Abstract

This dissertation proposes a framework of techniques that can be used for virtual face mirroring. Virtual face mirroring is the process of mapping a series of actions from a series of human face images to a 3-D virtual character, so that the virtual face is driven to perform the same actions as those of the human face. We choose Candide-3 wireframe model to be our virtual character.

The mirroring process is divided into three major steps: face detection, facial feature extraction and driving animation. The face region of an input image is detected using Viola-Jones classifier. Then the face region is fitted to Active Shape Model to determine the positions of facial feature points. Finally we convert the driving animation problem to minimising a distance measurement between face image and virtual face model, which is a non-linear least squares problem and can be solved by the Levenberg-Marquardt algorithm. The solution is a set of optimised values of the action units that animates the face model, and is directly used to drive the animation.

We build a software program to facilitate the mirroring process. The program basically meets the goal of real-time mirroring, but the performance is only acceptable. In this dissertation we will discuss in detail the technical background of our problem and the results of our software program. This work can potentially suggest new modes of Human-Computer Interaction.

Keywords: Facial animation, Active Shape Model, Candide-3, Leveberg-Marquardt algorithm

Acknowledgements

I would like to acknowledge all the guidance from my advisor, Mr Colin Dalton, who has always been very positive and encouraged me to discover and make decisions myself. I am also grateful for the advice from PhD student Mr Alexander Davies, who introduced me, a stranger, into the world of computer vision.

I can not thank my parents and friends back in China enough. This is only possible because of every little bit of their warm support. And finally my acknowledgements to my fellow student and ex-girlfriend, Miss Lu Yinyin, who made my life abroad much easier and more enjoyable.

Contents

1 Introduction

1-1 The Aim of Virtual Face Mirroring

Virtual face mirroring is the process of mapping a series of actions from a human face to a virtual character (e.g. a polygonal mesh), to drive the virtual character to perform in the same way as the human face does. We can acquire face images from a webcam and carry out the mapping in real time, so that a virtual face animation can be constructed synchronously. During this process, facial action is tracked, interpreted, and finally used to drive the character according to such a rule that the resemblance of virtual face is satisfactory. Such an application may be particularly of interest in the field of Human-Computer Interaction, and can potentially be applied in games, remote communication, etc.

The aim of our project is to study and experiment with related techniques, propose a framework of techniques that can be utilised to achieve real-time face mirroring, and finally build a practical software system for virtual face mirroring. The program will integrate techniques of a wide range, including facial recognition and 3-D modelling and animation. Efficiency of the program will also be a priority. Finally the result of the program is evaluated, to verify the feasibility and effect of our framework.

1-2 The Mirroring Process

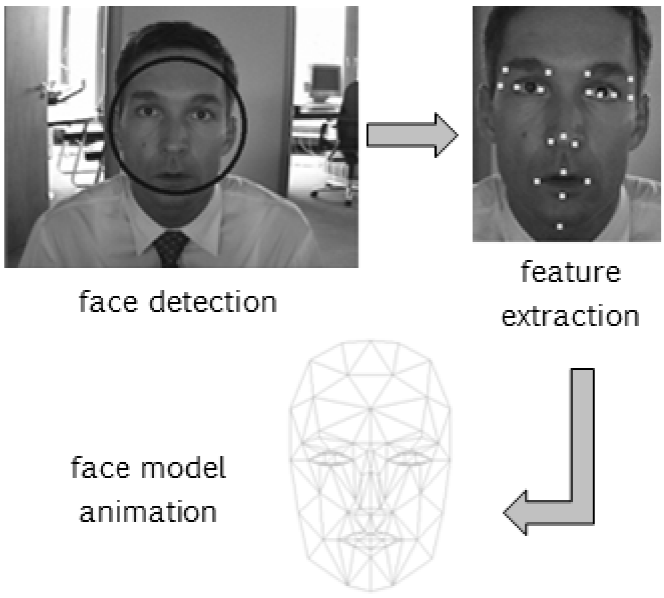


Figure 1 The mirroring process

In this project, we divide the whole process into three major steps (Figure 1):

- Face detection. The region of the image where a face appears is discovered.

- Facial feature extraction. We further discover where the facial features are in the image. By feature, we mean a point on the face that contains some important information, such as corners of mouth.
- Driving animation. The feature knowledge should be interpreted into a certain form in order to be able to control the virtual face.

Besides, some pre-processing of the face image is also necessary such as blurring, as in most circumstances images acquired from webcam are subject to noise.

1-3 Desirable Properties

Pantic et al. [6] did a synthetic research of different approaches on automatic analysis of facial expressions. They proposed "an ideal system for facial expression analysis", in which they listed a number of functionalities such an ideal system should support and properties it should have. Although the final aim of the article differs from ours, these functionalities and properties can be viewed as basic design factors that should be considered for our system. These include "Automatic facial image acquisition", "Subjects of any age, ethnicity and outlook", "No special markers/make-up required", etc.

Of the functionalities proposed in the ideal system, those responsible for face detection usually handle a certain variation, and some of them are external (not affected by facial performance) such as lighting. The ideal system should also deal with "partially occluded faces" and "rigid head motions", and in addition, require no special markers.

These are all factors affecting robustness of the system and user experience. We may want to consider complicating the system to support some or all of these specifications, but we must start from the simplest scenarios. And also some of these factors may not affect our aim significantly (e.g. occluded faces, as serious users can be expected to not cover their face when they want to see how the virtual character follows them) so we will not take them into account. What may really make a difference for us is variation in lighting.

Another property that we should pay attention to is "real-time". We need to generate facial animation synchronously as face images are captured, so algorithms requiring smaller amount of calculation may be preferable. And when programming we can also apply some technique to reduce waiting time between frames.

1-4 Comparisons to Some Related Works

Our work is related to other problems such as face recognition [1] and facial expression classification [2]. The aim of face recognition is to identify an individual from his / her face image, so it needs knowledge about the particular individual beforehand. In our aim, however, the identity of the user is irrelevant.

Facial expression classification differs from our work in the usage of facial knowledge obtained from image. Facial expression classification focuses on understanding what expression is shown in a particular pattern of facial feature. It aims to analyse and define different types of expression, as well as to build methods to classify a particular image into the known types. In our task, the next step after facial knowledge is to mirror it onto a virtual face, and we will not necessarily classify facial actions into types.

The remaining parts of this dissertation will be arranged as follows: Sections 2~4 will present some available solutions to the three major steps respectively, and discuss on the techniques we will choose. Section 5 will give details about our software implementation of chosen techniques and the design of mirroring program, and will also demonstrate and discuss the mirroring results. The dissertation will be concluded in Section 6.

2 Face Detection

Pantic et al [6] divide approaches of facial detection into two types: holistic and analytic. In holistic approaches a face is perceived as a whole (this is also the way humans recognise faces [7]). In contrast, analytic approaches first examine individual features (eyes, mouth, etc). The relationship between detected features then decides the position of the whole face. We will focus on holistic approaches as they are simple, so that we do not need to consider many factors at the same time.

In this section we first look into Principal Components Analysis, a useful mathematical technique to reduce dimension of a linear problem space. It has a direct application in face detection, that is, the eigenface method, but is also used in feature extraction techniques we will discuss later.

We then introduce another widely-used detection algorithm, the Viola-Jones classifier.

2-1 Principal Components Analysis

Principal Components Analysis is a linear algebra method to approximate a higher dimension vector space with one with a lower dimension, so that a problem in the higher dimension space can be scaled down and solved more easily. In applications, a set of data of a specific kind (problem set, such as facial images) generally show some common characters. This means that large as the problem space is, the particular points in which we may be interested are likely to cluster in a linear subspace of the problem space, thus can be approximated without losing much information.

PCA can be used in both face detection and feature extraction. Gong et al. [3] gives an example for face model. If we represent a set of M face images of the same dimension as M N-dimensional vectors (for example, intensities of all pixels) $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_\mathrm{M}$, we have their mean and covariance matrix:

$$\boldsymbol{\mu} = \frac{1}{\mathrm{M}} \sum_{\mathrm{i}=1}^{\mathrm{M}} \mathbf{x}_\mathrm{i}$$

$$\mathbf{C} = \frac{1}{\mathrm{M}} \sum_{\mathrm{i}=1}^{\mathrm{M}} (\mathbf{x}_\mathrm{i} - \boldsymbol{\mu})(\mathbf{x}_\mathrm{i} - \boldsymbol{\mu})^\mathrm{T}$$

Let $\lambda_1, \lambda_2, \dots, \lambda_\mathrm{K}$ be the K $(\mathrm{K} \leq \mathrm{N})$ largest eigenvalues of $\mathbf{C}$ and $\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_\mathrm{K}$ be the corresponding eigenvectors. Apply a linear transformation $\boldsymbol{\alpha} = \mathbf{U}^\mathrm{T}(\mathbf{x} - \boldsymbol{\mu})$, where $\mathbf{U} = [\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_\mathrm{K}]$, then an N-dimensional vector $\mathbf{x}$ is mapped to a K-dimensional vector $\boldsymbol{\alpha}$. The original vector is approximated by

$$\tilde{\mathbf{x}} = \sum_{i=1}^{K} \alpha_i \mathbf{u}_i + \boldsymbol{\mu}$$

where α_i is the i-th component of $\boldsymbol{\alpha}$. This is a linear composition of all eigenvectors.

If we project all the original M vectors to get $\boldsymbol{\alpha}_1, \boldsymbol{\alpha}_2, \ldots, \boldsymbol{\alpha}_M$, this set of vectors has scatter $\mathbf{U}^T\mathbf{C}\mathbf{U}$. By choosing to use $\mathbf{U}$, PCA maximises the determinant $|\mathbf{U}^T\mathbf{C}\mathbf{U}|$ and thus retains most variance.

We have done a simple demonstration on 14 images (Figure 2). The images are scaled down to a rather small resolution because we can not allocate enough memory in the program, and thus we choose landscape photos rather than face images as the former are not as detailed. Figure 3 shows the mean image and first 7 eigenvectors. Figure 4 compares an image and the approximated image built from the projected values.

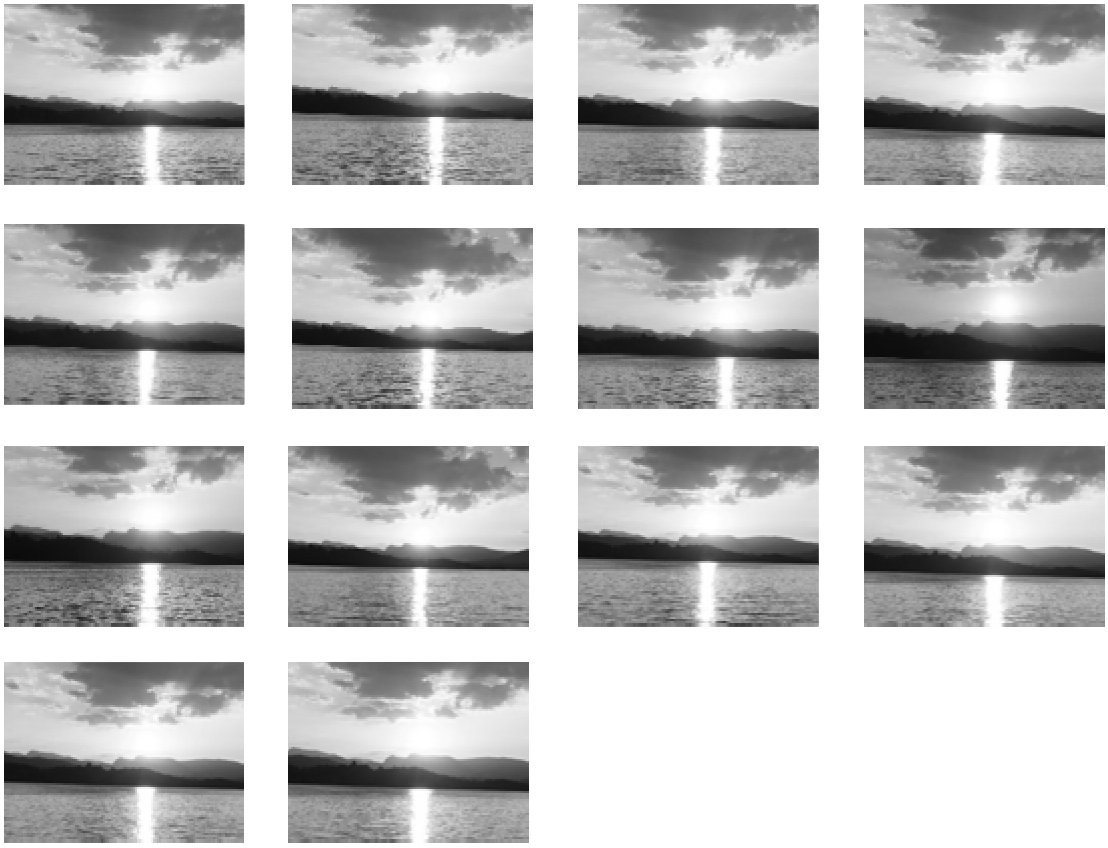


Figure 2 Input images of PCA

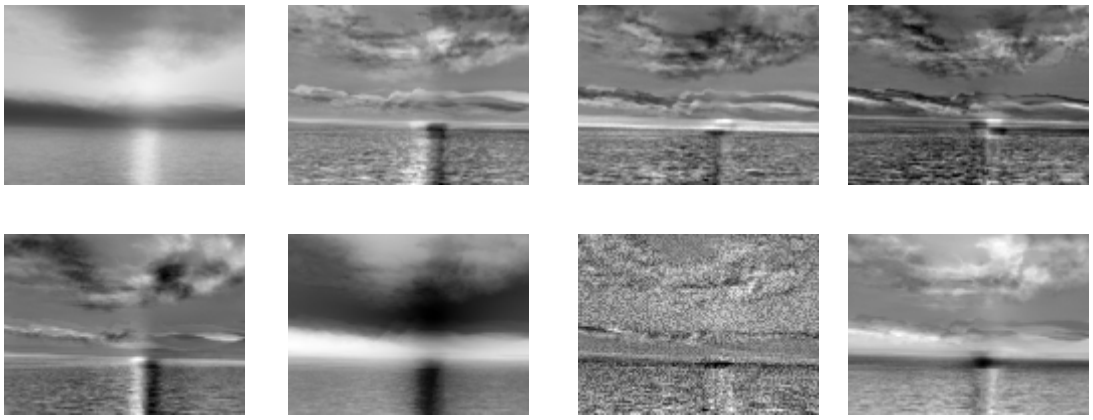


Figure 3 Mean image (the first) and the first seven eigenvectors

2-2 Eigenface

Figure 4 Original image (left) and projected image (right)

Figure 5 Projection result for an image that does not belong to the original class

Eigenface [4] is an application of PCA in face detection by Turk and Pentland. Similar as in our previous demonstration, they apply PCA on a set of training face images and get the projected subspace, called the "face space".

When detecting, an image first is projected onto the face space, resulting in an approximated image of the original image. There detection criteria is based on the fact that non-face images change significantly when projected onto the face space (we give an example using our landscape image set in 2-1), whilst images that contain face do not (as demonstrated in Figure 4). Taking a pixel, a region of image around can be projected to the face space, and the projection is measured by the distance between original and projected images, which is recorded as "faceness" of the pixel. Apply this process for each pixel, and find a pixel where "faceness" obtains minimum value. So this will be considered centre of face.

Turk and Pentland made the above discussion on the assumption that the input image is sure to contain a face. We would like to add that if this is not the case, it will be most likely that

for every pixel the "faceness" is quite large. Therefore, the minimum "faceness" can be compared to a threshold to determine whether it is really a face.

However, we have not favoured eigenface as our choice. One reason is that the aim of Turk and Pentland is recognition, or in other words, identification, and the above discussion on face detection only serves as a prerequisite of recognition. So they did not give much information on how the detection worked in their experiment. Another reason is that as mentioned in section 2-1, we need to keep the scale of data low when doing PCA†, which means quite a lot of details provided by the face image may have to be abandoned.

2-3 Viola-Jones classifier

Our choice for face detection is the widely-used Viola-Jones classifier [5].

Viola-Jones classifier detects objects based on simple features of the image. Three kinds of features are used (Figure 6). A two-rectangle feature is constructed at two horizontally or vertically adjacent rectangles that have the same size and shape. The value of this feature is defined as the difference between the sum of all pixels in one rectangle and that in the other. Similarly, a three-rectangle feature has three horizontally adjacent rectangles, the difference between the sum of all pixels in the centre rectangle and that in the other two is used. A four-rectangle feature is has four rectangles in a 2×2 grid, and the difference between the sum of all pixels in one diagonal pair of rectangles and that in the other pair is used.

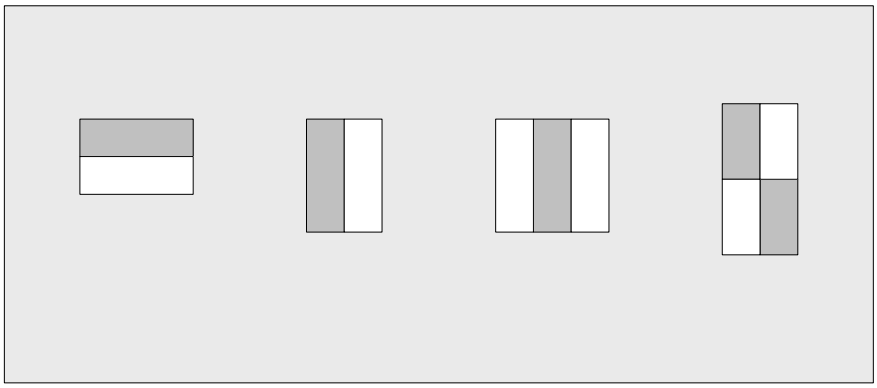


Figure 6 The three kinds of features

Firstly, the method needs a training set of both positive and negative images. It uses AdaBoost to determine what particular features best describe an object class. AdaBoost is a method to boost performance of a learning algorithm by combining several weak classifiers to form a strong classifier. In this case, weak classifiers are simply comparison between a

† We verified PCA using the function provided by OpenCV [18]. Data from all input images should be merged into one data structure before it can be processed. If we use 50 images with resolution of 160×120 (and the resolution of webcam is 640×480), they will require almost one million bytes.

single feature and a threshold value, which gives a Boolean output. The weak classifiers are trained and their error rates evaluated, and very few (say only 2) of them that show lowest error rates are selected to form a strong classifier.

The characteristics of this result classifier is that it has high detection rate but low rejection rate, which means that it rarely takes a right object wrong, but is likely to take a wrong object right (a high “false positive” rate). In order to reduce the false positive rate, the method then applies AdaBoost again to boost a set of n such classifiers. The result is a rejection cascade, in which the classifiers are ordered from the simplest to the most complex.

When detecting, all sub-windows of an image is passed to the first classifier in the cascade. If a sub-window is accepted, it will be passed to the second classifier, otherwise it will be instantly rejected. The second classifier then passes all the sub-windows it accepts to the third classifier, and so on. Only when a sub-window is accepted by the last classifier (thus accepted by all) will it be accepted by the cascade. Thus the possibility of false positive is significantly lowered.

In practice, the first several classifiers are capable of rejecting most sub-windows. As these classifiers are the simplest, they effectively reduced workload for the later, more complex classifiers. Consequently Viola-Jones classifier is quite efficient and fast.

OpenCV has implemented Viola-Jones classifier, mainly for the popular demand of face detection and the efficiency of this algorithm. Classifiers can still be trained to detect other rigid objects.

3 Facial Feature Extraction

Yang et al [8] classify related methods into four types: knowledge-based, feature-based, template-based and appearance-based. Facial feature is not the only way to be the medium between face detection and animation, but we choose this because they are straightforward. We can directly get positions of all facial features, which contain most of the information we may want to know about facial actions. And they can be easily interpreted into animation data.

When a face is detected, the next step is to interpret the information it holds. We have focused on two methods: Active Appearance Model and Active Shape Model. Both methods are linear in that they use Principal Components Analysis, which we believe makes them basic and simple. We will give introduction of the two methods, and compare them to make our choice.

3-1 Active Appearance Model

Active Appearance Model [9] is widely used in situations such as object tracking, facial expression recognition and medical image analysis **[**10]. Active Appearance Model makes use of variations in not only shape (coordinates of feature points), but also texture (in this case, intensities of pixels within the feature shape). Models are built based on Principal Component Analysis. A model learned from a set of training images is then used for matching and describing new images, finding appropriate parameter values by an approximated optimisation procedure. As is introduced in [9] and **[**10], the process of building and fitting are roughly as follows.

Facial features in training images are first labelled. Images are analysed to build a statistical shape model, a mean shape generated by aligning feature points between different images.

Figure 7 Feature points (left) and shape-free patch(right)

This figure is quoted from reference [10].

Then each image is transformed, so that its feature points are fitted to those of the mean shape. The result of is called a "shape-free patch." (Figure 7) From the texture data in the shape-free patch, a texture vector $\mathbf{g}$ can be constructed and then normalised.

By applying Principal Component Analysis, the relation between shape and texture of a training image and those of the mean shape can be written as

$$\mathbf{x} = \bar{\mathbf{x}} + \mathbf{Q}_s\mathbf{c}$$
$$\mathbf{g} = \bar{\mathbf{g}} + \mathbf{Q}_g\mathbf{c}$$

where $\mathbf{x}$ and $\bar{\mathbf{x}}$ are respectively shape vectors (merging x- and y-coordinates of all feature points into a single row vector) of the training image and the mean shape, $\bar{\mathbf{g}}$ is the mean texture (texture vector of shape-free patch), $\mathbf{c}$ is a vector of appearance parameters, which controls both shape and texture, and $\mathbf{Q}_s$ and $\mathbf{Q}_g$ are matrices defining the modes of variation derived from training set.

An image can be approximated from $\mathbf{c}$. To achieve this, a shape is generated by transforming $\mathbf{x}$, and a texture is generated (for the shape-free patch) by transforming $\mathbf{g}$. The texture is then warped to fit to the shape. Two others parameter vectors $\mathbf{t}$ and $\mathbf{u}$ are used to control these transformations. Thus, all model parameters are denoted as a vector $\mathbf{p} = \begin{bmatrix} \mathbf{c} \\ \mathbf{t} \\ \mathbf{u} \end{bmatrix}$.

When using the trained model to describe a new image, its texture is first sampled and projected to the texture model space, resulting in a texture vector $\mathbf{g}_c$. The difference between the image and the model (whose texture vector is denoted by $\mathbf{g}_m$) is defined as

$$\mathbf{r}(\mathbf{p}) = \mathbf{g}_c - \mathbf{g}_m$$

and error is evaluated by

$$E = |\mathbf{r}(\mathbf{p})|^2.$$

The relationship between the step $\Delta\mathbf{p}$ and $\mathbf{r}(\mathbf{p})$ is given as

$$\Delta\mathbf{p} = -\mathbf{R} \cdot \mathbf{r}(\mathbf{p})$$

where $\mathbf{R}$ is the gradient matrix. Active Appearance Model assumes $\mathbf{R}$ to be fixed, and pre-computes it for convenience. Then we aim to find the appropriate value of parameters in $\mathbf{p}$ to minimise E, therefore optimising the fitting. This is done iteratively.

Each time, a predicted value for $\Delta\mathbf{p}$ is calculated, and then applied to $\mathbf{p}$. As $\Delta\mathbf{p}$ is not a precise value, we use an adjustable scalar k, initially set to 1, to control how much of $\Delta\mathbf{p}$ should be applied. Thus we have a new value of error $E' = |\mathbf{r}(\mathbf{p} + k\Delta\mathbf{p})|^2$, which is then compared with the old value of E. According to the result, we evaluate the quality of convergence and decide whether to accept the change to $\mathbf{p}$ (so that the new value will be $\mathbf{p} + \Delta\mathbf{p}$), or to adjust k and re-evaluate.

When no more improvement is attained, the image is described with the appearance model by the optimised value of $\mathbf{p}$. We can calculate features, such as shape $\mathbf{x}$ and texture $\mathbf{g}$, from $\mathbf{p}$.

3-2 Active Shape Model

Another method, Active Shape Model [11] [3] [20], is associated with Active Appearance Model. Similarly, the training of the model requires a set of images with their feature points, or "landmarks", marked manually. And PCA is also applied during the training.

An Active Shape Model consists of a shape model and a set of profile models, the same number as landmarks. The shape model describes the pattern of all landmarks, built using PCA. And each profile model is built on small areas of the image near the corresponding landmark, describing how contents near the landmark are expected to look. They can be seen as templates. For example, based on the shape, at every landmark we can build the normal (perpendicular to the local direction of shape edge) and choose several points on the normal that are adjacent to the landmark. The intensity values of these points (including the landmark itself) can be the template.

When a new image is matched to the model, a start shape is first constructed from the rough position of the face (this can be the output of face detection). We first move all landmarks independently in their neighbourhoods and find a position where the local profile best matches the template of the corresponding profile model. Then we analyse the overall shape of all landmarks and adjust it to best fit the shape model. We iteratively operate this process (each iteration includes profile matching and shape matching, but not a series of iterations of profile matching followed by a series of iterations of shape matching) until convergence (Figure 8). The result is the final optimised positions of all landmarks.

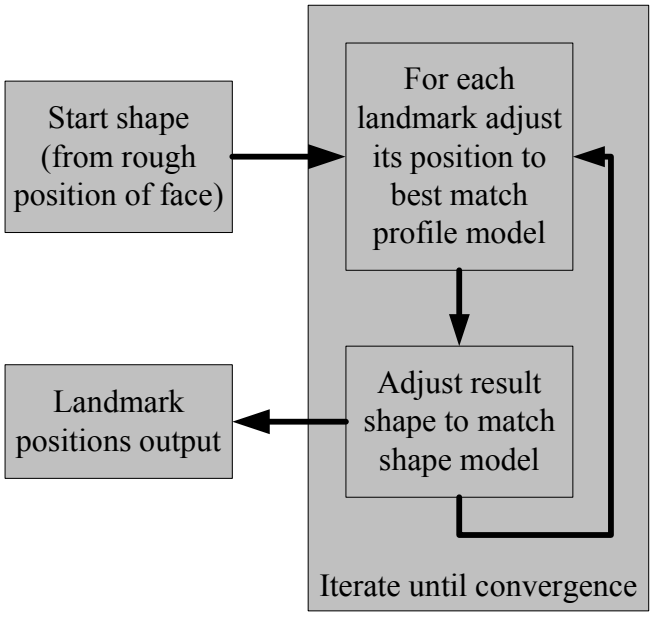


Figure 8 Active Shape Model matching process

3-3 Comparison of AAM and ASM

As the name suggests, Active Shape Model focuses on the shape of facial features. And it does not make good use of all available information provided by the face image (only intensities near the landmarks are used). According to Yan et al. [12], who combined ASM and AAM in their work, this property of ASM leads to fast convergence, but also means that the result of ASM are often suboptimal.

AAM, on the other hand, makes use of information obtained from all points within the shape model, and thus AAM may use fewer feature points than ASM. According to Cootes et al. [7], AAM tends to be more robust than ASM search alone. But as AAM involves warping of the shape and interpolation between feature points, the amount of calculation is much larger than ASM. Another problem as addressed by Yan et al. is that AAM are often affected by lighting conditions.

For our purpose, because the face model only needs to perform the same action as in the input image, we only care about its shape. In other words, the extraction of positions of feature points will meet our demand well, and ASM is already capable to achieve this. Another factor is that we aim for a real-time program, so ASM is also preferable in that it is faster. As a result, we have decided to use Active Shape Model as our technique for facial extraction.

4 Driving Animation

When facial features are extracted, our final question is how to convert them into data that is helpful for constructing the virtual face animation. This is the core step of the mirroring process.

We will focus on a linear animation paradigm, that is, blendshape, and give a brief introduction to a widely used example, the Facial Action Coding System. Finally we will discuss our mirroring rule, which is described as a problem of optimising a measurement of matching between the feature shape and virtual face model. The problem is solved using the Levenberg-Marquardt algorithm, and the solution will be the appropriate values of action units, which can directly drive the animation.

4-1 Blendshape Animation

Blendshape [13] is a powerful technique for representing and manipulate animation and is a good approach for facial animation, given the complexity and richness of facial actions. Blendshape tries to build all possible facial appearance from a series of (ideally completely independent) basic shapes, or "targets" (Figure 9). For each blendshape target, a weight parameter controls the proportion of contribution it has in the overall appearance, so facial appearance can be represented by a simple linear combination of blendshape targets. Or from a different point of view, an aspect of animation is controlled by a corresponding parameter.

Figure 9 An example of blendshape

Three blendshape bases representing "RAISE-BROW, SMIRK, and JAW-OPEN" respectively. These pictures are quoted from reference [13].

This idea of linear combination is the same as that in Principal Components Analysis, but blendshape targets can be purposely built to be intelligible, that is, to represent a meaningful (for humans) facial action. PCA eigenvectors, on the other hand, are completely derived from mathematical analysis.

At this point what we need is just a rule for suggesting proper values for blendshape parameters, according to the extracted feature points. We can apply this mapping rule to every frame of the input face action (as long as facial features are successfully extracted), and output the resulting parameters to the blendshape model at the same time. Then we can get an animation of the blendshape model, and the mirroring process is completed.

4-2 Facial Action Coding System

Facial Action Coding System is a widely used method to classify different aspects of facial actions developed by Ekman et al. in 1978. It was developed to directly measure facial behaviour [14]. In the system, facial expressions are represented by 46 independent atomic motions, called action units. Analysis of a face video using Facial Action Coding System was at first carried out manually by trained human FACS coders, which causes great inconvenience. Therefore, research on automating FACS has been also been conducted. As an example, we quote the action units that Donato et al. [15] researched on as an example.

Figure 10 Examples of action units

This figure is quoted from reference [15].

The decomposition of facial expressions into independent aspects of animation is consistent with blendshape. From our point of view, FACS can be adapted as our way of representing facial animation (though it is capable of more tasks). As in feature extraction we focus on the position of feature points, which are directly involved in and determined by facial action units, representing facial animation as action units will be a much straightforward way to connect facial features and animation.

4-3 The Mapping Rule

4-3-1 Geometric Approach

We first proposed a straightforward approach. Take the global Z-rotation as an easy example, when we have extracted feature points with ASM, we can draw a straight line across the two highest points on the face edge. See Figure 11, the line forms an angle $\angle\alpha$ with the horizontal line. $\angle\alpha$ can be easily calculated from the coordinates of the two points. In Figure 11, $\angle\beta$ is the Z-rotation angle, and we assume that $\angle\alpha = \angle\beta$.

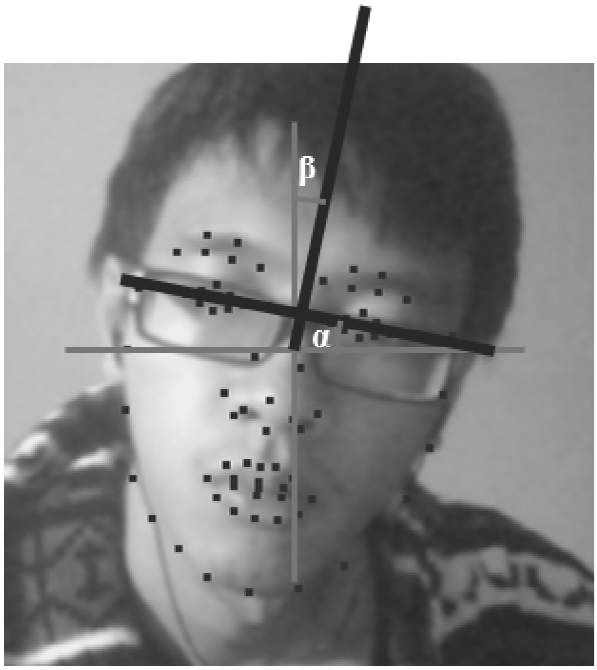


Figure 11 Global Z-rotation

This assumption is arguable. To make $\angle\alpha$ and $\angle\beta$ look equal, we have intentionally altered the orientation of the straight line when we plot Figure 11, and as a result, the line does not strictly pass through the two highest points on the face edge. The same issue still exists even if we choose other similar measurements such as the line passing through centres of eyes. Besides, the Z-rotation is not the only factor that influences the position of these two highest points. If rotations around other axes are also applied, they also partially determine the value of $\angle\beta$, or in other words, $\angle\beta$ no longer stands for Z-rotation alone. And in this case the relation between $\angle\alpha$ and $\angle\beta$ also becomes more complicated.

This is just a simple example of the complication of this approach. The position of a certain set of feature points can be influenced by many facial actions, and the same position of a certain set of feature point may be the result of different facial actions. In this approach we have to figure out a hierarchy of all factors and the order in which all parameters should be processed. Only after this can we proceed to building mirroring criteria for all parameters, not to mention the fact that each parameter is a unique case.

Based on the complicated and non-generic nature of this approach, we decide that we will not utilise it as the principal method for mirroring. Instead we need a holistic method to connect face image and face model.

4-3-2 Distance-based Approach

The approach we have undertaken is inspired by the work of Hou et al. [24]. Their aim is also mapping facial actions to a face model (and in fact they use the same face model as we do). In their approach, they convert the mapping problem to minimising an "energy function", which is used to measure the distance between face image and face model. Their energy function consists of three parts, including the sum of geometric distances of corresponding points between image and face model. We simplify the case by considering only this primary part, leaving out the other two.

The face model, basically, is built on a collection of 3-D vertices. Surfaces are defined on these vertices to form a polygonal shape, which is the face model. Based on this understanding, we can use a vector of 3-D points $\mathbf{c}$ to represent the model. In the case of blendshape models, the model is controlled by parameters, so $\mathbf{c}$ is a function of the parameter vector $\mathbf{t}$, which contains n parameters. Thus we write the model as $\mathbf{c}(\mathbf{t})$.

The 3-D coordinates are, however, internal data of the face model. As the model will ultimately be displayed to the user on screens, which is a flat 2-D surface, the 3-D coordinates need to be projected onto a 2-D image. We denote the projection as $\mathbf{u} = \mathbf{u}\big(\mathbf{c}(\mathbf{t})\big)$, where $\mathbf{u}$ is a vector of 2-D points and has the same number of vertices as in $\mathbf{c}(\mathbf{t})$.

On the other hand, feature points extracted from the ASM can also be expressed as a vector $\mathbf{u}'$. Assume that $\mathbf{u}$ and $\mathbf{u}'$ have the same dimension m (if not, we can select a subset of all available points to use), the total distance between the facial image and the model is expressed by

$$E = \sum_{k=1}^{m} \|\mathbf{u}'_k - \mathbf{u}_k\|^2$$

Or considering the fact that E $\mathbf{u}_k$ are dependent $\mathbf{t}$, this can be written as an explicit form:

$$E(\mathbf{t}) = \sum_{k=1}^{m} \left\|\mathbf{u}'_k - \mathbf{u}\big(\mathbf{c}(\mathbf{t})\big)_k\right\|^2$$

where† the subscript k means the k-th element of the vector. The mapping is done by finding the values of $\mathbf{t}$ such that $E(\mathbf{t})$ obtains the local minimum. This is a non-linear least squares problem. Hou et al. did not provide details of how they solved the problem. Our solution is by applying the Levenberg-Marquardt algorithm.

If we are more concerned about some pairs of points, for example, if we require more precise mapping in the eye, we can apply a weighting factor to each component of the sum in $E(\mathbf{t})$. Setting weighting factor for a particular pair will raise their influence on $E(\mathbf{t})$.

The above sum of distances is deemed by Hou et al. as the "external energy" incurred by face image. The other two parts of energy function depict the temporal and spatial smoothness, represented as derivatives to time. They depend solely on the face model itself and thus comprise the "internal energy". The purpose of including internal energy is to achieve smooth changes of parameter value, which is also preferable for our aim. However we still see this as a secondary feature that improves performance, so have not considered it. As a matter of fact, when we finish implementation the load of calculation result in delays between frames. So the animation is not smooth anyway.

4-4 Levenberg-Marquardt Method

In his work [25], Marquardt describes his method for solving the non-linear least squares problem as an "optimum interpolation" between the two classic approaches, Taylor series method and gradient method. We will give basic points of the method in detail. We have also noticed that the description of problem given by Marquardt is generic, including the situation of multivariable functions, but we only need a single-variable function situation. Therefore we will simplify Marquardt's expression.

The problem is defined as such: given a set of m data pairs $(x_i, y_i), i = 1,2, \dots m$, and a model function to be fitted to the data $y = f(x, \boldsymbol{\beta})$, where x is the independent variable and $\boldsymbol{\beta} = [\beta_1 \quad \dots \quad \beta_n]^T$ is a vector of n parameters, the problem is to estimate the values of parameters in $\boldsymbol{\beta}$ to minimise

$$\Phi = \sum_{i=1}^{m} \left(y_i - f(x_i, \boldsymbol{\beta})\right)^2$$

If we see the function $y = f(x, \boldsymbol{\beta})$ as a curve on the x-y plane, the result of minimising Φ is that the curve fits best to the known m points on the plane.

---

† For convenience of implementation, we treat the x- and y-coordinates separately. So it will be more sensible to rewrite the expression as such:

$$E(\mathbf{t}) = \sum_{k=1}^{m} \left(\mathbf{u}'_{kx} - \mathbf{u}\left(\mathbf{c}(\mathbf{t})\right)_{kx}\right)^2 + \sum_{k=1}^{m} \left(\mathbf{u}'_{ky} - \mathbf{u}\left(\mathbf{c}(\mathbf{t})\right)_{ky}\right)^2$$

Note that this equation has actually $2m$ components. And this is the right parameter to use when programming, instead of m.

Two classic methods are used to solve the problem. The Taylor series method uses the

first-order Taylor expansion to approximate the function:

$$f(x_i, \boldsymbol{\beta}_0 + \boldsymbol{\delta}) \approx f(x_i, \boldsymbol{\beta}_0) + \mathbf{J}_i \boldsymbol{\delta}$$

where $\boldsymbol{\beta}_0$ is the initial value of $\boldsymbol{\beta}$, $\boldsymbol{\delta} = [\delta_1 \quad \dots \quad \delta_n]^T$ is a small correction on $\boldsymbol{\beta}_0$, and $\mathbf{J}_i = [\frac{\partial f(x_i, \boldsymbol{\beta})}{\beta_1} \quad \dots \quad \frac{\partial f(x_i, \boldsymbol{\beta})}{\beta_n}]$ is the gradient of f with respect to $\boldsymbol{\beta}$. Then we have

$$\Phi(\boldsymbol{\beta}_0 + \boldsymbol{\delta}) \approx \sum_{i=1}^{m} (y_i - f(x_i, \boldsymbol{\beta}_0) - \mathbf{J}_i \boldsymbol{\delta})^2$$

or written in the matrix form:

$$\Phi(\boldsymbol{\beta}_0 + \boldsymbol{\delta}) \approx \|\mathbf{y} - \mathbf{f}(\boldsymbol{\beta}_0) - \mathbf{J}\boldsymbol{\delta}\|^2$$

where $\mathbf{y} = [y_1 \quad \cdots \quad y_m]^T$, $\mathbf{f}(\boldsymbol{\beta}_0) = [f(x_i, \boldsymbol{\beta}_0) \quad \dots \quad f(x_m, \boldsymbol{\beta}_0)]^T$, and $\mathbf{J} = \begin{bmatrix} \mathbf{J}_1 \\ \vdots \\ \mathbf{J}_m \end{bmatrix}$ is the Jacobian matrix.

ΦIn order for Φ to obtain the local minimum, we set its first derivative is zero and get the relation between $\boldsymbol{\delta}$ and other variables:

$$(\mathbf{J}^T\mathbf{J})\boldsymbol{\delta} = \mathbf{J}^T(\mathbf{y} - \mathbf{f}(\boldsymbol{\beta}_0))$$

This is a set of linear equations of $\boldsymbol{\delta}$, and can be solved relatively easily.

Meanwhile, the gradient method directly sets $\boldsymbol{\delta}$ to be the negation of the gradient of Φ with respect to $\boldsymbol{\beta}$:

$$\boldsymbol{\delta} = -\left[\frac{\partial \Phi}{\partial \beta_1} \quad \dots \quad \frac{\partial \Phi}{\partial \beta_n}\right]^T$$

When an optimised value of $\boldsymbol{\delta}$ is calculated as above, $\boldsymbol{\beta}$ is corrected by a fraction of it. Iteratively apply this correction until $\boldsymbol{\beta}$ convergence and the optimal values of $\boldsymbol{\beta}$ is approximated.

The Taylor series method converges rapidly, but divergence of the successive iterates may happen. The gradient method has better ability to converge, but after the first few iterations convergence may become very slow. Levenberg and Marquardt modified Taylor series method by introducing a factor λ for the :

$$(\mathbf{J}^T\mathbf{J} + \lambda\mathbf{I})\boldsymbol{\delta} = \mathbf{J}^T(\mathbf{y} - \mathbf{f}(\boldsymbol{\beta}_0))$$

The value of λ is dynamically adjusted in every iteration, to achieve better convergence. When λ is set to small values, the method is closer to the Taylor series method, and when λ is set to large values, the method shows properties of the gradient method. In this way, the

Levenberg-Marquardt algorithm compensates the weakness of Taylor series method with the best features of gradient method, and vice versa.

5 Implementation

By far, we have discussed relevant techniques for all the three major steps of the mirroring process, and made our choice. In this section we will give details of our software implementation. We have made use of a range of publicly available libraries that implement the aforementioned algorithms, along with Qt [16] for Graphics User Interface. We develop our mirroring program in Microsoft Visual Studio 2008 using a Qt-incorporated C++. And at last we will give example results of the program.

5-1 OpenCV and Cascade Classifier

One of the key software kits we use is the well-known open source library for computer vision, OpenCV [17] [18]. OpenCV implements a vast range of techniques in or related to computer vision, including image processing and machine learning. We choose OpenCV because it is basic, popular and easy to use. Another reason is that we use C++ as our programming language. OpenCV supports, and is written in, C/C++, and is optimised. This combination works quite efficiently.

As the acquisition and manipulation of images form the foundation of our application, we have applied OpenCV functions for all aspects of image processing, from input (including acquiring images from webcam), output and internal representation, to basic processing such as scaling, converting between colour and gray-scale, and flipping†. Besides, we have also made use of the OpenCV implementation of the Viola-Jones classifier (Figure 12), which is called "cascade classifier" in the programming. OpenCV also contains several versions of trained classifier data, for detecting both faces and other rigid objects, so we do not need to

Figure 12 Result of the demo program of cascade classifier

† Webcams do not work like mirrors, but like in chatting applications, users may prefer to see their own face on the screen as they would do in a mirror. Therefore acquired images need to be flipped horizontally first.

build classifier from scratch.

5-2 Implementation of ASM

We first intended to build our own Active Shape Model. But given the fact that the only realistic option for us is still publicly available and well used training sets such as the BioID set developed by Jesorsky et al. [19], we decided that this time-consuming process may not lead us to a better solution than those readily publicised. As a result we make use of two libraries that provide both functionalities of feature extraction by ASM and trained models.

5-2-1 Stasm

Stasm is an open source library for Active Shape Model implemented by Stephen Milborrow [20], making use of OpenCV. It also contains several trained ASM data files. We use a 76-feature model, which is built using other data sets but tested on BioID. Stasm also improves the classical ASM in several ways. One is that they use a 2-D region near landmarks to build the profile model instead a 1-D line segment, obtaining good results for the nose and eyes.

Stasm provides a single function which reads an image and outputs extracted results. The function first searches face using Viola-Jones or Rolway detector to provide the start shape, and then (if a face is detected) apply ASM to find face features. The function is well encapsulated so users do not need to know these details.

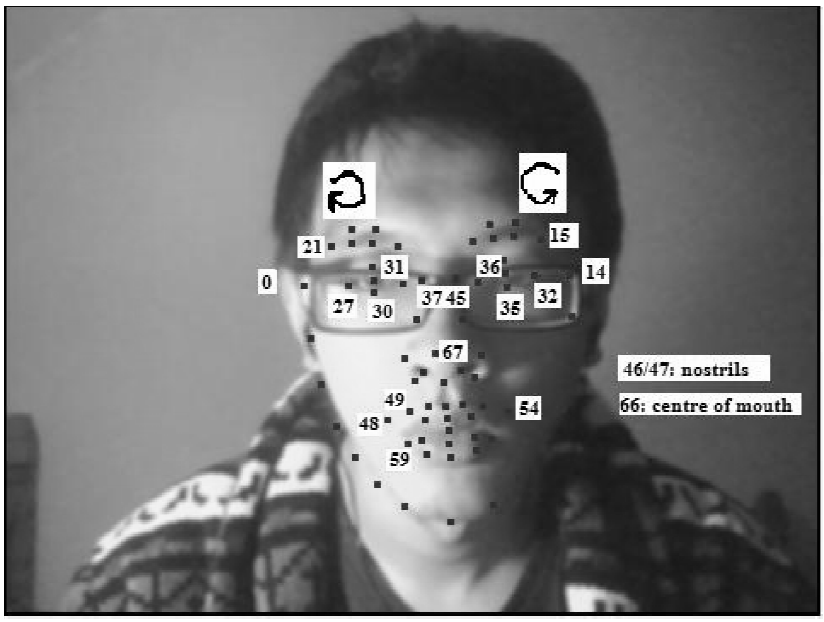
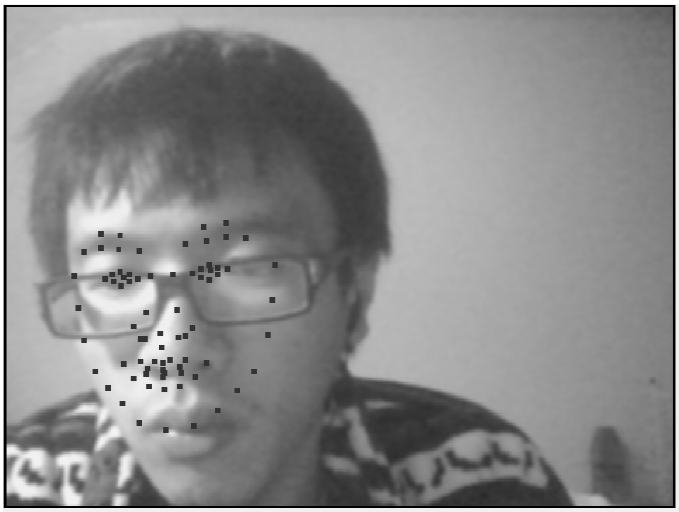


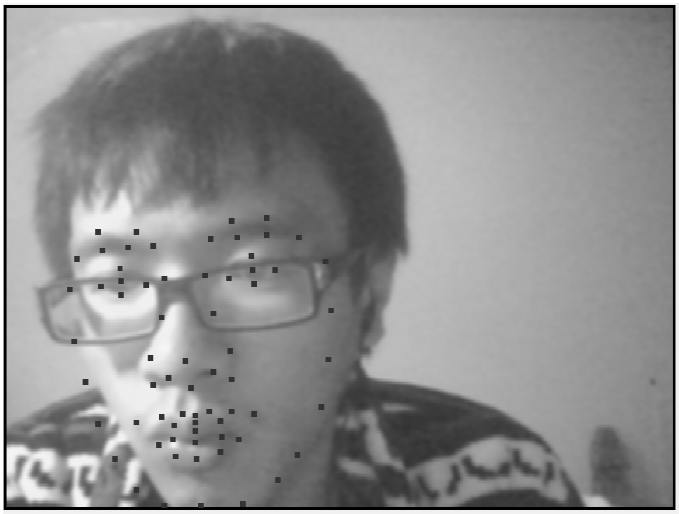
Figure 13 Feature points in the 68-feature ASM

Points 0~14 are on the edge of face. Points in the brows and eyes are numbered in the direction indicated by the arrows, starting from 15 and 21 respectively (with 31 and 36 on the centres). Numbers of other points are selectively given.

One of the drawbacks is that an image has to be saved to a temporary file before being processed, despite the fact that the function already receives a pointer to the image data. For our purpose, ASM search is done for every frame, so this is redundant and repetitive, and may influence processing speed, depending on hardware.



(a)



(b)

Figure 14 Example extraction results of the two ASM libraries

These two images are screen-shots from continuous mirroring sessions. (a) is from Stasm and (b) from ASMLibrary. In the experiement ASMLibrary can achieve considerably good results, though it still occasionally makes the same mistake as Stasm. But given a similar face position, Stasm hardly gets the right place and during the session there are worse cases than shown in (a).

5-2-2 ASMLibrary

ASMLibrary is another publicly available library written by Yao Wei [21]. It contains a

trained 68-feature model (Figure 13). Compared to the 76-feature model of Stasm, the missing 8 feature points are all in the eyes. Apart from these, all features are the same. ASMLibrary is purely for ASM, so unlike in the Stasm case, we have to do face detection separately. We use the implementation of Viola-Jones detector provided by OpenCV, and then build a start shape in the detected region using a function provided in ASMLibrary, before finally passing it to the ASM fitter function for recognition.

5-2-3 Comparison

We implemented two versions of mirroring program using the two libraries respectively. From our observation, recognition by ASMLibrary is slightly better than by Stasm. One example is given in Figure 14.

There is no significant difference between the two libraries regarding processing speed (both quite slow on an Intel 1.80GHz dual core processor, roughly one frame per second). But when face detection fails, the Stasm version runs smoothly whilst in the ASMLibrary version delay still occurs between frames, indicating that the configuration of the Stasm internal face detector makes it much faster than the OpenCV implementation of Viola-Jones detector.

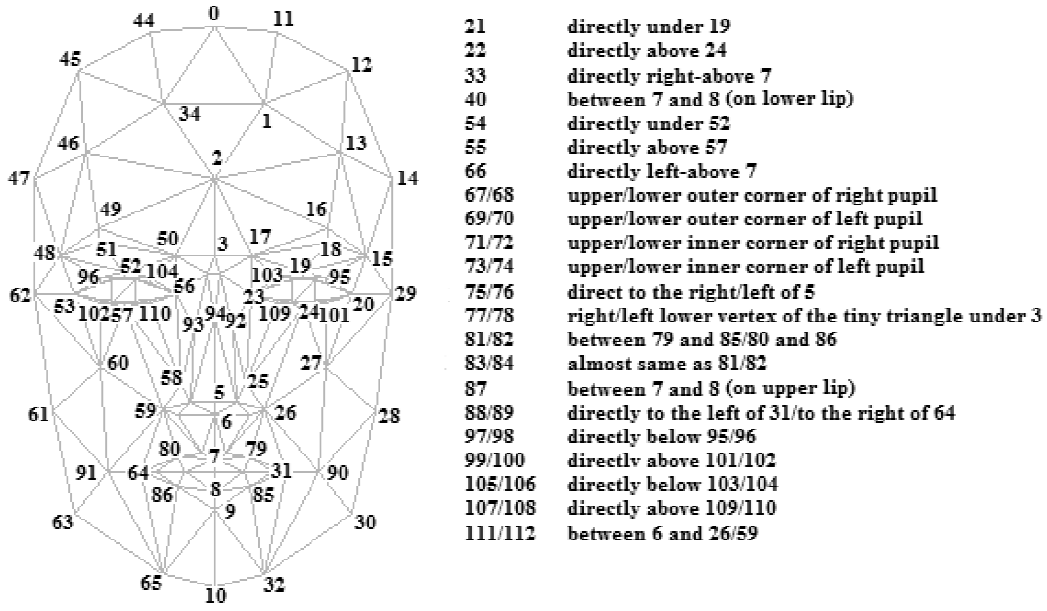


Figure 15 Candide model vertices

5-3 Candide Face Model

Candide [22] is a publicly available parameterised wireframe face model. The version we used, Candide-3, contains 113 vertices, 184 triangular surfaces, 61 parameters (including 9 geometric transformation parameters, 14 action units and 38 shape units). Figure 15 gives the 2-D projection of Candide in its initial state, or “mean shape”, and also marks all the indices allocated to vertices.

J. Ahlberg developed a library for handling wireframe models called eruFace, and Candide is actually a specific case, as well as the main focus, of this library. The eruFace library reads model data from a formatted file, so theoretically with eruFace one can also build their own face model as long as they can describe their model with the required format. However, eruFace is not well maintained and depends on several other libraries, so during our adaptation we had to make some modifications to make it work. With eruFace, Candide

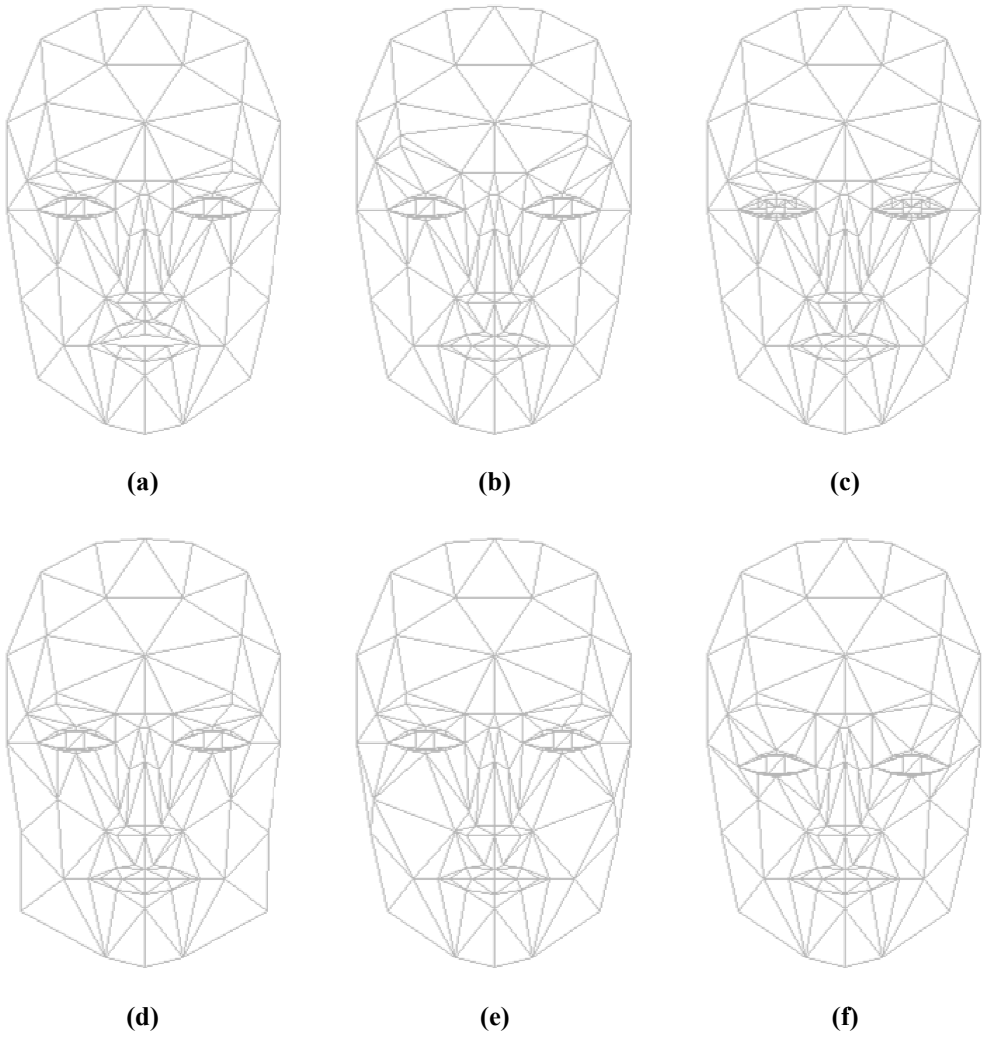


Figure 16 Parameters of Candide model

The first line shows three action units:
(a) parameter 13: Upper lip lip raiser (AU10), (b) parameter 17: Outer brow raiser,
(c) parameter 20: Eye closed (AU42/43/44/45).

The second line shows three shape units:
(d) parameter 27: Chin width, (e) parameter 40: Cheeks separation distance,
(f) parameter 50: Eyes vertical position.

model can also be rendered with textures, but this has little influence of our demonstration. So we prefer to avoid this extra calculation.

Within the Candide model, action units are parameters that control multiple vertices to perform meaningful facial actions, whilst shape units only control individual aspects of face shape. Examples are given in Figure 16. It is worth noticing that the action units are a subset of those in the Facial Action Coding System. All parameters are indexed, with 0~8 being geometric transformation parameters, 9~22 action units and 23~60 shape units.

5-4 Levmar Library and the Mirroring Scheme

We find an open-source library of the Levenberg-Marquardt algorithm – Levmar [26]. The library provides both single- and double-precision implementation of the algorithm. And either pre-calculated analytic or numerical (estimated) Jacobian matrices can be used. We use the double-precision version with numerical Jacobian matrix. Within the Candide model, the mapping between parameters and face model vertex coordinates is complicated, and we have chosen relatively l arge numbers of parameters and vertices, therefore an analytic Jacobian matrix is neither practical nor necessary.

The LevMar function needs the following parameters:

- Number and values of models,
- Number and initial values of parameters,
- A target function describing the relation between parameters and model values, which can also be deemed as the function of the curve to be fitted to the given models,
- Other options, including maximum number of iterations.

The ASM models we chose have 68/76 feature points, but Candide has 113 vertices. Additionally, not all feature points have counterparts in the face model. Therefore we have to choose a subset from both to be used in our distance-based mirroring approach.

Hu et al. [23] also worked on Candide, and they named 19 key vertices in the Candide model that exactly match the ASM. We selected some of these vertices, and added some others of our own choice. The resulting mirroring scheme contains a total of 37 corresponding pairs (see Table). And among the 61 parameters of Candide model, we choose 17 (see Table). So if we fit our case into the mathematical expressions in section 4-3-2, the number of model points m is 37†, and the number of parameters n is 17. The model set $\mathbf{u}'$ is the coordinates of the 37 feature points, and target function $\mathbf{u}(\mathbf{c}(\mathbf{t}))$ is the coordinates of the 37 chosen vertices in Candide model, controlled by parameter vector $\mathbf{t}$. The Levenberg-Marquardt algorithm is operated to approximate optimal values of $\mathbf{t}$ that minimises the sum of distances

† Each model points consist of two values for x- and y-coordinates. For convenience we treat them individually in our program, thus the actual number of model values passed to the Levmar function is 74, twice of 37.

$E(\mathbf{t})$, and when solved these values are used directly to drive the Candide model.

Table 1 Corresponding Pairs

Feature point number	Candide model vertex number	Feature point number	Candide model vertex number
14	29	43	26
0	62	39	59
11	28	44	92
3	61	38	93
10	30	42	111
4	63	40	112
8	32	47	75
6	65	46	76
7	10	54	31
15	15	48	64
16	16	51	7
18	17	57	8
21	48	61	40
22	49	64	87
24	50	63	81
20	18	65	82
26	51	62	83
67	5	60	84
41	6		

Table 2 Chosen Parameters

Parameter number	Description
0	Global rotation X
1	Global rotation Y
2	Global rotation Z
3	Global translation X
4	Global translation Y
6	Global scale X
7	Global scale Y
12	Jaw drop
23	Head height
9	Nose wrinkler
13	Upper lip raiser (AU10)
14	Lip stretcher (AU20)
15	Lip corner depressor (AU13/15)
17	Outer brow raiser
18	Inner brows raiser

Table 2 Chosen Parameters

19	Brow lowerer
29	Nose middle vertical position

Since we use numerical Jacobian matrix, its estimation is included inside the function. The dimensions of Jacobian matrix is $m \times n$, which means that for each parameter m partial derivatives are calculated. And for each partial derivative the internal calculations of Candide model need to be done, which involves all n parameters. So the complexity of estimation of Jacobian matrix is at least square of n. For bigger values of n, the processing time may increase significantly.

To improve performance, we split the parameter vector into two sub-vectors and apply two rounds of Levenberg-Marquardt approximation instead of one, and wrote two target functions accordingly. The first round deals with the first 9 parameters, which control global aspects of the face model (geometric transformations including rotation, translation and scaling, and head height). The second round processes the remaining 8 parameters, which are action units that control facial features. Since for $n_1, n_2 > 0$ it holds that $n_1^2 + n_2^2 < (n_1 + n_2)^2$, this design can reduce the amount of calculation needed. And moreover, in our experiment although one extra round of approximation is conducted, the total number of iterations in the two-round version roughly stays the same as the one-round version (around 90). As a result the two-round version runs faster.

In addition, we have also applied a weighting scalar for each pair. For points on the outer contour we set the weight to 1.0, and for internal feature points we set the weight to 1.1~1.35. This is because internal feature points appear in smaller neighbourhoods, and their absolute value of distances are naturally smaller. Therefore we manually increase their contribution to the total distance.

It is worth noticing that we have not chosen any point / vertex in the eye, as vertices in the eye of Candide model are dense, meanwhile in our chosen ASMs only 5 or 9 feature points are available in the eye. But this will not affect scalability. In our implementation all the above indices and weights are stored in lists, and thus new pairs can be easily added when more detailed mirroring is wished.

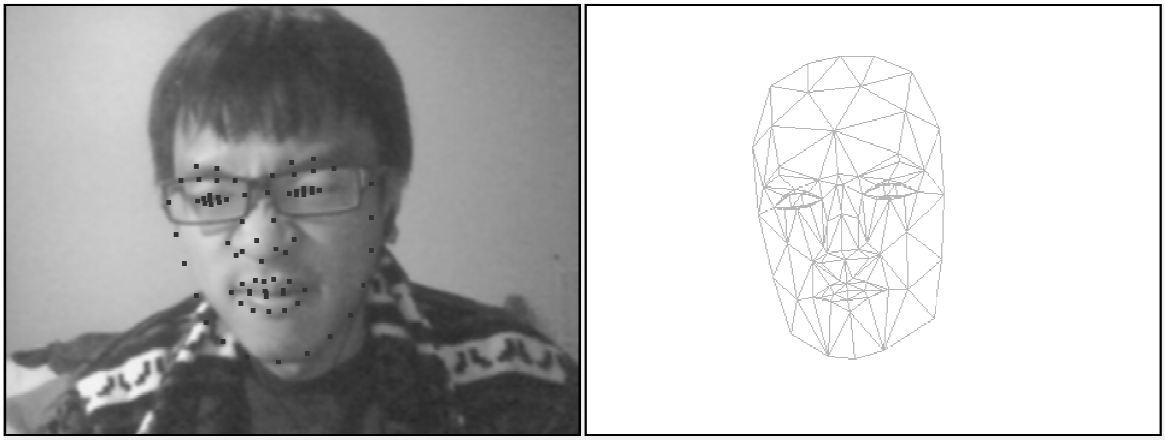
5-5 Example Results

Some successful mirroring results are listed in Figure 17.

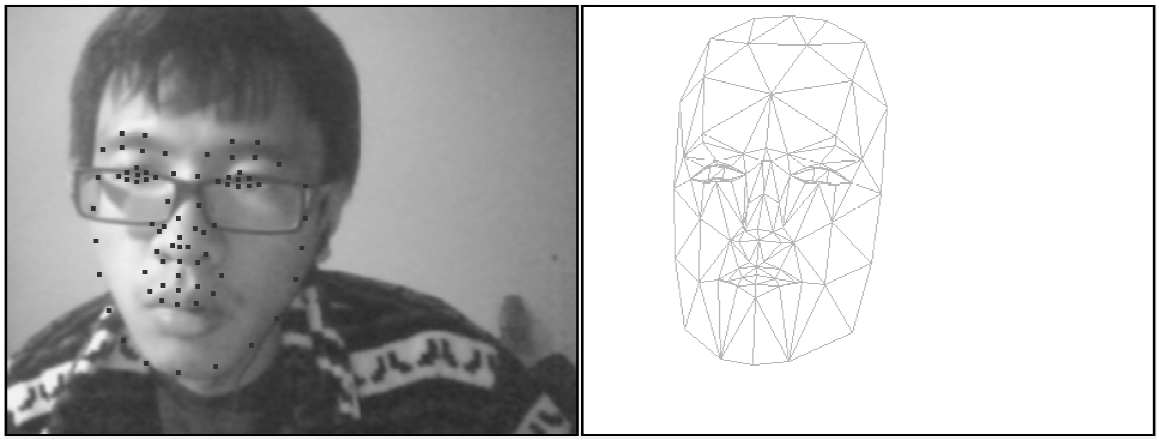
5-5-1 Performance of the Mirroring Scheme

From the results we can see that our method works fairly well in mirroring the global position of face. In (c) we can see a hint of widened nose, but the mirroring of the internal features is still not always satisfactory, even if we have applied weights on those points. This may also

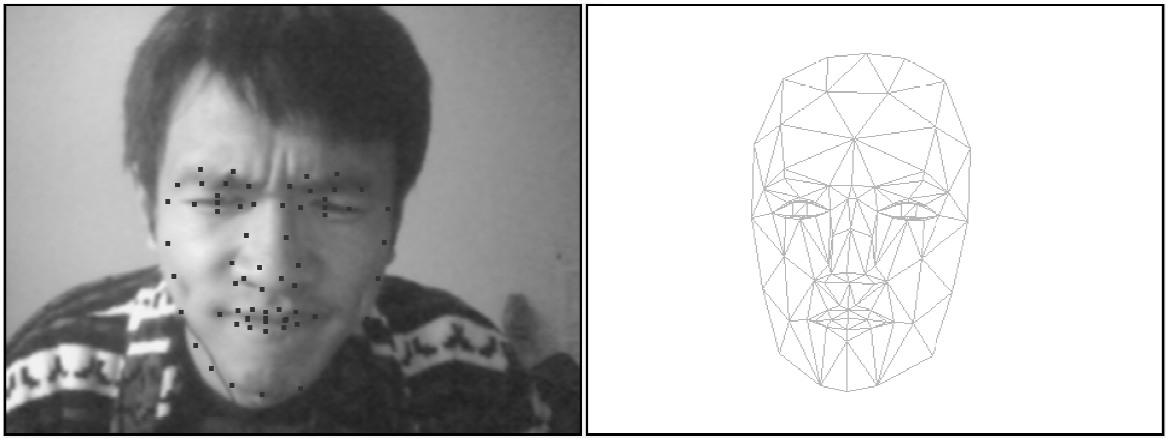
be caused by the fact that we have chosen a relatively small portion of all available parameters of the Candide model, especially shape units.



(a)

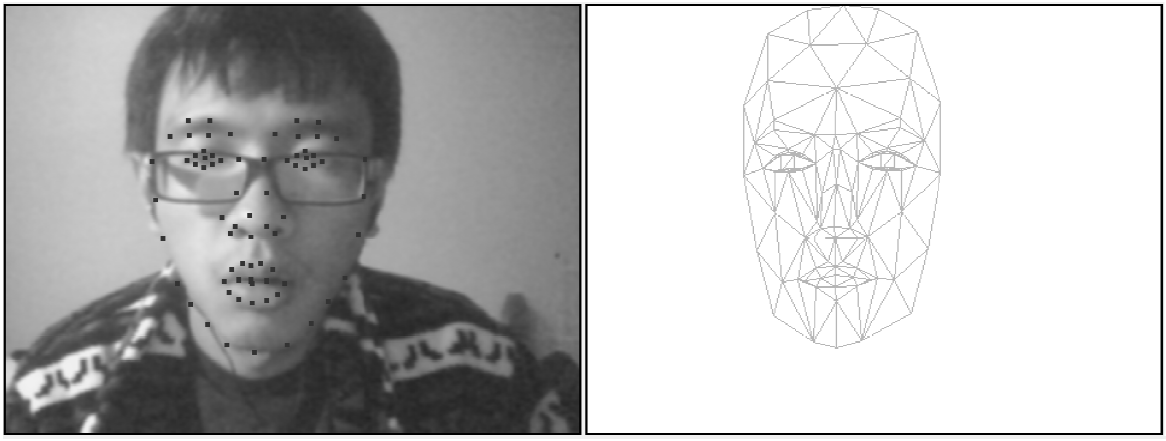


(b)

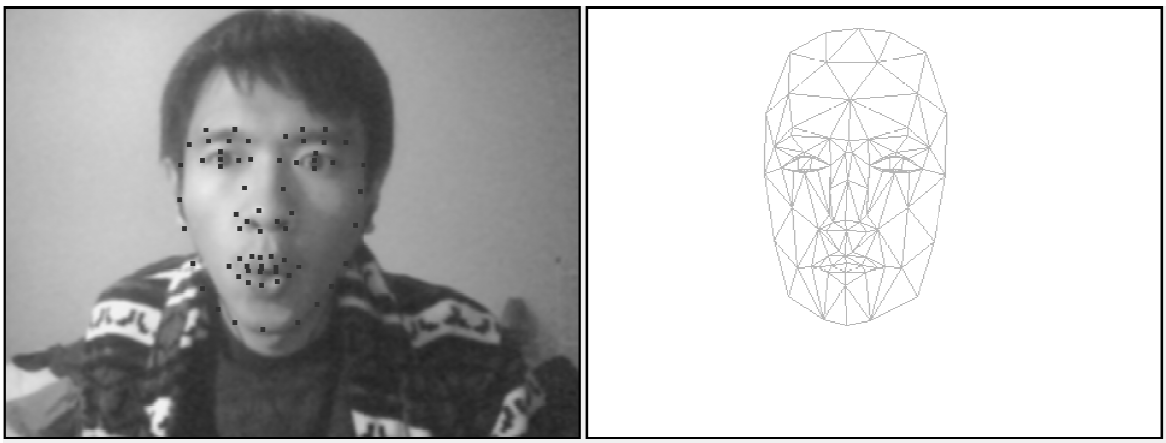


(c)

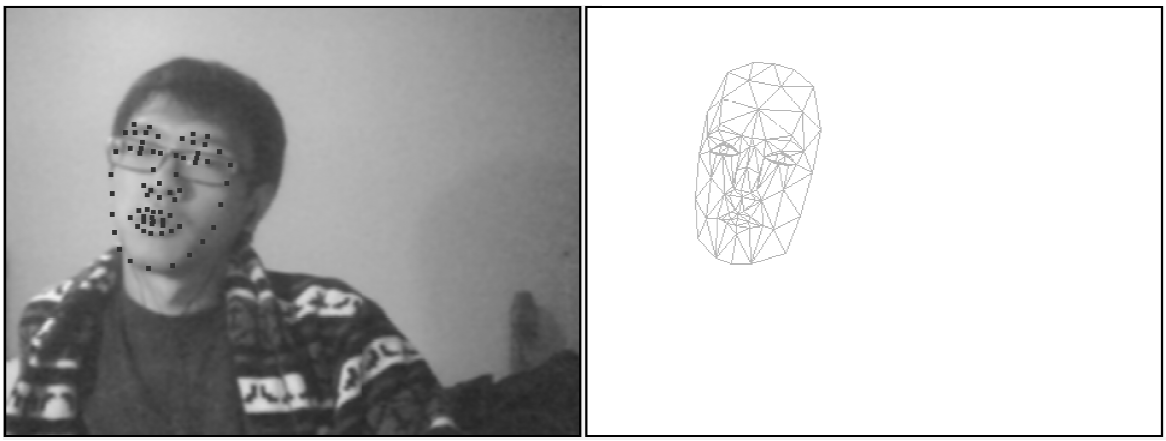
Figure 17 Example mirroring results (to be continued)



(d)

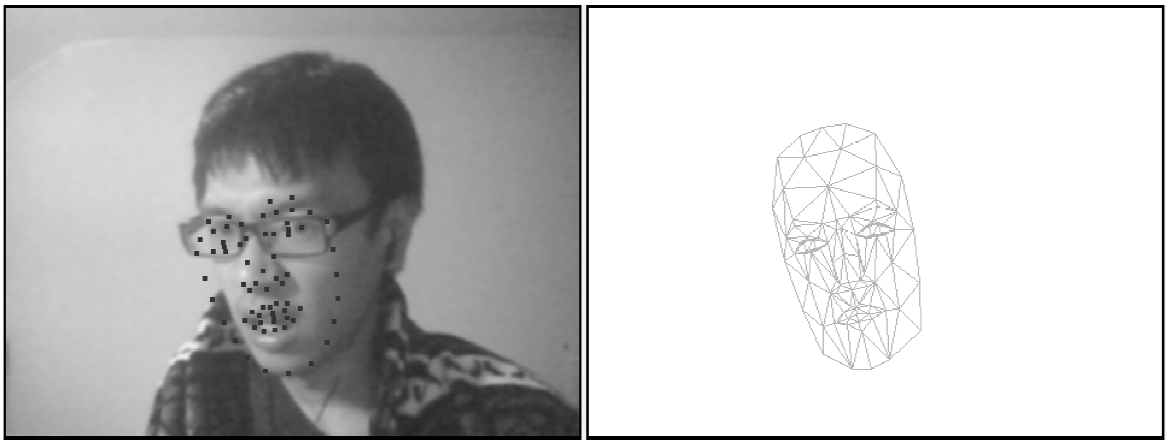


(e)



(f)

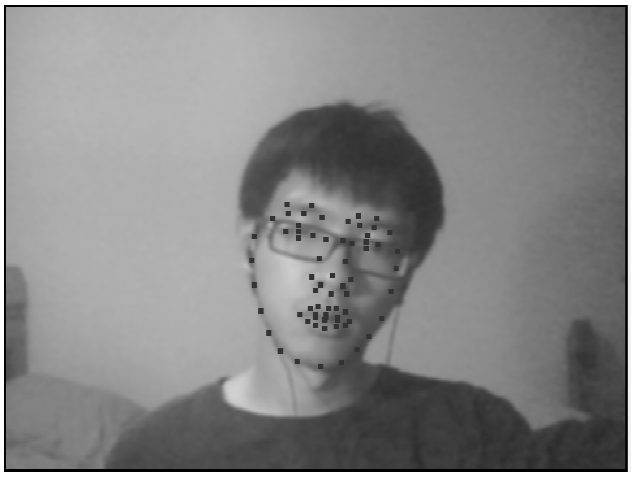
Figure 17 (continued) Example mirroring results



(g)

Figure 17 (continued) Example mirroring results

(a)



(b)

Figure 18
Performance of extraction under different head rotation

5-5-2 Performance of the Active Shape Model

The quality of feature extraction plays a fundamental role in the mirroring. As can be demonstrated by (b), (e) and (g), where feature positions in the mouth are extracted incorrectly and the mirroring results are consequently not good.

The two key factors that affect the ASM performance are the quality of training sets and the performance of face detector. In our experiement, the performance of cascade classifier exacerbates in weak lighting (Figure 18), as in (a) half of the face is much darker than the other half. As for the ASM training set, we have already made a comparison between Stasm and ASMLibrary in section 5-2-3, but both libraries may give wrong suggestions for facial features, especially in the mouth and eyes, and when lighting is weak. Section Our

5-5-3 Performance of the Program

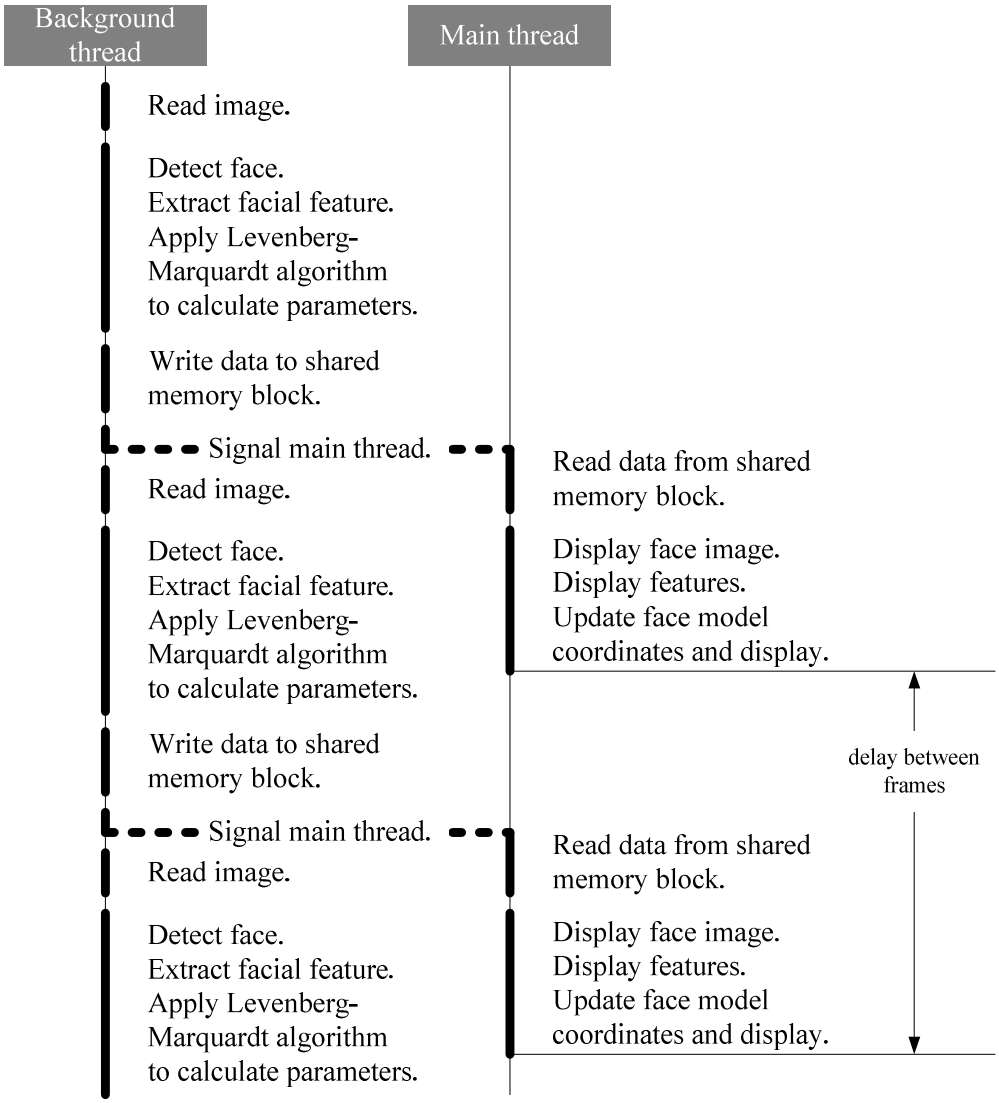


Figure 19 The two threads

Both ASM and Levenberg-Marquardt algorithm require plenty of calculation. From our perspective the processing speed of our mirroring program is barely acceptable for a real-time system (roughly one frame per second on our Intel 1.80GHz dual core processor). Yet we have addressed this requirement on speed, and enhanced our mirroring program by designing it as a two-thread program (Figure 19), rather than having everything executed serially. We have put all the calculation in the background thread, and the main thread is responsible for user interface. The two threads run in parallel and communicate by a shared memory block, and the manipulation of shared resource by both threads has been kept to a minimum.

6 Discussion and Conclusion

We have proposed a framework of techniques that enable virtual face mirroring. We have emphasised on linear methods a lot, as they are basic, simple, and easy to implement. From the screenshots of our mirroring program shown in Section 5, the overall quality of mirroring is such that we can claim that the goal of virtual face mirroring is basically accomplished, and that the techniques we have chosen suit our aim.

However, if we compare our results to the desirable properties we discussed in section 1-3, our mirroring program does not handle variation in lighting well. Both face detection and facial feature extraction may fail even when the face action is only moderate. This means that our models are not sufficiently robust. We may improve by developing our own models.

The quality of mirroring of some internal parts of face, especially mouth, is much worse than the overall pose and position, indicating that the distance function still need to focus more on internal vertices, and the situation is not as simple as merely magnifying the weighting factor.

And we have to mention that the processing speed of our program is barely acceptable as a real-time system, though we have already adopted a two-thread design to improve. If our framework is to gain practical use, extensive optimisation is a prerequisite that must be focused on.

References

[1] K. W. Bowyer, K. Chang, and P. Flynn, “A Survey of Approaches to Three-dimensional Face Recognition”, Pattern Recognition, vol. 1, pp. 358~361, 2004

[2] R. A. Patil, V. Sahula, and A. S. Mandal “Automatic Recognition of Facial Expressions in Image Sequences: A Review”, International Conference on Industrial and Information Systems, pp. 408~413, 2010

[3] S. Gong, S. J. McKenna, and A. Psarrou, “Dynamic Vision: From Images to Face Recognition”, Imperial College Press, 2000.

[4] M. Turk, and A. Pentland, “Eigenfaces for Recognition”, Journal of Cognitive Neuroscience vol.3, no.1, pp. 71~86, 1991.

[5] P. Viola, and M. Jones, “Rapid Object Detecting using a Boosted Cascade of Simple Features”, Computer Vision and Pattern Recognition, vol. 1, pp. 511~518, 2001.

[6] M. Pantic, and L. J. M. Rothkrantz, “Automatic Analysis of Facial Expressions: The State of the Art”, IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 22, no. 12, pp. 1424~1445, Dec. 2000.

[7] P. Sinha, B. Balas, Y. Ostrovsky, and R. Russell, “Face Recognition by Humans: Nineteen Results All Computer Vision Researchers Should Know About”, Proceedings of the IEEE, vol. 94, no. 11, pp. 1948~1962, Nov. 2006.

[8] M. Yang, D. Kriegman, and N. Ahuja, “Detecting Faces in Images: A Survey”, IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 24, no. 1, pp. 34~58, Jan. 2002.

[9] T. F. Cootes, G. J. Edwards, and C. J. Taylor, “Active Appearance Models”, IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 23, no. 6, pp. 681~685, 2001.

[10] X. Gao, X. Li, and D. Tao, “A Review of Active Appearance Models”, IEEE Transactions on Systems, Man, and Cybernetics—Part C: Applications and Reviews, vol. 40, no. 2, pp. 145~158, 2010.

[11] T. F. Cootes, C. J. Taylor, D. H. Cooper, and J. Graham, “Active Shape Models – Their Training and Application”, Computer Vision and Image Understanding, vol. 61, no. 1, pp. 38~59, 1995.

[12] S. Yan , C. Liu , S. Z. Li , H. Zhang , H. Shum and Q. Cheng, “Face Alignment Using Texture-constrained Active Shape Models”, Image and Vision Computing, vol. 21, pp.69~75,

2003.

[13] J. P. Lewis, and K. Anjyo, “Direct-Manipulation Blendshapes”, Computer Graphics and Applications, IEEE, vol. 30, no. 4, pp.42~50, 2010.

[14] M. S. Bartlett, J. C. Hager, P. Ekman, and T. J. Sejnowski, “Measuring Facial Expressions By Computer Image Analysis”, Psychophysiology, vol. 36, pp. 253~263, 1999.

[15] G. Donato, M. S. Bartlett, J. C. Hager, P. Ekman, and T. J. Sejnowski “Classifying Facial Actions”, IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 21, no. 10, pp. 974~989, 1999.

[16] Nokia Corporation, Qt – A cross-platform application and UI framework, Retrieved September 30, 2011 from http://qt.nokia.com/products/.

[17] G. Bradski, and A. Kaehler, “Learning OpenCV”, O’Reilly, 2008.

[18] “OpenCV 2.1 C++ Reference”, Retrieved September 29, 2011 from http://opencv.willowgarage.com/documentation/cpp/.

[19] O. Jesorsky, K. Kirchberg, and R. Frischholz “Robust Face Detection Using the Hausdorff Distance”, Third International Conference on Audio- and Video-based Biometric Person Authentication, pp.90~95, 2001.

[20] S. Milborrow, “Locating Facial Features with an Extended Active Shape Model”, European Conference on Computer Vision, pp.504~513, 2008.

[21] W. Yao, “ASMLibrary – Library of Active Shape Model”, Retrieved September 27, 2011 from http://code.google.com/p/asmlibrary/.

[22] J. Ahlberg, “CANDIDE – a parameterized face”, Retrieved September 27, 2011 from http://www.icg.isy.liu.se/candide/.

[23] F. Hu, Y. Lin, B. Zou, and M. Zhang, “Individual 3D Face Generation Based on Candide-3 for Face Recognition”, Congress on Image and Signal Processing, vol. 1, pp. 646~650, 2008.

[24] Y. Hou, P. Fan, I. Ravyse, and V. Enescu, “Smooth Adaptive Fitting of 3D Face Model for the Estimation of Rigid and Non-rigid Facial Motion in Video Sequences”, International Conference on Image and Graphics, pp.477~484, 2009.

[25] D. Marquardt, “An Algorithm for Least-Squares Estimation of Nonlinear Parameters”, SIAM Journal on Applied Mathematics, vol. 11, no. 2, pp. 431~441, 1963.

[26] M. I. A. Lourakis, “Levmar: Levenberg-Marquardt nonlinear least squares algorithms in C/C++”, Retrieved September 28, 2011 from http://www.ics.forth.gr/~lourakis/levmar/.

Appendix: Codes

```
*********************************************************************************************

//ASM Thread

#include "asmthread.h"
#include "LevMarMirror.h"
#include "cv.h"
#include <fstream>
#include <vector>

extern cv::Mat captureImage;
extern bool capturing;
extern bool AsmProcessing;
extern bool MainProcessing;
extern int* featurePoints;
extern double* parameters1;
extern double* parameters2;
extern cv::VideoCapture videoCap1;
asm_model AsmModel;
cv::CascadeClassifier Classifier;
extern int iterations;

asm_shape *detect(cv::Mat& img, cv::CascadeClassifier& cascade);

void AsmThread::run()
{
     int nmarks = 0;
     int marks[200];
     static FILE *AsmModelFile = NULL;

     while(capturing)
     {
          cv::Mat capture;
          cv::Mat asmImage;
          videoCap1 >> capture;
          cv::resize(capture, asmImage, cv::Size(480, 360));
          cv::flip(asmImage, asmImage, 1);
          cv::GaussianBlur(asmImage, asmImage, cv::Size(5, 5), 1.5);
          cv::cvtColor(asmImage, asmImage, CV_BGR2RGB);

          asmfitting fit;
          if(AsmModelFile == NULL)
          {
               fit.Read("E:\\Studies\\asmlibrary-win-6.0\\example\\my68-1d.amf");

     Classifier.load("E:\\Studies\\OpenCV2.1\\data\\haarcascades\\haarcascade_frontalf
ace_alt.xml");
                    }

          asm_shape *detShape = detect(asmImage, Classifier);
          asm_shape initShape;
          if(detShape != NULL)
          {
               InitShapeFromDetBox(initShape,    *detShape,    fit.GetMappingDetShape(),
fit.GetMeanFaceWidth());
               delete detShape;
               cv::Mat grayImage;
               cv::cvtColor(asmImage, grayImage, CV_RGB2GRAY);
               fit.Fitting(initShape, &(grayImage.operator IplImage()));
               nmarks = initShape.NPoints();
          }
          else
               nmarks = 0;

          for(int i = 0; i < nmarks; i++)
          {
               marks[2 * i] = initShape[i].x;
               marks[2 * i + 1] = initShape[i].y;
          }

          int itcount = 0;
          if(nmarks > 0)
               itcount = LevMarMirror::LevMarMirroring(marks);


          while(MainProcessing) ;
          AsmProcessing = true;
          featurePoints = new int[2 * nmarks + 1];
          *featurePoints = nmarks;
```

```
        memcpy(featurePoints + 1, marks, 2 * nmarks * sizeof(int));
        captureImage = asmImage.clone();
        parameters1 = new double[LevMarMirror::m1];
        parameters2 = new double[LevMarMirror::m2];
        memcpy(parameters1, LevMarMirror::p1, LevMarMirror::m1 * sizeof(double));
        memcpy(parameters2, LevMarMirror::p2, LevMarMirror::m2 * sizeof(double));
        iterations = itcount;
        AsmProcessing = false;

        parent->AsmFinished();
    }
}

asm_shape *detect(cv::Mat& img, cv::CascadeClassifier& cascade)
{
    std::vector<cv::Rect> faces;
    cv::Rect PrimeFace;
    double scale = 3.0;
    cv::Mat gray, smallImg( cvRound (img.rows/scale), cvRound(img.cols/scale), CV_8UC1 );

    cv::cvtColor( img, gray, CV_BGR2GRAY );
    cv::resize( gray, smallImg, smallImg.size(), 0, 0);
    cv::equalizeHist( smallImg, smallImg );

   cascade.detectMultiScale( smallImg, faces,
      1.1, 2, 0
      |CV_HAAR_SCALE_IMAGE
      ,
        cv::Size(30, 30) );
    if(faces.size() > 0)
    {
        for(int i = 0; i < faces.size(); i++)
        {
            if(i == 0)
                PrimeFace = faces[0];
            else if(faces[i].area() > PrimeFace.area())
            {
                PrimeFace = faces[i];
            }
        }
        PrimeFace.x *= scale;
        PrimeFace.y *= scale;

        PrimeFace.width *= scale;
        PrimeFace.height *= scale;

        asm_shape *Detected = new asm_shape();
        Detected->Resize(2);
        (*Detected)[0].x = PrimeFace.x;
        (*Detected)[0].y = PrimeFace.y;
        (*Detected)[1].x = PrimeFace.x + PrimeFace.width;
        (*Detected)[1].y = PrimeFace.y + PrimeFace.height;
        return Detected;
    }
    else
        return NULL;
}


*****************************************************************************************
*****************************************************************************************

//Main Thread

#include "MirrorTrial.h"
#include "asmthread.h"
#include <QLabel>
#include <QString>
#include <QtGui/QMessageBox>
#include <fstream>
#include <vector>
#include "LevMarMirror.h"

cv::Mat captureImage;
AsmThread *asmthr;
bool capturing;
bool AsmProcessing = false;
bool MainProcessing = false;
int *featurePoints;
double *parameters1;
double *parameters2;
cv::VideoCapture videoCap1;
cv::Mat faceImage;
int iterations;
```

```
MirrorTrial::MirrorTrial(QWidget *parent, Qt::WFlags flags)
    : QMainWindow(parent, flags)
{
    ui.setupUi(this);
    QObject::connect(ui.Button1, SIGNAL(clicked()), this, SLOT(Button1_Click()));
    asmthr = new AsmThread(this);
    QObject::connect(this, SIGNAL(ImageGrabbed()), this, SLOT(ReceiveImage()));


    capturing = false;

    FaceModel.read("candide3_.wfm");
    faceImage = cv::Mat(360, 480, CV_8UC3);
    faceImage.setTo(CV_RGB(255, 255, 255), cv::Mat::ones(360, 480, CV_8UC1));
    FaceModel.updateImageCoords(3, 480, 360);
    FaceModel.draw(faceImage, CV_RGB(0, 255, 0));
    QPixmap   Pixmap1   =   QPixmap::fromImage(QImage(faceImage.data,   faceImage.cols,
faceImage.rows, QImage::Format_RGB888));
    ui.LabelFaceModel->setPixmap(Pixmap1);
}

MirrorTrial::~MirrorTrial()
{

}

void MirrorTrial::ShowFeatures(const std::vector<QPoint> &p)
{
    static std::vector<QLabel*> labels;
    if(!labels.empty())
    {
        for(int i = 0; i < labels.size(); i++)
            delete labels[i];
        labels.clear();
    }
    for(int i = 0; i < p.size(); i++)
    {
        labels.push_back(new QLabel(ui.centralWidget));
        labels[i]->setGeometry(p[i].x() + ui.LabelImage->geometry().x() - 2, p[i].y() +
ui.LabelImage->geometry().y() - 2, 4, 4);
        labels[i]->setStyleSheet("border: 2px solid #333333");
        labels[i]->show();
    }
}

void MirrorTrial::Button1_Click()
{
    if(capturing)
    {
        capturing = false;
        ui.Button1->setText("Start");
        videoCap1.release();
    }
    else
    {
        capturing = true;
        ui.Button1->setText("Stop");
        videoCap1 = cv::VideoCapture(0);
        videoCap1.set(CV_CAP_PROP_FPS, 12.0);
        if(videoCap1.isOpened())
            asmthr->start();
        else
            Button1_Click();
    }
}

void MirrorTrial::ReceiveImage()
{
    std::vector<QPoint> feature;
    cv::Mat image;
    int *featureCoords;
    double *candideParameters1;
    double *candideParameters2;
    int itcount;

    while(AsmProcessing) ;
    MainProcessing = true;
    image = captureImage.clone();
    featureCoords = new int[2 * featurePoints[0] + 1];
    *featureCoords = featurePoints[0];
    memcpy(featureCoords, featurePoints, (2 * featurePoints[0] + 1) * sizeof(int));
    delete featurePoints;
    candideParameters1 = new double[LevMarMirror::m1];
    candideParameters2 = new double[LevMarMirror::m2];
    memcpy(candideParameters1, parameters1, LevMarMirror::m1 * sizeof(double));
```

```
    memcpy(candideParameters2, parameters2, LevMarMirror::m2 * sizeof(double));
    delete parameters1;
    delete parameters2;
    itcount = iterations;
    MainProcessing = false;

    QPixmap Pixmap1 = QPixmap::fromImage(QImage(image.data, image.cols, image.rows,
QImage::Format_RGB888));
    ui.LabelImage->setPixmap(Pixmap1);
    for(int i = 0; i < featureCoords[0]; i++)
        feature.push_back(QPoint(featureCoords[2 * i + 1], featureCoords[2 * i + 2]));
    setWindowTitle("MirrorTrial - " + QString::number(featureCoords[0]) + " Features, "
+ QString::number(itcount) + " Iterations");
    ShowFeatures(feature);

    std::vector<double> v;
    FaceModel.getAllParams(v);
    for(int i = 0; i < LevMarMirror::m1; i++)
        v[LevMarMirror::CandideParams1[i]] = candideParameters1[i];
    for(int i = 0; i < LevMarMirror::m2; i++)
        v[LevMarMirror::CandideParams2[i]] = candideParameters2[i];
    FaceModel.setAllParams(v);
    FaceModel.updateImageCoords(3, 480, 360);
    faceImage.setTo(CV_RGB(255, 255, 255), cv::Mat::ones(360, 480, CV_8UC1));
    FaceModel.draw(faceImage, CV_RGB(0, 255, 0));
    Pixmap1 = QPixmap::fromImage(QImage(faceImage.data, faceImage.cols, faceImage.rows,
QImage::Format_RGB888));
    ui.LabelFaceModel->setPixmap(Pixmap1);
}

void MirrorTrial::AsmFinished()
{
    emit ImageGrabbed();
}

*******************************************************************************************
*******************************************************************************************

//Functionalities needed for Levenberg-Marquardt algorithm

#include "LevMarMirror.h"

#define TWO_ROUND

#ifndef TWO_ROUND
int LevMarMirror::m1 = 17;
int LevMarMirror::m2 = 0;
#else
int LevMarMirror::m1 = 9;
int LevMarMirror::m2 = 8;
#endif

int LevMarMirror::n = 74;
int LevMarMirror::AsmPoints[] = {14, 0, 11, 3, 10, 4, 8, 6, 7, //9 points
                                 15, 16, 18, 21, 22, 24, 20, 26, 67, 41, 43, 39, //12
points
                                 44, 38, 42, 40, 47, 46, 54, 48, 51, 57, 61, 64, 63,
65, 62, 60}; //16 points
int LevMarMirror::CandideVertices[] = {29, 62, 28, 61, 30, 63, 32, 65, 10,
                                       15, 16, 17, 48, 49, 50, 18, 51, 5, 6, 26, 59,
                                       92, 93, 111, 112, 75, 76, 31, 64, 7, 8, 40, 87,
81, 82, 83, 84};
double MatchWeights[] = {1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
                         1.35, 1.35, 1.35, 1.35, 1.35, 1.35, 1.35, 1.35, 1.35, 1.35,
1.35,
                          1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.25, 1.25, 1.25, 1.25, 1.25, 1.25,
1.1, 1.1, 1.1, 1.1};

#ifndef TWO_ROUND
int LevMarMirror::CandideParams1[] = {0, 1, 2, 3, 4, 6, 7, 12, 23, 9, 13, 14, 15, 17, 18,
19, 29};
int LevMarMirror::CandideParams2[];
#else
int LevMarMirror::CandideParams1[] = {0, 1, 2, 3, 4, 6, 7, 12, 23};
int LevMarMirror::CandideParams2[] = {9, 13, 14, 15, 17, 18, 19, 29};
#endif

#ifndef TWO_ROUND
double LevMarMirror::p1[17];
double LevMarMirror::p2[];
#else
double LevMarMirror::p1[9];
double LevMarMirror::p2[8];
#endif
```

```
int LevMarMirror::LevMarMirroring(int *features)
{
    double *x = new double[n];
    for(int i = 0; i < m1; i++)
        p1[i] = 0.0;
    for(int i = 0; i < m2; i++)
        p2[i] = 0.0;
    for(int i = 0; i < n; i++)
        x[i] = features[2 * AsmPoints[i / 2] + i % 2] * MatchWeights[i / 2];

    eruFace::Model tempModel;
    tempModel.read("candide3_.wfm");
    int it1 = dlevmar_dif(candide1, p1, x, m1, n, 500, NULL, NULL, NULL, NULL, (void *)
&tempModel);
    int it2 = dlevmar_dif(candide2, p2, x, m2, n, 500, NULL, NULL, NULL, NULL, (void *)
&tempModel);
    delete x;
    return it1 + it2;
}

void LevMarMirror::candide1(double *p, double *x, int m, int n, void *data)
{
register int i;

    eruFace::Model* CandideModel = (eruFace::Model*) data;
    std::vector<double> v;
    CandideModel->getAllParams(v);
    for(i = 0; i < m; i++)
    {
        v[CandideParams1[i]] = p[i];
    }
    CandideModel->setAllParams(v);
    CandideModel->updateImageCoords(3, 480, 360);
    for(i = 0; i < n; i++)
    {
        x[i] = CandideModel->imageCoord(CandideVertices[i / 2])[i % 2] * MatchWeights[i
/ 2];
    }
}

void LevMarMirror::candide2(double *p, double *x, int m, int n, void *data)
{
register int i;

    eruFace::Model* CandideModel = (eruFace::Model*) data;
    std::vector<double> v;
    CandideModel->getAllParams(v);
    for(i = 0; i < m; i++)
    {
        v[CandideParams2[i]] = p[i];
    }
    CandideModel->setAllParams(v);
    CandideModel->updateImageCoords(3, 480, 360);
    for(i = 0; i < n; i++)
    {
        x[i] = CandideModel->imageCoord(CandideVertices[i / 2])[i % 2] * MatchWeights[i
/ 2];
    }
}

*******************************************************************************
*******************************************************************************
```

DEPARTMENT OF COMPUTER SCIENCE

Virtual face mirroring

Bingwei Wang

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Science in the Faculty of Engineering

---

September 2011 | CSMSC-11

Declaration

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Science in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Bingwei Wang, September 2011