Available: <https://gitlab.eeecs.qub.ac.uk/40126559/CSC3002-Project/tree/master/Javadoc/com/example/bande/csc3002app>.

[33]"Building Effective Unit Tests | Android Developers", Developer.android.com, 2018. [Online]. Available: <https://developer.android.com/training/testing/unit-testing/index.html>.

[34]"Automating User Interface Tests | Android Developers", Developer.android.com, 2018. [Online]. Available: <https://developer.android.com/training/testing/ui-testing/index.html>.

[35]"CSC3002 Image Processing Git, Automated testing suite", GitLab, 2018. [Online]. Available: <https://gitlab.eeecs.qub.ac.uk/40126559/CSC3002-Project/tree/master/Application/app/src/androidTest/java/com/example/bande/csc3002app>.

[36]"ActivityTestRule | Android Developers", Developer.android.com, 2018. [Online]. Available: <https://developer.android.com/reference/android/support/test/rule/ActivityTestRule.html>.

Appendix:

|  |  |
| --- | --- |
| AA.  The application start-up screen. | AB.  The application main-screen. |
| AC.  The application adjust-image screen. | AD.  The application image-properties screen. |
| AE.  The application RGB screen (colour ratio). | AF.  The application image-filter screen. |
| AG.  The application facial recognition screen. | AH.  The application facial recognition results screen. |
| AI.  onClickUploadImage() method used in StartupScreen.java to manage button clicks. | |
| AK.  onActivityResult() method used in StartupScreen.java to provide functionality for uploading or capturing images from the user’s device for use in the application. | |
| AL.  getMaintainedImage() method used in StartupScreen.java to scale down the user’s image if necessary to prevent a crash. | |
| AM.  onClickCaptureImage() method used in StartupScreen.java to manage button clicks. | |
| AN.  A sub-section of the onActivityResult() method used in StartupScreen.java to provide functionality for capturing images from the user’s device camera for use in the application. | |
| AO.  onClickSaveToDevice() method which listens for the users desire to save their image to the device, then carries out the necessary steps to do so. | |
| AP.  read() method in the Image.java class used to initialise the image object by organising the image data into allocated memory. | |
| AQ.  setPixel() method uses given colour channel data arrays and indices to modify the look of an image. | |
| AR.  onClickDecreaseBrightness() and onClickIncreaseBrightness() methods are used to manage the user interaction with buttons. Facilitating the user to increase or decrease the brightness of their image. | |
| AS.  adjustBrightness() method uses an algorithm to iterate through the data of the image belonging to the user, resulting in a modified image which has either increased or decreased in brightness. | |
| AT.  onClickDecreaseContrast() and onClickIncreaseContrast() methods are used to manage user interaction with buttons. Facilitating the user to increase or decrease the contrast of their image.    https://i.gyazo.com/6c6ac89fde163d1d81261ce475f9b589.png | |
| AU.  adjustContrast() method uses an algorithm described in section 3.3 to iterate through the data of the image belonging to the user, returning a modified image which has either increased or decreased in contrast. | |
| AV.  onClickDecreaseSaturation() and onClickIncreaseSaturation() methods are used to manage user interaction with buttons. Facilitating the user to increase or decrease the saturation of their image.    https://i.gyazo.com/28d0261bc0db1266c25514a4c3cf86b1.png | |
| AW.  adjustSaturation() method uses an algorithm described in section 3.3 to iterate through the data of the image belonging to the user, returning a modified image which has increased or decreased in saturation. | |
| AX.  onClickDecreaseExposure() and onClickIncreaseExposure() methods are used to manage user interaction with buttons. Facilitating the user to increase or decrease the exposure of their image. | |
| AY.  adjustExposure() method uses an algorithm described in section 3.3 to iterate through the data of the image belonging to the user, returning a modified image which has increased or decreased in exposure levels. | |
| AZ.  onClickFilterNoise() method is used to manage user interaction with buttons. Facilitating the user to increase the amount of noise in their image. | |
| BA.  addNoise() method uses an algorithm described in section 3.3 to iterate through the data of the image belonging to the user, returning a modified image which has increased levels of noise. | |
| BB.  onClickFilterBlurGauss() and onClickFilterBlurUniform() methods are used to manage user interaction with buttons. Facilitating the user to increase the amount of blur in their image. | |
| BC.  gaussianKernel() is a method used by the addBlur() method to retrieve a 2D array representing the kernel used when adding Gaussian blur. | |
| BD.  uniformKernel() is a method used by the addBlur() method to retrieve a 2D array representing the kernel used when adding Uniform blur. | |
| BE.  addBlur() method uses an algorithm described in section 3.3 to manipulate the user’s image data to result in added levels of blurring. | |
| BF.  onClickFilterEdge() method is used to manage user interaction with buttons. Facilitating the user to carry out edge detection on their image. | |
| BG.  sobelKernelGx() and sobelKernelGy() are methods used by the edgeDection() method to retrieve the necessary kernel filters to complete edge detection on the user’s image. | |
| BH.  edgeDetection() method uses an algorithm described in section 3.3 to carry out edge detection on the user’s image.    (Algorithm continued on next page.) | |
| BI.  onClickFilterGreyscale() method is used to manage user interaction with buttons. Facilitating the user to convert their image to black and white. | |
| BJ.  convertToGreyscale() method uses an algorithm described in section 3.3 to assist the user in converting their image to black and white. | |
| BK.  onClickDecreaseRed() and onClickIncreaseRed() methods are button listener methods used to control the user’s interaction with these user interface elements. Result in an increased amount of image red.  https://i.gyazo.com/8593c5e5ddab06ae461164e40567f929.png | |
| BL.  onClickDecreaseGreen() and onClickIncreaseGreen() methods are button listener methods used to control the user’s interaction with these user interface elements. Result in an increased amount of image green. | |
| BM.  onClickDecreaseBlue() and onClickIncreaseBlue() methods are button listener methods used to control the user’s interaction with these user interface elements. Result in an increased amount of image blue. | |
| BN.  adjustRGB() method makes use of an algorithm which iterates through pixels of the user’s image adjusting the colour channel ratio of the image. | |
| BO.  The assets folder of the application. Stores the training images for multiple personnel used in calculating image patch correlation. An 8x8 image database. | |
| BP.  loadDatabase() method uses the applications asset manager to populate an array persons and how many training images they have. This information is used to load the image data itself. | |
| BQ.  getPersonTrainingImages() returns a 4D array of a person’s training image data (i.e. pixel values). | |
| BR.  checkUserInputValue() and checkUserInputIsInteger() are input validation methods to ensure the information provided by the user matches the necessary criteria. | |
| BS.  onClickCorrelation() is a button listener method used to control the users button interaction on this screen. Once clicked it validates the user input, executes the image correlation algorithm and differentiates the person index to retrieve the matched persons actual name, before calculating the persons reconstruction image and passing the information to the next activity. | |
| BT.  ImageCorrelation.java class instructor used to initialise variables used in face recognition algorithm. | |
| BU.  calculateImageCorrelation() method is the facial recognition algorithm. It initialises a patch, which is analysed for each training image belonging to every person in the database. The person whose image patches have the most correlation to the users own image is found to be the match. A reconstruction is then generated, which is the highest scoring patches belonging to the matched person, merged together into a new image.    (\*\*\*Continued next page.)    (\*\*\*Continued next page.) | |
| BV.  preCalculation() method is used to initialise calculations used in the image patch correlation calculation such as the sum of all patches, the sum of all patches squared and the sigma. | |
| BW.  Three methods used in calculating the matched persons reconstruction image. Accumulates the highest scoring for this person across all their training images and combines them into a new image. | |
| BX.  getMatchedPersonIndex() method is used to differentiate between the scores and find the highest scoring person as the match for the user’s image. Returns the index of the matched person which can be used to find out their name etc. | |
| BY.  A portion of the correlation algorithm which calculates the score between the current training patch and the user’s patch, saving the score if it is the highest recorded. | |
| BZ.  Experiment: The effect of increasing image brightness on the accuracy of correlation calculations. | |
| CA.  Experiment: The effect of increasing image contrast on the accuracy of correlation calculations. | |
| CB.  Experiment: The effect of increasing image exposure on the accuracy of correlation calculations. | |
| CC.  Experiment: The effect of increasing image noise on the accuracy of correlation calculations. | |
| CD.  Experiment: The effect of increasing image blur on the accuracy of correlation calculations. | |
| CE.  Experiment: The effect of increasing patch-size on the accuracy of correlation calculations. | |
| CF.  Experiment: The effect of increasing algorithm range on the accuracy of correlation calculations. | |
| CG.  Experiment: Finding the best combination of patch-size and range algorithm parameters to achieve optimal facial recognition accuracy. | |

CH.

The solution work plan as a development guide to ensure that the system requirements are met, and the development process runs smoothly.

|  |  |  |
| --- | --- | --- |
| Completion date | Task | Description |
| 25th October | Draft proposal & requirements | After researching the problem draft a solution with initial requirements for review in the weekly meeting. |
| 1st November | Initial application for development | Agree on a development language and platform to develop for (e.g. Android). Create an initial application to begin developing requirements. |
|  | Update draft proposal & requirements | From feedback received from the last meeting update the proposed solution. |
| 8th November | GUI navigation | Implement a means of navigation for the applications screens. Use this task to help get more familiar with android development. |
| 15th November | Capture an image | Develop a means of capturing an image for use using the devices build-in camera. |
|  | Upload an image | Develop a means of uploading an image for use from the devices storage. |
|  | Preview an image | Develop a GUI screen allowing the user to preview the image they have captured/uploaded. |
| 22nd November | Image toolkit | Implement the classes required to load an image and find out more information about it. E.g. resolution. |
| 29th November | Basic image processing | Implement an algorithm which completes basic image processing on an image. Changes to the image should be reflected in the image preview. |
| 20th December | Changing brightness, contrast, RGB ratios. | Develop algorithms that allow the user to change image brightness, contrast and RGB ratios. These changes should be reflected in the image preview. |
| 10th January | Adding/removing image noise. | Develop an algorithm which allows the user to add or remove noise from their image. |
|  | Adding image filters. | Develop a means of the user adding a pre-defined filter to their image. |
| 24th January | Basic face detection | Complete research and develop a basic face detection algorithm to build on in later tasks. |
| 7th February | Face detection | Build on the previous basic face detection algorithm so that it is suitable for use with face recognition. |
| 28th February | Basic face recognition | Research and develop a basic face recognition algorithm to build on in later tasks. |
| 14th March | Face recognition | Build on the previous basic face recognition algorithm so that it is suitable for use with face recognition. |
| 8th April | Final submission | Application code refinements and bug fixing so that the application is ready for final submission on 8th April. |