Real-Time Facial Animation for Untrained Users

Lander King

Master of Computing in Computer Science and Mathematics
The University of Bath
May 2017

Real-time Facial Animation for Untrained Users

Real-time Facial Animation for Untrained Users

Submitted by: Lander King

COPYRIGHT

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Computer Science and Mathematics in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed:

Abstract

In this final year project, I have developed a user independent real-time facial animation system that can animate a 3D model of a face, by passively tracking points on your face using a RGB webcam. This system requires no training to be completed before they can use the system. I propose a novel way of doing this by trying to match the passively tracked points of the user to a sum of a 16 previously defined expression shapes, by using a trust region method to find the best variable to get the sum that is the closest to the tracked points. It then applies the variables to find the sum of a set of 3D models that match up with the expression shapes. The final sum will now represent the original facial expression observed by the system. To find a good method for finding the variables of the sum, I implemented 3 different methods to test and compare against. The most efficient method of the proposed methods finds the correct sum of expression shapes with an average error of 10 pixels per tracked point. The system calculated this all within 5 to 10 milliseconds per frame. With this accuracy and performance of the system, as well as the low requirements of only needing a webcam to use it, the system becomes open to a wide array of uses. Once such use could be using it in collaboration with general computing tasks such as social networking and as an input to video games.

Contents

List of Figures

Acknowledgements

I would like the thank my project supervisor Dr Darren Cosker for the useful advice and support he has given me throughout the year while I was undertaking this project. I would also like to thanks my family and university friends for the all if the support they have given me throughout the year.

Chapter 1

Introduction

Animation is the process of making an illusion of motion or change. This is often done by a rapid display of sequence of still images that only slightly vary from each other. This method of animation has been used since the 1800s with devices like flipbooks and Zoetropes, which then later evolved into hand-drawn animation. This has been used for animated motion pictures since the 1900s, by having the sequence of the images that are traced onto acetate which was then used to colour the images. Each of the images on acetate were then photographed by a rostrum camera.

Hand-drawn animation or traditional animation required the artists and animators to draw every frame of the motion picture, hence requiring a lot of work even for a relatively short film. Once computers became fast enough, the solution to the large amount of work was to use said computers to assist with the animation. Computer animation allows some or all of the workload to be passed over to the computer to work on. Additionally, computers allowed us to create computer generated images (CGI) which in turn allows us to change from only working with 2D images to the choice between 2D and 3D animations.

1.1 Computer Animation

Computer animation is now used in many modern movies and video games. The basic background of computer animation still draws from the principles used in traditional animation as stated by (Lasseter), who proposed 12 standard animation principles and how they apply when 3D computer animation is done.

Computer animation has various techniques that are used for the creation of animations. The main three that are used are Key-frame, procedural and Motion Capture/Performance Capture animation.

Instead of drawing each frame of the animation, key-frame animation allows the animators to reduce the work load by allowing a computer to calculate the in-between motions between 2

frames drawn by the animators by using interpolation. This gives the animators a good control over the overall animation while also making it less tedious to draw. This technique still requires a lot of artistic skill to draw good keyframes that link well together and the fact the animator may still need to adjust the automatic interpolation generated by the computer.

Procedural animation on the other hand is animation that is generated by a set of rules with some randomness. This is often used but not limited to particle systems and character animations. This form of animation allows for more variable and detailed animations compared to predefined animations.

In more recent years' motion capture has been used as method of generating animations. This method allows us to capture and record performances of actors, giving us a more realistic image of the possible animation, while also reducing the amount of work and time required by the animators to produce a final animation.

1.2 Motion Capture

Motion capture, as explained above, allows us to use real world data for the animation of digital characters. There are two main ways the motion capture is done.

The first being active tracking, in which we track physical markers that a placed on key locations on the actors. The 2D or 3D vector positions of these makers are then used to control the corresponding points on a digital model. This method has the problem of the actors needing to be set up in suits that are covered in markers, additionally these markers can be obscured from the capture device, making the final animation less accurate due to the method acquiring less than the required amount of data to produce an accurate result. Active tracking is not readily available to the public as it requires expensive specialist hardware and software.

The alternative of active tracking is passive tracking, this uses computer vision techniques to detect digital landmarks that are a digital alternative to the physical markers used in active tracking. Although this method removes some of the problems with active tracking, it introduces problems with how you end up generating the tracked positions and how they translate to the model you are trying to animate.

No matter the tracking method used you often have problems with the captured data being somewhat noisy and often only represents a partial solution of the desired final animations, hence some cleanup of the animation generated by the motion capture is needed. This motivates the need for a motion capture system that has minimal noise and give a better generation of the desired animation.

Motion capture can be further split up into two sections, the capture of full body motions and the more refined and subtle motions of an actor's face. Each of these have different subproblems to deal with to generate more detailed and accurate animations. The capture methods used for full body performances are mainly done with active tracking methods, while tracking the fine detail of an actor's face has a variety of different methods that use active or passive tracking methods some of which are explored below.

1.3 Facial Animation

The idea of using motion capture to track facial landmarks that are used to create realistic animations of an actor's facial expression on a digital model is a widely used technique for modern movies and video games. This can be achieved by using active or passive tracking of facial markers with a head mounted camera(s). As with standard motion capture the data collected from the facial capture is then transferred to a model to be animated. This is often an offline process compared to a useful real-time process.

A more readily available method to facial expression capture would allow the general public easier access to the technology and thus use it in many ways, such as input to a video game and video calls to friends and family. It could also be used in the production of independent video games and movies where they don't have access to the expensive systems that the current studios use.

Other more practical solutions to real time facial animation have been developed using a passive tracking method by using RGBD cameras instead, such as the Microsoft's Kinect camera (Weise et al., 2011). But this is limited as conventional users normally only have access to a standard RGB web camera/ camera.

With the increasing image quality of general purpose cameras and computational power of consumer hardware, these techniques can be adapted for consumer-level use, only requiring a single RGB camera for input. (Cao et al., 2013) (Weng et al., 2013). These solutions often work by taking 2D landmark position of the input face and matching them to a 3D blendshape model which is then used to control the model of the character.

These solutions to real time facial animation give accurate results, but have some problems that could reduce the accuracy of the output, such as the current lighting of the person's face. I also find that both these solutions require a per-user training session before the system can be used. This increases the time required for the setup of the system, which I believe is unnecessary time that could be avoidable.

(Cao, Hou and Zhou, 2014) have recently developed a system that can detect facial expressions to a higher accuracy than previous real time solutions without the need for time consuming per person training.

This system still has its limitations, as mentioned in the conclusion of the paper related to the system. Including the problem of occlusions to the face with prolonged occlusion causing the system to lose overall accuracy (Cao, Hou and Zhou, 2014, p.9). However, by implementing a new way for facial tracking and animation, the system has lost performance compared to

previous solutions, meaning it couldn't be used on devices with a lower computational budget or/and lower image quality like a mobile phone or tablet.

Hence, being able to have a performance based facial animation system that has a low computational time and high accuracy tracking, without the need for time consuming per person setup, is useful for both consumer level applications and for the movie/video game industry applications.

1.4 Project Objectives

The primary objective of this project is to develop a facial animation system that can animate a 3D model of the face, with only the use of the user's facial expressions. For the project to be successful, the developed system will have to attempt to complete the following secondary objectives.

- **Accuracy:** The system should accurately give the desired facial animation
- **Performance**: The system should have a minim cost in computing performance.
- **User Accessibility**: The system should be able to work for any user

Chapter 2

Literature Survey

This project proposes the development of a facial animation system that provides the output of a computer animated model of a face by only using your face and your facial expressions as the input. Before the development of such a system can begin it will be useful to carry out research into the area of facial animation and facial performance capture to see how this problem has been tackled previously, to help me fully recognize the problem and to ensure my solution is worthwhile and relevant within the field.

One of the main problems that has been addressed is the need for a way to capture and then represent the face and it expressions within the system, so we can run the appropriate algorithm to get the desired output. Hence it makes sense to examine the different techniques for this, highlighting the benefits and problems that arises with each technique.

Continuing from this main problem, it is generally understood that the system will require a method of going from the digital representation of the face and its expressions to an animated mode, this is often done via some form of retargeting. The retargeting method will generally depend on the method used for capture and representation chosen. Examining the benefits and problems with the retargeting methods while help the best way to approach the problem, as this retargeting step is where most of the computation is needed.

It will also make sense to examine past and existing attempts of facial animation and facial performance capture, both researched based and readily available commercial solutions. To see how they approached this problem and why they did so. By analysing these it will identify popular solutions to the problem at hand while highlighting the problems not yet solved by them. Hence could show a slightly alternate approach to the problem that if attempted would make this a worthwhile project to pursue.

At the end of this research, I should be able to describe what core underlying features are needed for a facial animation system to run and have a clear classification of how the proposed system compares to the previous developed systems.

2.1 Acquisition and Capture

Throughout the years many different performance capture system have used different way of capturing the highly detailed expressions a human face can make. This generally ranges from using a single RGB or RGB-Depth (eg Microsoft Kinect) cameras to large multi camera arrays.

To begin with, systems that use single RGB cameras include, they use this method for capture as it is the most general and simplest way of capturing the required data, this is also a readily available way of capturing visual data as most people these days have a mobile phone with a RGB camera. This option is often the most portable and inexpensive way of doing the capture and allows the camera to be attached to an actor during their performance, where with other methods doing this would have extra complexity. When using this method for capture it is often common is use more techniques to help with the tracking of facial expressions which will be explained further in the next section. This method of capture is use by not limited to , (Weng et al., 2014) (Cao et al., 2015).

The natural extension of a using a single RGB camera is to extend the number of cameras you used for the capture. This has been employed in some high resolution performance capture systems (Bradley et al., 2010), (Beeler et al., 2011). In (Bradley et al., 2010) they use 14 high definition cameras arranged in 7 binocular stereo pairs, of which are zoomed in on a small patch of the face capturing it in high detail. In contrast (Beeler et al., 2011) can use any n differing viewpoints for the capture. Both of these methods use uniform illumination to make sure the face being captured is nicely illuminated before being use in there retrospective system.

It is stated in (Bradley et al., 2010) the use of the 7 binocular pairs allows the capture process to be simple and automatic without the need for separate geometry scans or excessively expensive hardware. As it uses any high resolutions it doesn't require any active methods for tracking that it states can be uncomfortable for the actors using it affecting their performance. A downside of this type of system is that it's not very portable so can't be used in conjunction with body motion capture performances which is often the case for animated films and modern video games.

An alternative to using more than one cameras is to use a single RGB-Depth camera that can convey depth information alongside the standard colour data. With this depth data, it should be possible to get more accurate information to guide the animation of a model. A popular RGB-Depth camera used it the Microsoft Kinect, which simultaneously captures a 640 x 640 colour image and corresponding depth map at 30 Hertz, this is computed via triangulation of an infrared projector and camera. This capture is used in multiple facial animation systems but not limited to (Bouaziz, Wang and Pauly, 2013),(Weise et al., 2011). A Kinect system is used as it is stated as being a low-cost acquisition device with ease of deployment and

constant operability in a natural environment. These advantages come at a price of a significant degradation in the quality of the captured data compared to a capture methods used in state of the art performance capture systems based on high definition camera areas and marker based systems. this is as this capture system is inherently low resolution and has a high amount of noise.

The capture of the face does not need to be limited to only visual data, it's also possible to capture an actor's vocal performance. This additional data could then be used to enhance the system when visual data from the capture of the face is to not sufficient enough to run the latter algorithms. This has recently been implemented in (Liu et al., 2015)

In this paper they propose a facial animation system that require no user-specific training with the use the combination of a RGB-D Kinect sensor and audio as input. With the colour and depth input it reconstructs 3d facial expressions and 3d Mouth shapes. It also employs a DDN acoustic model to extract Phoneme State Posterior Probabilities (PSPP) from the input audio frames. It then uses a lip motion regressor to refine the 3d mouth shapes on both the PSPP and the expression weights of the 3D mouth shapes. Then the final output to computed by combining the mouth and face model.

By using more than one input data type this implementation of a facial animation system solves some of the problems that occurs in other solutions, such as this system being robust to occlusions to the mouth, head rotations. Additionally, instead of using just speech synthesis for the mouth animation the image data can help get accurate results when background noise is present.

The main drawbacks of this implementation as stated in the paper is that the training of the lip motion regressor takes a long time and it also assumes the correlation between the audio input and the mouth shape is user independent. But everyone has different characteristic when pronouncing the same speech, hence make this system less general than desired. This could be avoided by using user-specific lip motion regressor but then the system loses the ability of being used by a general user.
Overall there is good selection of different captures methods that my system could employ, they all have their own benefits and drawbacks. For whichever a system chooses to use they would need to develop their system so it can use these benefits to their full potential and to minimise the effect of the drawbacks.

2.2 Facial Feature Tracking

From the visual data gathered from in the acquisition stage, the system will need to have a method of finding and tracking the important part of the face that will then be used to drive /generate the animated model. There have been many attempts to find the ideal way of

getting this data, but often resort to a similar method of tracking/ finding landmarks upon the face, of which can be detected automatically with a computer algorithm, or by using a physical method. Some systems like (Bradley et al., 2010) don't use this step as they start building a model straight from the visual data from the acquisition stage and can then track the required geometry.

Physical/Active methods often require some form of set up before the capture of the face can being, this include but not limited to facial markers and facial make-up/paint. These can be used to accurately get 3D data of a face from just some 2d visual data. In (Ravikumar et al.) they use a combination of a combination of both facial makers and sparse make-up patches between these markers. As explained in the paper just using facial makers does not capture the complete fidelity of a face and using a larger number of markers can cause problems such as occlusion, swapping and merging of markers, while still not getting the complete fidelity. //add more here

The most basic of the digital attempts of tracking facial features is by finding N number of 2D landmarks on an image of a face, the landmarks are often are put on the key parts if a face eg nose, mouth, eyes ect. With a larger number of landmark positions, we can a represent a larger set of facial details with more detail, but this would come at a cost of increased cost in computation as we would be solving for a larger set of data in later steps of the system. The Tracking of these points is often done via some form of regression model. The C++ library dlib uses the implementation of the method described in (Kazemi and Sullivan, 2014), Which can perform feature tracking of 194 landmarks in milliseconds with an accuracy better or equivalent to other state of the art methods . It does this by using a cascade of regression functions that updates the current estimate of the facial landmarks. For $S^t = \left(x_1^T, x_2^T, .., x_p^T\right) \in \mathbb{R}^{2p}$ denoting the current estimate of p facial landmarks in image I and for every regressor $r_t(.,.)$ in the cascade it finds an update vector from the S^t and I, that is then used to update the current estimate accordingly.

$$S^{t+1} = S^t + r_t(I, S^t)$$

The Regression model is trained using a set of pre label image (I_1, S_1) ,..., (I_n, S_n), For the initial estimate of facial landmarks used in this method is the average of training data's, scaled and centred onto to a bounding box given by face detector, which in turn means the output is guaranteed to be within the linear subspace given by the training data. It is then shown that the method trains each r_t using a gradient tree boosting algorithm.

2D landmarks can give a good estimation of a facial expressions but depending of the extent of the model trained that can lack in detail when observing non frontal faces and is prone to errors when occlusions block the specific facial detail it's tracking. 2D landmarking are used in an initial step in (Cao et al., 2013) before using regression to obtain a 3D facial shape which can then be used later in the system for the generation of the animated model.

For this project I could employ any of these methods of facial feature tracking, preferably focusing on 2D landmarking because of the reasons stated previously. but as my project is focused on the animation of a model not the face and facial expression recognition it would be out of the scope of the project to implement one of these methods from scratch, but this could be an interesting avenue for further projects and research.

2.3 Generating the Model

With these tracked positions of an actor's facial features, the system will need a way of using these points and some of the data from the acquisition step (eg Depth data from a RGB-D camera). As shown in the following implementations of this part of the system, this process can be a fully online algorithm that require no setup, or one that requires a training of a model or some other pre-process to help the algorithm.

In the High Resolution facial performance capture system (Bradley et al., 2010) which used a multi camera array for the capture method described previously. To then generate and animate a high-resolution model of the actor's face. To generate the required high resolution mesh for each of frame of a video, they perform binocular stereo for each of the 7 facial regions that have been captured, producing 7 overlapping depth images. These depth images are then merged into a single mesh using a multi-view reconstruction system by using an iteratively constrain binocular reconstruction and pair merging described in the paper. This produces a mesh with approximately with 1 million polygons for each frame.

With these per-frame meshes, it the system then reconstructs the motion of the face by tracking geometry and texture over time. They proposed to use an optical flow based approach for the motion reconstruction, that works by choosing a singular reference mesh then to compute a mapping between it and all other frames via a frame propagation method. It also computes a per frame high quality texture map that will capture the appearance changes like sweating and wrinkles.

Optical flow is a method to detect the discrete image displacements between a sequence of ordered images caused by the movement of the object or camera. This is done by considering a pixel I(x,y,t) at time t in the first frame, then in the next frame is would have moved by (dx,dy) after a time dt so we can say

$$I(x, y, t) = I(x + dx, y + dy, t + dt)$$

By then taking the Taylor approximation, and removing common terms we get the following

equation.

$$I_x u + I_y v + I_t = 0$$

Called the Optical Flow equation, where I_x and I_y are the image gradients and I_t the gradient against time. This is an equation where u and v are unknown variables so with need to be solved for, but as this can only be solved with some other set of equations, given by some extra constraints. these constraints depend on the chosen method of solving for the estimating the flow. In this paper, they use the Lucas-Kanade method for solving for this equation.

To deal with some of the inherent problems of optical flow like geometry drifting caused by small errors accumulating over time and the errors induced by rapid deformations like lip movement, the system uses explicit mouth tracking and textured-based drift correction. Overall this paper achieves what it proposed to accomplish, but as shown in the results the system does multiple problems that could be addressed with further development, these problems include the fact that very fast movement can lead to incorrect geometry tracking due to motion blur and inaccurate optical flow, also an error in tracking has the possibility of causing an early termination of the face sequence. This system also requires a manual process required for one frame in the sequence to make a reference mesh used for the propagation step.

This system can make detailed animated facial model based upon the face of the actor but compared to my proposed project in which need to animate a model of some target character, this could be done by retargeting the animation of the produced model to the target model, but this would add increased complexity.

In all of the more recent real time solutions for facial animation, they use blendshape models to aid the modelling of the desired animation from the tracked positions. Hence it would be useful to research the what and how blendshapes work and how they are useful for a facial animation system.

2.3.1 Blendshape Models

Most of the recent papers in this area use blendshapes in some part of their solution. This is because using blendshapes or Morph target animation is a common way of doing per-vertex animation of facial animations.

It works by storing a neutral expression of a model and some extra target deformations/expressions, then to animate a face we can then morph/blend between the base and one or more target deformations.
A model can have any number of blendshapes defining unique expressions of a face B =

$[b_0,...,b_n]$, with b_0 being the base and b_1 to b_n are the additional expression blendshapes. Then we can define any new facial expression as a linear combination of these expressions.

$$B = B_0 + \sum_{i=1}^{n} e_i B_i \quad (1)$$

where **e** is a vector of expression coefficients, with each coefficient e_i $1 < i < n$ bounded between 0 and 1,with all weights summing to a total of 1.

In facial animation systems that use a blendshape model they we want to be find these expression coefficients via some form of optimisation algorithm from the data given in the previous steps. Once we have these coefficients we can now animate any new model with the same basis of expression blendshapes.

It is often in facial animation systems that the used blendshape model is based on the Facial Action Coding System (FACS) using a basis of 46 expression blendshapes (Cao et al., 2013). FACS encodes any anatomically possible facial expression which can be deconstructed into specific Action Units (AUs) and their temporal segments that produces the expression, where AUs are the fundamental actions of individual muscles or groups of muscles.

In a facial animation system we could use a generic blendshape model but as shown in some papers it produces a less accurate facial animation results, this is because most people's FACS basic facial expressions and face shape doesn't perfectly match up to the generic blendshapes defined in the model.

As done in (Cao et al., 2013) , (Weng et al., 2014),(Weise et al., 2011) it is possible to solve this problem by training a user specific blendshape model as a onetime prepossessing step before the main system can run. By using this solution to the problem, we end up limiting the solution to only trained users, and increasing the setup time needed before the system can be used for facial animation.

The system developed in (Weise et al., 2011) that uses user-defined blendshapes in an attempt to achieve better results. To customise the generic blend shapes, they recorded a pre-defined sequence of example expressions for each user with the kinect. To overcome some of the noisiness of the kinect data the user was asked to perform slight head rotations while in a neutral expression. These example expressions are then reconstructed using a linear PCA morph-able model combined with non-rigid alignment methods with additional texture constrains in the mouth and eye regions to produce high quality meshes for each of the recorded expressions. They then pass these meshes, a generic blendshape model and

approximated blendshape weights for each expression into the example-base facial rigging system proposed in (Li, Weise and Pauly, 2010) to generate the desired user specific blendshapes model that represents the example expressions.

An alternative method of producing a user specific blendshape model is used in (Cao et al., 2013). In which they the record the specific user doing different head poses at differing angles and 15 different facial expressions at 3 different angles, in total capturing 60 RGB images for each user. After the capture these images processed to detect 75 2D landmark positions. They then use these setup images with the assist of FaceWarehouse, which is a 3D facial expression data base containing data of 150 individuals, with 46 FACS blendshapes each. Then a bilinear face model with identity and expression attributes is built from the database. Then any facial expression can be approximated by the tensor contractions

$$F = C_r \times_2 \boldsymbol{w}_{id}^T \times_3 \boldsymbol{w}_{exp}^T$$

Where C_r is a rank-three core tensor, w_{exp}^T and w_{id}^T are vectors presenting the expression and identity weights respectively? From this they use an iterative two-step method to compute for the blendshapes. which can be solve for a transformation matrix M_i and vectors w_{exp}^T and w_{id}^T by minimising for each image i. This matrix maps this tensor model to the landmarked positions in the setup pictures.

$$E_d = \sum_{k=1}^{75} ||\Pi_Q(M_i\,(C_r \times_2 w_{id}^T \times_3 w_{exp,i}^T)^{(v_k)}) - \boldsymbol{u}_i^{(k)}||^2$$

where $u_i^{(k)}$ if the position vector of the k'th landmark in the i'th setup image and Π_Q is a function mapping 3D points into 2D screen co-ordinates. In the second step they refine the identity weights by minimising

$$E_{joint} = \sum_{i=1}^{n} Ed$$

for n training images, with a fixed M_i and w_{exp}^T. These steps are repeated until the results converge. Once these converge they construct the user specific blendshapes $[B_i]$ as

$$B_i = C_r \times_2 \boldsymbol{w}_{id} \times_3 \left(\boldsymbol{U}_{exp}\boldsymbol{d_i}\right), 0 \leq i \leq 47$$

with $\boldsymbol{U}_{exp}$ a transformation matrix for the expression mode in FaceWarehouse and $\boldsymbol{d_i}$ an expression weight vector with only the i'th element equal to 1 and all others being 0.
Each method demonstrated for generating user specific blendshape model generally dependent upon the capture method used in each system, as it limits you to a number of possible ways of doing computing these models. The second method could easily be implemented in the first paper but in doing so we are limiting the method to only a portion of the captured data.

2.3.2 Adapting Blendshapes

An alternative to using user specific or generic blendshapes is to use a Dynamic Expression Model (DEM) based on a set of generic blendshapes. This method is employed in (Bouaziz, Wang and Pauly, 2013), (Cao, Hou and Zhou, 2014). Using a DEM now allows the system to adapt the original generic blendshape model based upon the viewed facial expressions of the actor, meaning anyone can use the system without a pre-processing step for each user. The method used for DEM adaptation can depend on the implementation of the rest of the facial animation system and how the initial facial data is captured.

In (Bouaziz, Wang and Pauly, 2013) that uses a kinect sensor for input data. It defines an adaptive DEM that combines an identity Principle Component Analysis (PCA) model, dynamic expression template and a parameterized deformation model in a low-dimensional representation suitable for real time learning.

With the template blend shape model $\mathbf{B}^*$, The Identity PCA model which can approximate any specific face model in a neutral expression as $\mathbf{b}_0 = \mathbf{m} + \mathbf{Py}$ where $\mathbf{m}$ is a mean face, $\mathbf{P} = [\mathbf{p}_1,\ldots,\mathbf{p}_n]$ is the first n PCA eigenvectors generated from using PCA on a stacked vertex coordinate vectors from a database of face meshes in a neutral expression and $\mathbf{y}$ is a suitable vector of linear coefficients.

An expression transfer operator is defined $\boldsymbol{T}_i^*$ that maps a given neutral expression $\mathbf{b}_0$ to the i'th expression $\mathbf{b}_i$ we obtain this from the transformation needed to go from the neutral expression in the template blendshape model $\boldsymbol{b}_0^*$ to any specific expression $\boldsymbol{b}_i^*$

It also states that to overcome problems that the deformation transfer copies expressions without any details of the particular facial dynamics of the user, they need to employ further deformation fields to each blendshape $\mathbf{b}_i$. This deformation field can be defined as a linear combination $\mathbf{Ez}$, where $\mathbf{E}$ is the *m* last eigenvectors of the graph Laplacian matrix L computed on the 3D face mesh and $\mathbf{z}$ are the *m* spectral coefficients. With all these

components, it allows the parameterization of any facial expression b_i as

$$b_i = T_i^* b_0 + Ez_i = T_i^*(\mathrm{m} + \mathrm{Py} + \mathrm{Ez}_0) + Ez_i$$

Then during the run time optimization algorithm, it then refines the DEM, It adapts the blendshape model by solving for the PCA parameters **y** and the deformation coefficients **z**, while keeping the calculated blendshape weights fixed. by defining the model refinement energy E_{ref} stated in the paper based upon the fitting energy and finding the new parameters and coefficients that minimises a combined energy of every frame that has been captured.

$$\underset{y,z}{\operatorname{argmin}} \sum_{j=1}^{t} \frac{\gamma^{t-j}}{\sum_{j=1}^{t} \gamma^{t-j}} E_{ref}^{j}$$

where t represents the current frame and $0 \leq \gamma \leq 1$ helps define an exponential decay for the frame history so more recent frames have an increased importance. To solve this, they use Gauss-Seidel optimization.

In this DEM refinement step they implement a heuristic that removes the blendshapes that have already been optimized many time that further optimization would be a waste of computation. This helps increase computational performance once a refined blendshape model has been found.

An alternative approach to DEM adaptation is used in (Cao, Hou and Zhou, 2014) in which they only use a RGB camera for the capture method, and produces its facial animation via regression. This paper uses a similar method to blendshape representation as (Weise et al., 2011) where we can represent any new 3D facial mesh as a linear combination of expression blendshapes **B** plus a rotation ≤ and translation t.

$$F = R(Be^T) + t$$

Then as in (Cao et al., 2013) this paper implements the FaceWarehouse database meaning we can represent the expression blendshapes **B** of a certain user as

$$\boldsymbol{B} = C \times_2 \boldsymbol{u}^t$$

where **u** is a user identity vector, and C is the tensor from the database. In the DEM

optimization step of this paper they update this user identity vector and the camera projection matrix of an ideal pinhole camera **Q**, which is used to parameterize the perspective project operator Π_Q used to project a 3D shape onto a 2D surface, as this is useful when trying to solve for the landmarked positions on the face.

To avoid updating the DEM for every frame, the system employs representative frame selection which will take the first L frames for the initial optimization, but afterward it will only use frames for the optimization if it's expression coefficients and rigid rotation parameters are significantly different from the linear space formed by the existing frames. With each new representative frame is given to the optimization algorithm, it first optimizes the identity vector followed by the camera matrix, It then re-fits the shapes vector for all frames within the representative frame set with the new values of **u** and **Q.**

When solving for the new user identity vector **u,** the system fixes Q and the shape vectors of all representative frames and then minimises the total displacements observed in the frames, under some constraints.

Then to optimize the camera matrix, the system fixes the identity **u** and the shape vectors, then generates the 3D mesh F^l for each frame, then optimize the following energy

$$E_{im} = \sum_{l,k} ||\Pi_Q F^{l,(v_k)} - s_k^l||^2$$

where s_k^l are the 2D coordinates of landmark and $F^{l,(v_k)}$ is the 3D coordinates of the corresponding vertex on mesh F, its stated that this equation can be converting into a least squares problem so can be simply computed analytically. This optimization step only solves for Q and u once for each new representative frame, due to the limiting nature of the computational budget.

Over time the number frames in the representative frame set becomes stable, showing that the DEM converges to an accurate model for the observed model, this happen very quickly in their implementation. Some issues do persist in this adaptation method, including the fact that a new representative frame that contain inaccurate 2D landmarks causing the system to become less accurate, but as this is rare occurrence.

As shown when using DEM adaptation in the system it means that as time passes the final result of the system is often closer to the real facial expressions tracked compared to the result at the start.

This finds a happy medium between the generic and user specific blendshape models, as it wouldn't require the system to have an offline pre-processing step to generating the blendshape model while still getting accurate results after the adaptive model has converged.

2.3.3 Real-time Tracking in system that use Blendshapes

(Weise et al., 2011) uses the user specific blendshape model to define a parameter space for the real-time tracking method. They separate the rigid and non-rigid motion estimating the rigid transformation of the user's face before running optimization of blendshape weights. For the rigid motion tracking it aligns the reconstructed frame from the previous frame to the depth map of the current frame by using ICP with point-plane constraints obtaining a translation vector and a quaternion which represents the rotation observed. These vectors are then smoothed as a weighted average, to ensure less averaging happens when fast motion is observed and high-frequency noise is removed in the rigid pose. The means they get a stable reconstruction when no movement is observed and fast motion can still be recovered.

With the rigid pose, the system can now estimate the new blendshape weights for the observed facial expression. To avoid generating unrealistic face poses the system uses a set of predefined blendshapes animation sequences, that define a space of realistic facial expressions. This is employed by formulating the problem as a maximum a posterior (MAP) estimation.

$$x^* = \underset{\mathbf{x}}{\operatorname{argmax}}\ \mathrm{p}(\mathrm{D}|\mathrm{x})\mathrm{p}(\mathrm{x}, \mathrm{X_n})$$

where $\mathrm{D} = (\mathrm{G}, \mathrm{I})$ is the input data from capture with depth map G and RGB image I, **x** is the blendshape weights vector and $\mathrm{X_n}$ the sequence of n previously reconstructed blendshape vectors. In this MAP the prior is defined by $\mathrm{p}(\mathrm{x}, \mathrm{X_n})$ and the likelihood is model by $\mathrm{p}(\mathrm{D}|\mathrm{x})$. To solve this MAP problem, they minimise the logarithm.

$$x^* = \underset{x}{\operatorname{argmin}}[-\ln p(G|x) - \ln p(I|x) - \ln p(x, \mathrm{X_n})]$$

where is probability functions are defined in the paper. this equation can be solved for by using an iterative gradient solver, as the gradients can be computer analytically, which converges with just a couple of iterations.

By using X_n when solving for the current frames blendshapes weights, this is semi-adaptive by making sure the new model isn't going to be widely inaccurate, hence increasing the accuracy of the final output model.

Overall the system developed in (Weise et al., 2011) that runs at a frame rate of 20Hrtz with a latency < 150ms on their test computer, and gives accurate facial animation results using a Kinect sensor to capture 2d image data and 3d depth data. The use of the prior means that the reconstructed pose is credible, even if it not inevitably close to the captured data. the limitations are but not limited to the need for user support during the pre-processing step to aid the landmarking of lip and eye features, to generate the user specific blendshapes and the fact the capture method captures low resolution and noisy data.

In (Cao et al., 2013) the method of the real-time tracking works by using a facial shape regressor, which outputs the 3D position of landmarks in each frame from the video camera and then solving of the transformation matrix M and the expression coefficient vector **a.** This regressor is a Two-level boosted regressor trained on the same data that is also used to produce the user-defined blendshape model. In the run-time regression, the system obtains the 3D facial shape, with the current frame *I* and the previous frames facial shape S' in which is a neutral facial shape for the first frame captured. It does this by transforming S' to it most similar shape S_r from the base shape set $\{ S_i^o \}$ via a rotation and translation M^a, It then uses the most N similar shapes to this transformed shape and are then passes into the regressor. With all the regression results for the N shapes, it takes the median shape as the final shape and then transforms it back the original position with the inverse of M^a to get the facial shape *S* for the current frame.

With the 3D facial shape, the tracking process solves for the rigid head pose represents as a matrix *M* and the non-rigid expressions coefficients **a**. Which can be both solved in the process of minimising the following

$$E_t = \sum_{k=1}^{75} ||M(B_0 + \sum_{i=1}^{46} a_i B_i)^{(v_k)} - S^k||^2$$

where S^k is the 3D position of the k'th landmark in S and v_k is the matching vertex index on the face mesh. As in (Weise et al., 2011) this paper also uses a prior to enhance the tracking of the system, this prior is based on the Gaussian mixture model of **a** and A_n which is a set of the expression vectors of the n previous frames, the prior E_{prior} is then the negative logarithm of the Gaussian probability. By adding this to the previous energy we can solve for M and a via

$$\arg\min_{M,a}(E_t + E_{prior})$$

Which is solved via an iterative two-part approach. The first part uses the expression vector from the previous frame and solves for M which can be done by using singular value decomposition (SVD). With M found they then fix it and then optimise for **a** which can be done using an iterative gradient solver. This two-part method is iterated until *M* and **a** converges.

Overall in (Cao et al., 2013) they have develop a facial animation system that has a robust real-time tacking method that has comparable to other techniques that running at a frame rate over 24 on their testing computer. It is observed that all the generated 3D points are less than 6 pixels away from ground truth with 73% of them less than 3 pixels away. The system has problems when dramatic lighting changes and facial occlusions are observed. One down side of the this systems the required time for the setup and pre-processing, that as stated in the paper can take up to 45 minutes.

An adaption to (Cao et al., 2013) is developed in (Weng et al., 2014), which still uses regression to solve for the problem, but increases the performance by regressing directly to the facial motion parameters from the 2D video frames. It also solves the lighting problem by performing histogram normalization on the appearance vectors in the run-time and training data.

As shown in these methods for the real-time tracking face tracking, often involves splitting the solution into the rigid and non-rigid transformations, meaning any system that does this can reduce the complexity of solving for each of the transformations. In these methods, they both uses a prior based on the facial shapes previously generated to aid the generation of the current frames shape. This shows that it useful to have some part of your optimization stage to help say if your proposed facial shape is likely to be correct. They also both use an Iterative gradient solver when solving for the optimum blendshape expression vector, this indicates it would be useful to understand how these works and how best to implement them.

2.4 Solving for Blendshape Expression Vector

As stated above we need a method for finding the optimum blendshape expression vector, this as shown by the previous solutions is often done by minimising an objective/cost function that compares the currently viewed face and the final output.

$$\underset{\boldsymbol{e}}{\text{Argmin}} \sum \left\| F^j - (B_0 + \sum_{i=1}^{n} e_i B_i)^j \right\|^2 \quad (2)$$

To minimise such a function that doesn't have an easy to compute closed-from solution, we need the use of an iterative solver to minimise the function. From my studies at university and from my research, I have come across two popular types of methods that are used to minimise such functions, the first being the (Gradient) Descent Methods, and the second the Trust Region Methods. Both method have advantages and disadvantages of which are discussed later as they will be crucial to deciding which method I would use in my project.

An alternative to an iterative solver for the minimum is to use principle component analysis (PCA) to help give a closed form solution of the blendshape expression coefficients of which has been used in other solutions to this problem. This will be useful to research if it obtains an equal or better result compared to the iterative methods

2.4.1 Descent Methods

In general descent methods work by computing a descent direction s_n at each step of the iteration which is then used carry out a one dimension search along this direction to find a new iterate with a lower objective function.

The main motivation of the various algorithms of this method is the choice of how the descent direction is calculated and how the search along this direction is carried out. By changing how these are done can have changes on the rate of convergence and the computational cost of the algorithm.

As stated in (Xiong and De la Torre, 2013), the most used optimisation technique is Newton's method. Newton's method works well for smooth functions that have easily computable second derivative. Newtons method assumes for a given smooth function f(x) that it can be approximated by a quadratic function within a neighbourhood of the minimum of $f(x)$. Given an initial guess $x_0 \in \mathbb{R}^{N*1}$ and that the Hessian matrix of $f(x)$ at x_n is always positive definite then the minimum can be found by find the iterate x_{n+1} such that,

$$x_{n+1} = x_n - \boldsymbol{H}^{-1}(x_n)\boldsymbol{J}(x_n)$$

Where $\boldsymbol{J}(x_n) \in \mathbb{R}^{N*1}$ and $\boldsymbol{H}(x_n) \in \mathbb{R}^{N*N}$ are the Jacobian and Hessian matrices evaluated at x_n. Newton's method has the advantages that given the initial guess is sufficiently close to the minimum of the function it will always find the minimum with a locally quadratic convergence rate. As stated in the paper Newton's method has many problems when using it for computer vision problems. These are the fact that in computer vision problems the Hessian of the objective function isn't always positive definite across the function, additionally it requires the objective function to be twice differentiable which is often not the case in many computer vision applications, The hessian could be estimated numerically but this is very

computationally expensive, so isn't optimal for the use of real time system that I am trying to develop, especially if you consider the additional cost of inversing the hessian with a large dimension. Finally, as Newton's method requires the initial guess to be sufficiently close to the minimum for a guaranteed convergence, this limit computer vision problems to give a decent initial guess when using Newton's method but this isn't always possible in these situations.

Many other descent methods exist to try and solve the problems and limitations of Newton's method while trying to achieve comparable convergence rate.

Quasi Newton methods were developed as an alternative to Newton's methods, they can be used when the Jacobian or Hessian of the objective function is unavailable or too expensive to calculate at every step in the iteration. This is done by find a relatively close estimation of either the Jacobian and/or the Hessian matrices. The BFGS quasi newton method, computes the descent direction from a given initial approximate Hessian matrix B_0

$$s_n = -B_0^{-1}\boldsymbol{J}(x_n)$$

Then the next iteration of x_n is

$$x_{n+1} = x_n + \alpha_n s_n$$

Where α_n is a suitable step length to move along the descent direction, found by performing a line search algorithm. The BFGS algorithm then will update the approximate Hessian as so

$$B_{n+1}^{-1} = (I - p_n d_n y_n^T)B_n^{-1}(I - p_n y_n d_n^T) + p_n d_n d_n^T$$

Where $d_n = x_{n+1} - x_n$, $y_n = \boldsymbol{J}(x_{n+1}) - \boldsymbol{J}(x_n)$ and $p_n = 1/y_n^T d_n$. The BFGS method has been shown to have a reliable performance even when trying to optimize for non-smooth objective functions. BFGS still has its limitations as it still only removes some of the computation and memory complexity from Newton's method, meaning it is still computationally expensive to use in computer vision problems that are often solving for large dimensions.

An alternative descent method is proposed in (Xiong and De la Torre, 2013) that is designed to work for face alignment, it tries to solve for some of the problems and limitations of Newton's and other descent methods applied to computer vision problems. They proposed a Supervised Descent Method (SDM) of which learns the descent directions in a supervised method for Non-linear Least Squared functions. It does this without the need of calculating the Jacobian or Hessian matrices. They state that this method achieves state of the art performance when applied to facial feature alignment/tracking. Compared to Newton's

method the SDM learns a sequence of descent directions R_n and bias terms b_n such that

$$x_{n+1} = x_n + R_n\phi_n + b_n$$

Where ϕ_n is the Scale-Invariant Feature Transform (SIFT) at x_n. The method learns R_n and b_n from the training images $\{d^i\}$ and their corresponding facial landmarks $\{x_*^i\}$, by minimising the following

$$\underset{R_n,b_n}{\operatorname{argmin}} \sum_{d^i} \sum_{x_n^i} \left\| \Delta x_*^{ni} - R_n\phi_n^i - b_n \right\|^2$$

With $\Delta x_*^{ni} = x_*^i - x_n^i$, after R_{n-1} and b_{n-1} are calculated we apply it to get the next iterate mean we can revaluate the terms Δx_*^{ni} and ϕ_n^i using the new iteration of the minimum.

As stated in the paper the SDM convergences within the same number of iterations as Newton's method while also not being as susceptible to bad initialization. But in comparison each iteration of SDM is faster, this would be ideal for the real-time system is would like to develop. Although this method is designed specifically for face alignment it can be can be adapted to other computer vision techniques. A problem with SDM is that is that it would require some work for it to be extended to solve for a constrained optimisation problem, which is often the case when searching for the expression vectors of blendshapes as each expression coefficient needs to be between 0 and 1.

2.4.2 Trust Region Methods

Trust Region methods on the other hand build a quadratic model $m_n(x)$ of the objective function $f(x)$ at each step of the iteration, this quadratic model is then minimised within the neighbourhood R_n close to the current iterate x_n where the model is trusted to approximate the objective function well. The sub problem of minimising the quadratic model can be solved efficiently if the neighbourhood used is very simple.

As most trust region method relies on the derivatives of the objective function or a suitable approximation of them, it as stated in the previous chapter means that the computational cost of these methods for computer vision techniques can often be expensive, But there have been some method that have been designed to work without the need to compute the derivates of the objective function.

A derivative free trust region method is implemented in the dlib library, the Bound Optimization BY Quadratic Approximation (BOBYQA) method. The workings of this trust region method are explained in (Powell, 2009). It attempts to find the least value of $f(x),\ x \in \mathbb{R}^n$ such that every element of x is bound accordingly,

$$a_i \leq x_i \leq b_i \; i = 1,2,\dots n$$

For a lower and upper bounds a,b $\in \mathbb{R}^n$ that are defined by the user of the algorithm, The quadratic approximation is such that for m number of interpolation points y_j,

$$m_k(y_j) = f(y_j)$$

Where $m_k()$ is the quadratic approximation on the kth iteration. The exact value of m is to be set by the user of the algorithm, but is advised in the paper to be around 2n+1. It works out the minimum by taking a vector x_k such that f(x_k) = minError! Bookmark not defined. and adding a step length d_k to it such that it has length less than or equal to the defined trust region radius Δ_k, and that $x_k + d_k$ is still within the bounds and not equal to any of the other interpolation points. It then updates the values like so

$$x_{k+1} = \begin{cases} x_k , & f(x_k+d_k) \geq f(x_k) \\ x_k+d_k, & f(x_k+d_k) < f(x_k) \end{cases}$$

It then updates Δ_k and m_k according to new iterate, with m_k being subject to the same constraints as before with an updated set of interpolants points

$$\tilde{y}_j = \begin{cases} y_j , & j \neq t \\ x_k+d_k, & j = t \end{cases}$$

This continues until certain conditions are achieved. For this algorithm to work it requires the user to supply an initial starting point x_0 and an initial trust region radius Δ_1.

The method uses the frobenius norm for the construction and the updating of the quadratic approximation. Which is the norm defined as such

$$\|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2}$$

The exact method of the construction and updating m_k is further explained in the paper, they also explain the methods of picking d_k and the termination conditions are further explained in the paper.

The BOBYQA algorithm used for the finding of the minimum of the blendshape expression vector seems to be ideal for my project, this is as it could find a bound solution for the expression coefficients as is required and it also has a low computational complexity that

converges within a reasonable number of iterations. Although none of the facial animation methods I have researched use a trust region method for solving for the blendshape expression vector, I believe it will be interesting to investigate which method would produce the best results.

2.4.3 Principle Component Analysis (PCA)

PCA is a procedure that allows us to find a set of eigen vectors that forms an orthonormal basis for a set of observed data. It was invented by Pearson, K in 1901 in the paper (Pearson, 1901). The eigen vectors got from the PCA algorithm are often called the principle components of the data. For each of the principle components there is an associated variance that describes how much of the data is described by that principle component. PCA can also help with reducing dimensionality of data by taking only principles components with the greatest variances, meaning it can help when displaying or working with high-dimensional datasets.

With the eigenvectors given by PCA, we can project an observed data vector $x \in \mathbb{R}^n$ to the new basis by multiplying it by the matrix with the eigenvectors as the columns E.

$$X^* = XE$$

PCA can be done either by using singular value decomposition (SVD) on a matrix containing the observed data, or by doing eigenvalue decomposition (EVD) on the covariance matrix of the data.

(Smith, 2002) explains how PCA is computed using EVD. Given a matrix A containing the observations as the rows of the matrix. We find the empirical mean of along each of the columns in A, we then subtract the mean from each row in matrix A to obtain matrix B. It then follows that we can find the covariance matrix C by using B.

$$C = \frac{1}{n-1}\bar{B} \cdot B$$

With $\bar{B}$ the conjugate transpose of B and n begin the number of data observations. With the covariance matrix, we can then find the eigenvectors and their eigenvectors by eigenvalue decomposition such that

$$V^{-1}CV = D$$

With V being the matrix with the eigenvector as the rows of the matrix and D being a diagonal matrix of eigenvalues. With the eigenvectors and their eigenvalues found we can then sort them into a decreasing order of eigenvalues. We can then take the N number of eigenvectors

to represent the orthogonal basis of the projected data points.

By running PCA on a set of blendshapes, we can use the orthogonal basis of eigenvectors calculated by PCA as a new set of blendshapes, meaning we can do the following calculation to find the expression coefficient vector B in terms of the eigenvectors for the current facial expression observed x and the average blendshape $\bar{x}$.

$$B = V^T(x - \bar{x}) \tag{3}$$

This should be quick for a computer to compute as it is only simple matrix vector multiplication. A downside of this method as explained state (Ravikumar et al.) is that using blendshapes defined by the PCA eigenvectors only have a mathematical meaning there is no significance between each of the individual blendshape in the model. It also has the problem of needing to recalculate the eigenvectors if you want to later add any new expressions to the original blendshape model. To be able for this method to work best it will require the user to experiment to try and find the best number of eigenvectors to use in (3) that will obtain the PCA expression vector that has the least amount of error.

When using PCA on a set of data it is often assumed the data points are Gaussian/normally distributed, this is as PCA works best when working on data that is Gaussian. When working on data that can't be approximated well by a Gaussian distribution the eigenvectors found do not properly represent the original data. This assumption would need to be checked if I was to implement this method in my project.

After looking at all these methods for solving and searching for the blendshape expression vector, I can now use this information I have leant of each method and their respective advantages and disadvantages to guide me went it get the design and implementation stage of the project.

Chapter 3

Requirements

Before the design and implementation of the system, I needed to set some requirements for my developed system. These requirements are heavily based upon the research concluded in the literature and technology survey. In my original project proposal, I had laid out some nonfunctional and functional requirements, stating things my project should be accountable for.

The following were the functional requirements that I stated. These requirements are the criteria that my final project solution will have to at least meet for it to be deemed a good project.

- The source code should have comprehensible comments/notes throughout
- The project must use a webcam/ front facing camera as a video input for the real-time face animation.
- The system should work with any webcam on Windows
- The system should be able to run for any user without any training before using it.
- The system should run efficiently enough that it would be portable to mobile devices and still run at a desirable frame rate.
- The system should only use passive methods of facial tracking
- The system developed must work in real time.

These requirements are heavily based on the information gained in the literature review and original goal of the project meaning. These requirements will be the cornerstone used when designing and implementing the facial animation system to make sure it doesn't go off track and stays true to what I originally planned for the project.

The next set of requirements I defined in the proposal were the non-functional requirements, these act as guidelines for how I should go about the non-coding parts of the project. I stated the following:

- Project should be completed by 28th April (final deadline is 5th May)
- The finished project must include a literature review.
- Project supervisor should be informed of the current state of the project weekly.

By following these it means that my project should be well finished in time for the deadline on the 5th of May. It will be important that I try to stick to the requirement that I should have weekly meetings with my project supervisor as it will make sure I always know what and where I should be heading with my project based on the current state of the project I will be doing. As I will have already written the literature review as of writing this section, this requirement is no longer key, it will still be important that I update the literature review whenever I conduct more research, if the opportunity arises while working on the project.

These requirements give a stable backbone for the project to work from, they are reasonable enough that they don't put too many restrictions on the design and implementation aspects of the project.

Chapter 4

Design

For the design for the facial animation system I will need to use the information researched in the literature review and the requirements started in the previous section to guide my design decisions. As is discussed in the literature review there are many sub-sections of the facial animation software that require alternative design decisions depending on the method used and the result the method outputs.

As proposed in the introduction I'm aiming to create an accurate real-time facial animation system, this requires me to design the software as such that the accuracy and the speed of it is as best as it can be. I will also need to make sure I stick to the requirements of the project that were stated in the previous chapter.

Figure 4 in the appendix shows the pipeline design of the software. It will have both a real-time process and an offline pre-process that is used to train a model to be used in the real-time system.

4.1 Real time pipeline

For the developed system to meet the requirement that anybody with a standard webcam or phone camera can use it means that the input to the system is going to be an RGB webcam. By having such a simple method of input to the system, it means that the system will be portable to many devices

With the constant feed from the webcam, the system will take a frame at a time to work with. For each frame from the webcam feed, the system will run a facial landmark tracker on the frame to label N number of landmarks on a visible face. With the current face shape detected the system will then run a method that will find the blendshape expression vector, it will do this with the data from the offline pre-process. To decide which method to use for the final design of the system, I will need to implement a couple of these methods and compare them to find the best one. The methods I have chosen and the method of testing them will be explored in the next chapter.

To then be able view the result of the pipeline, the system will need to apply the expression vector to a blendshape model to obtain the final mesh that represents the facial expression viewed in the current frame. It will then be up to the system to render this final mesh output, this will need to be done with the aid of a graphics library, as it is beyond the scope of this project to implement one.

4.2 Pre-process

Ideally the offline pre-process should be designed so that the user has little or no setup time required for the real-time software to work. For the software to do this we need the pre-process to build a model of the association between the different facial landmark shapes to the desired blendshape expression in the blendshape model. By doing so it means the real-time process can load the model data in, instead of computing the information every time we run the software.

To begin the pre-process, it will require a set of example expression images that will be used to define the expression shapes. This set of images should display all the desired blendshape expressions in the blendshape model. This set of images will be best if it encaptures multiple people doing the same expression, this helps the software to define an average for each of the facial expressions.

These images will then be put through a facial landmark detector to find the facial landmark shape of the expressions defined in the images. By having these shapes, it later enables us to be able to match the facial landmarks that are going to be detected in the real-time system to the ones defined in this pre-process to help find the blendshape expression vector.

To be able to find an accurate and fast method for facial animation, it will require me to test multiple methods to find which one is the best to use. By having the PCA method as one of the methods I will be testing, I will also need to incorporate all the PCA computations that are required to run the PCA method as part of the pre-process step.

4.3 Interface

Although most of the design of the facial animation system is with how the model should be animated, it is still required for the system to have an interface in which it should display. With the aim of the software being able to be used by anyone, the interface of the system should be easy to use and simple to understand. The interface should at least be that the window has different sections showing the webcam feed, the tracked points on the users face and the final animated model. This basic interface can be seen in Figure 5.

Chapter 5

Implementation and Testing

As decided when designing the facial animation system, I choose it to have a real-time process and an offline pre-process that is needed to run before the real-time process can be run. In this section I will cover the specifics of the implementation of both processes. I will also be outlining the methods of testing which will be used to judge the effectiveness of the methods used for the solving of the blendshape expression vector.

To implement the facial animation system, I will use Visual Studio to develop the system in the C++ language. To be able to implement some of the features of the system, I have used some publicly available libraries for C++, the main two being OpenCV and Dlib.

5.1 Pre-process Implementation

The main part of the pre-process is finding the facial landmarks on a set of images of facial expressions that match up to the facial expressions defined in the blendshape model.

In my current implementation, I load up a set of 16 600x600 resolution images that defines a range of different facial expressions, like smiling, puckered lips, raised eyebrows etc. These images also include example rigid transformations like head tilting and horizontal head rotations. Figure 1, shows these images with their respective expression model.

To get the facial landmarks of each expression, I run each image into dlib's facial landmark detector that was looked at in the literature review. This detects 68 landmarks on each of the images, with landmarks 0 to 16, 17 to 26, 27 to 35, 36 to 47 and 48 to 67 being the points representing the jaw line, the eyebrows, the nose, the eyes and the mouth respectively. Then each set of landmarks is saved into an array and written into a file for use within the real-time pipeline of the system. The implementation of the facial landmark detector is to first run a face detector on the image, this will give a bounding box to any faces seen in the image. The bounding box and image are then passed into a dlib pose detector which is trained to find the 68 landmark positions. The found facial landmark shapes for the images are then saved into the file examplesImages68.dat that will be loaded into the system before the real-time pipeline starts.

To define a blendshape model with the same number of expressions as the number of expression images, I load in 16 OBJ files that define the same expressions and rigid transformations. I then load it into a blendshape object that is defined in the header file blendshape.h. The neutral facial expression OBJ is used to define the base mesh of the blendshape, I then load in each of the other OBJ files as the blendshape model expression

meshes. As each of the OBJ files defines a full mesh of the facial expression, I remove the base mesh from all the other meshes to define blendshape expressions as a transformation of the vertices in the base mesh to the vertices in the blendshape expression mesh. The 16 OBJ mesh files that are used to define the blendshape model are some of the blendshapes defined in the Emily model from Faceware Tech. To make the code clearer and easier to extend for more images and expression meshes, I made sure the indexes for individual facial landmark shape and their equivalent blendshape expression were the same.

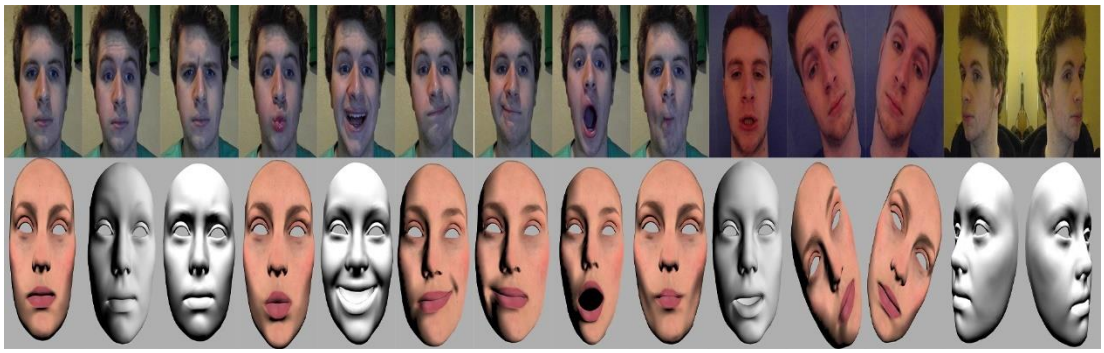


Figure 1. The relationship between the training images used in the pre-process and the models used to define the blendshape model.

5.2 Real-time pipeline Implementation

The implementation of the real-time pipeline is where I have ended up spending most of my time when coding my project, this is because it is where most of the computation of the facial animation system is happening. As stated in the design of system, the real-time part of the system is designed as a pipeline, this enables us to work on the data inputted into the system until it outputs the final animation.

5.2.1 Input and Landmark tracking

The first stage of the real-time pipeline is the input. As was stated previously, the input of the pipeline with a regular RGB webcam. To use a webcam in my software I create and use OpenCV's VideoCapture object, this allows my program to connect to any webcams connected to the computer, as long as it isn't currently being used by another process on the computer.

In the final implementation of my project, the VideoCapture object captures 600x600 images from the webcam. This resolution is used for two reasons, the first being that it is the same

resolution as the example facial expression images used in the pre-process, meaning it will then be easier to compare the facial landmarks of the images. The second reason for this choice of resolution is that with this size of image, the later computations that are run on the images from the webcam will be less expensive to run compared to higher resolution images, in terms of both memory and computational time. This resolution is also high enough that the images from the webcam are high enough detail to have a good detection of faces and their facial landmarks.

With the image data coming from the webcam, the system can now proceed to find if any faces are present in the current frame and if so then to detect the facial landmarks. The real-time process will also use the same facial landmark tracker implementation used in the pre-process. A slight problem I found with the landmark tracker is that the tracked landmarks on the webcam feed where noisy, showing a slight jitter even if the observed face was being kept still between multiple frames. This, as explained previously is a common trait with motion capture techniques. This could have been solved by using a Kalman particle filter, but the importance of this problem was low compared the other problems that needed to be solved in the project.

5.2.2 Expression Vector Solving

With the system now being able to get the input from a webcam and find facial landmarks on the users face, the next avenue of development was to implement the method of finding the blendshape expression vector. As stated before in this stage I will implement and test different methods for this, so I can find the best method to use.

Before this could be done I needed to implement a method of scaling and moving the example facial landmark shapes to the size and position of the facial landmark shape observed in the current frame. To do this I would use the bounding box of the face in the current face and the bounding box of the facial landmarks on the neutral expression shape defined in the pre-process. To find the factor required scaling the example shapes to the current shape, we compare the length and width of the bounding boxes.

Listing 1 Scaling calculation

```
Transform[0] = currentWidth / baseWidth;  // X Scale
Transform[1] = currentHeight / baseHeight; // Y Scale
```

This scaling is then applied to the expression shapes, within the function for finding the expression vector. To find the transformation needed to map the expression shapes to the position of the current shape, we use the coordinates of the midpoints from both bounding boxes and find the difference between them.

Listing 2 Transform calculation

```
Transform[2] = currentMidx - baseMidx; // X translation
```

```
Transform[3] = currentMidy - baseMidy; // Y translation
```

In the original implementation of finding the former transformations I used the full bounding box of the current face, but this caused an unwanted effect of the vertical scaling to become too large when the user raised their eyebrows, meaning the transformed points where no longer in the correct position. To fix this problem I changed the method to only use a bounding box defined by all the facial landmarks except for the landmarks that defined the eyebrow positions.

When implementing the methods for transforming the example shapes to the current shape, I thought it would be useful to add a method of finding the 2D rotation between them, to aid in the finding of the expression vector when the user tilts their head. To do this I used SVD on the familiar covariance matrix of the two shapes, which was done in the function findRotation. When this method worked, it found the correct rotation between the points but it would occasionally move the landmark coordinates off the range of the image, which made the found expression vector and the resulting animation incorrect. In the final implementation of project, I fixed the 2D rotation matrix to always be the identity matrix, to avoid this problem. Instead of finding this rigid transform analytically, I added two new models to the blendshape model that represented a left and right head tilt.

After this was completed I could start to implement the three alternative methods for finding the expression vectors, these being an all in one approach, a region based approach and a PCA approach. The main aim of these methods where to try to solve equation (2) with the minimal error possible within a reasonable time.

The first method I implemented was the all-in-one method which is a simple method to calculate the expression vector by putting eqn. (2) into the BOBYQA trust region method which is offered by the dlib library. This is further explained in the literature review.

Listing 3 BOBYQA applied to blendshape error

```
find_min_bobyqa(minFunction, expressionVector, 33
                , uniform_matrix<double>(16, 1, 0.0)
                , uniform_matrix<double>(16, 1, 1.0)
                , 0.075, 0.01, 30000);
```

This function tries to minimize the given function (minFunction), with the given column vector (expressionVector). By setting the BOBYQA method with 33=2n+1 interpolation, with n being the length of the expression vector, the method should achieve optimum results as stated in the paper explaining the method. I have set the minimum and maximum vector bound as required by the blendshape model, the values 0.075 and 0.01 represent the initial and final values of a trust region radius to be used when optimizing the solution. Once this method has found the minimum it will save it in the given column vector.

I followed this up by resetting the first variable of the expression vector to 1 and reran the bobqqa function with a smaller value for the final trust region radius equal to 0.005, to try and refine and or correct the expression vector found in the first pass of the function. Once this was done we then passed on the expression vector to the blendshape model.

The function minFunction is set to calculate the total error in the using a set expression vector in the blendshape solve.

Listing 4 Blendshape error function

```
double minFunction(const column_vector& m) {

	double result = 0.0;
	//Total L2 Norm of Difference between Estimated Points
	 and Observed points
	for (unsigned int i = 0; i < currentFaceShape.num_parts(); i++) {
		point weightedPoint = calcWeightedShape(m, i);
		result += calcDiffL2NormSquared(currentFaceShape.part(i), weightedPoint);
	}
	//Add L1 Norm to normallaize result (make sparse)
	result += calcL1Norm(m);
	return result;

}
```

The error is calculated by taking the L2 norm of the difference between the current face shape and the sum of the expression shapes for a certain expression vector (2), that is calculated in calcWeighedShape. To begin with I set it to only calculate the L2 norm but the final expression vector was quite dense which wasn't ideal, to try and avoid this I add the L1 norm of the expression vector to the previous error to try and make the expression vector sparser.

The second method for solving for the expression is like the first in that it still uses the bobyqa algorithm to minimize the error function but we solve for a certain region at a time. Instead of running the bobyqa algorithm twice on the same function, we now set it solve for the rigid transformations in the first pass and then for facial expression in the second pass.

Listing 5 Using BOBYQA for Regional Blendshape Solve

```
//Solve for facial rotations
find_min_bobyqa(rotationMinFunction, rotationVector, 9
		, uniform_matrix<double>(4, 1, 0.0)
		, uniform_matrix<double>(4, 1, 1.0)
		, 0.075, 0.01, 30000);
regionExpressionVector = concatColVec(expressVector, rotationVector);
regionSolve = getFinalShape(regionExpressionVector);
//Solve for facial expression
find_min_bobyqa(regionMinFunction, expressVector, 15, uniform_matrix<double>(12, 1, 0.0),
uniform_matrix<double>(12, 1, 1.0), 0.075, 0.01, 30000);
```

For the implementation of this method the bobyqa function needed two different functions to try and minimize, it was also required to use different column vectors for each pass of the function so that in the second pass it doesn't try to alter the fixed rotation found in the first pass. The first minimization function only uses the expression shapes that relate to the rigid rotations of the face. The second minimization function then solves for the expression vector

by only looking at the expression shapes that relate to differing facial expressions using the face shape found in the first pass as the base shape compared to the base neutral shape in the first method.

Listing 6 Calculation of the blendshape sum for the regional solve method

```
point calcRegionShape(const column_vector& m, int i) {

	assert(m.size() == (exampleShapes.size())-4);

	point currentPoint;
	currentPoint.x() = regionSolve[i].x();
	currentPoint.y() = regionSolve[i].y();

	//Check if  the current point is one associated with the eyebrow region
	if (i > 17 && i<= 26)
	{
		for (int j = 0; j < 6; j++) {
			currentPoint.x() += m(j) * (Transform[0] * (exampleShapes[j].part(i).x()
			exampleShapes[0].part(i).x()));
			currentPoint.y() += m(j) * (Transform[1] * (exampleShapes[j].part(i).y() -
			exampleShapes[0].part(i).y()));
		}
	}
	//Check if the current Point is associated with the mouth region
	else if (i<=16 ||(i >= 48 && i <= 68 )) {

		for (int j = 6; j < m.size(); j++) {
			currentPoint.x() += m(j) * (Transform[0] * (exampleShapes[j].part(i).x() -
			exampleShapes[0].part(i).x()));
			currentPoint.y() += m(j) * (Transform[1] * (exampleShapes[j].part(i).y() -
			exampleShapes[0].part(i).y()));
		}
	}

	return currentPoint;

}
```

Additionally, this method of solving for the expression vector only uses the expression shapes that correspond to the current facial landmark Eg, when looking at a landmark that represents an eyebrow it will only solve with the expression shapes that define different eyebrow positions. By doing this it means that the expression shapes that shouldn't have any changes in a set region compared to the neutral shape will no longer have an effect for a set landmark. Hence removing the error caused from the slight differences in the landmark positions in the expression shapes, that is caused by the inconsistencies in the images used in the pre-process. Also by reducing the number of expression shapes that we solve for when finding the expression vector, this will reduce the computational complexity of the method. As with the previous method I employ the L1 norm to make the expression vector sparse.

The third method I am going to implement is the PCA method that was explained in the literature review. To be able to use PCA method to solve for the expression vector, I will first need to setup the data so that I can run PCA on the expression shapes to find the orthonormal basis of eigenvectors that can be used to represent the expression shapes. With the eigenvectors, I will then need to find the 3D models that represent the eigenvectors, to do so I run the eigenvectors through the first method defined in this chapter to find the expression vector of each eigenvector face shape, that are then used to define the new PCA blendshape model. This is all done in the function PCAsetup which uses OpenCV PCA object to do all the required PCA calculations. To set up the expression shapes so they can be used by the PCA object, I needed to reorganize them into a singular matrix with each of the expression shapes being a row in the matrix, with the form;

$$(x_1, y_1, x_2, y_2, \dots, y_n)$$

With (x_1, y_1) being the x-y coordinates of the first landmark in the expression shape. To make sure the system only uses the most important eigenvectors for the PCA blendshape model, I made it so that it will only take 99.9% of the total energy defined by the eigenvalues. We can then find the expression vector by projecting the currently viewed face shape to the new coordinate system defined by the eigenvectors.

5.2.3 Interface and Rendering the 3D model

Once the expression vector has been found by either of the methods, the system will now need a way to display the final model of the current facial expression to the user of the software. As also stated in the design of the interface it will also need a way of displaying the webcam feed and the detected landmark positions.

To be able to achieve this I have used the C++ graphics library OpenGL 3.0. This enables the system to easily draw images and render 3D model in real time to the screen of the computer the system is running on. As shown in the design stage of the project, I required the window to be sectioned off into different sections showing the webcam feed with the tracked landmark and the rendering of the model.

To do this, once an OpenGL window had been created and we are within the rendering loop we could use the function glViewPort to define an area of the window that the program is currently allowed to draw on, called a viewport, allowing it to define the two separate areas of the windows as required.

The first viewport is where the program will show the feed of the camera with facial landmarks on top that are from the landmark tracker and the final expression shape is found by applying the expression vector to the set of examples expressions. To be able to do this I needed to make a rectangle that will fill up the entire area of the current view point, this rectangle will

then have the current frame of the webcam feed set as its texture. Before I could set the texture as the current frame of the webcam, I needed to manipulate the way the frame data was stored. This was because I needed to reformate the cv::Mat file type that the image was stored as within my program, into a suitable format that OpenGL could use as a texture. Once this was done the images could be loaded into a texture and the texture bound to the rectangle.

The second viewport will then be used to display the final model from the sum of blendshapes, but to do so I will need to load each of the models defined in that blendshape model and set it up so it can be rendered with OpenGL. To do so I load the OBJ file using the object loader given in the assimp library, this allow me to define the OBJ file as a vector of vertexes, that include the position vertices, the normal vector at that point and the texture coordinates of each vertices defined in the OBJ file. Once this is done all the relevant data is loaded into Vertex Array Objects (VAO), Vertex Buffer Objects (VBO) and Element Buffer Object (EBO) so they can all be rendered later.

Once all the OBJ of the expression models have been loaded and defined into a blendshape model, then the final model can be achieved by applying the blendshape model to equation (1).

Listing 7 Calculation of the final mesh using blendshapes

```
std::vector<Vertex> blendshapeSolve(Blendshape &blend,const column_vector &w) {

	std::vector<Vertex> blendModel;

	for (unsigned int i = 0; i < blend.baseModel.size(); i++) {
		Vertex newV = {blend.baseModel[i].Position , blend.baseModel[i].Normal,
		blend.baseModel[i].TexCoords };
		for (int j = 1; j < w.size(); j++) {
			newV.Position += (blend.blendshapes[j-1][i].Position * (float)w(j));
		}
		blendModel.push_back(newV);
	}
	return blendModel;

}
```

Then by binding the VAO and the buffers associated with the final mesh from the blendshape we can draw it to the screen

Listing 8 Drawing a mesh to the screen

```
//Draw Mesh
glBindVertexArray(this->VAO);
glBindBuffer(GL_ARRAY_BUFFER, this->VBO);
glBufferSubData(GL_ARRAY_BUFFER, 0, this->vertices.size() * sizeof(Vertex), &this->vertices[0]);
		glDrawElements(GL_TRIANGLES, this->indices.size(), GL_UNSIGNED_INT, 0);

glBindVertexArray(0);
```

OpenGL 3.0 also requires the programmer to define shaders that state what do at certain stages of the OpenGL's rendering pipeline, by default you are required to give at least one fragment shader and one vertex shader that defines how OpenGL should go about the fragment and vertex calculations before you can draw anything to the screen. In the current implementation, I use two types of shaders the first being the model shaders which are used when wanting to draw the models to the screen. The second set of shaders are the lighting shaders which are used when drawing the lights within the scene that provide the illumination for the objects within the scene.

Listing 9 Loading the model shaders

```
Shader modelShader("model.vs", "model.frag");
```

To finish drawing the model to the screen, we need to make sure the model is illuminated by light so that it is visible, and we also need to make sure the model is viewable with the viewport and of a decent size of it doesn't occupy too much or little of the given space. To illuminate the model we use the fragment shader of the model shader that defines the colour of the object according to the Phong illumination model, for a given light direction, viewing direction, light and object colour.

To make sure the object is always in view I needed to implement a projection camera model for OpenGL to use, this would allow me to define what positions I was looking at and from how far away I was able to view it from. To do this I used the model vertex shader to calculate the position of the object on screen.

Listing 10 Screen position calculation in model.vs

```
gl_Position = projection * view * model * vec4(position, 1.0f);
```

Once the camera model was implemented I still needed a way of finding the best place to look at so the full object was in view of the camera. To do so I find a bounding sphere on the object I wanted to look at by finding the centre of the object and the maximum position of the vertices in the object to define the center and radius of a bounding sphere. Then by setting the camera to look at the centre of the bounding sphere and to be at a distance three times the radius away from the center it means that the entire object will always be in full view.

Listing 11 Finding the camera position and look vector

```
glm::vec3 min = ourModel.getMinValues();
glm::vec3 max = ourModel.getMaxValues();
glm::vec3 center = glm::vec3((max.x + min.x) / 2, (max.y + min.y) / 2, (max.z + min.z) / 2);
float rad = glm::length(max - center);

glm::vec3 lightPos(center.x, center.y, center.z + 3 * rad);
```

5.3 Testing

To be able to verify which method of finding the expression vector and hence outputting a better animated face, I will need to test each method on three criteria. The first criteria for being a good expression vector is that it produces the minimal amount of difference in the calculated landmark shape compared to viewed landmarks meaning it is a good method of calculation. The second criteria will be to check how fast each method computes the expression vector meaning it is. The final criteria will be to check that the final animation outputted by the system is expressing the correct expression, this is a less analytical method of testing than the other two but will be crucial in making sure the system produces a good result.

For the trust region methods, I went about testing the first two criteria as so; I inputted multiple 1280 x 720 resolution videos into the system. In each of the videos a person will be performing the same expressions that are used to define the blendshape model, repeating each expression multiple times, returning to the neutral position between each of the subsequent expressions. The system will then record the error and time taken for each of the methods applied and output it to a file that can then be used to analysis each method. Using a fixed video means that I would be able to repeat the system on the same input to make sure the system produces the same result. By having the person perform the expressions in succession it means that in the analysis of the results, if a high amount of error is found I will be able to work out what expression was likely to be the cause of it. By having videos of different people performing the expressions, it means I can then make sure of the requirement that the system will work for anybody without any training is satisfied. Even though the videos are of a higher dimension than used in the real-time system, if the system can calculate the expression vector well in a relatively short time, it will mean that when it is done in the real-time system with the 600x600 webcam resolution it will be even faster. To check for the third criteria I checked, with my own judgement that the final animation was a good enough resemblance of the facial expression expressed in the videos.

It will also be interesting to see how much of a difference in results the system gives when using the L1 norm of the expression vector with the L2 norm in the minimization function, compared to only using the L2 in the minimization function. To test this, I will run one of the videos used before to be able to get the results so I can compare them later on.

I will also need to check the accuracy of the PCA expression vector using a different number of eigenvectors to define the orthogonal basis of expression and the PCA blendshape model. I will also run the same videos used for the trust regions methods to find the average error and the average time the computation takes to complete.

Chapter 6

Results

In this chapter I will be showing, interpreting and evaluating the data gathered from the testing phase of the project, that I described in the previous chapter. With the results of the testing I will finally be able to decide which method I will use for finding the expression vector in the facial animation system. I will also do analysis on any other important results of the facial animation system as it could be useful to understand what and why causes the certain details of the final animation given by the system.

6.1 Accuracy Analysis

As stated in the previous chapter, to test the system multiple videos were put through the facial animation system and it recorded the average displacement error between all the tracked landmarks and the final expression landmarks given by using the found expression vector in the blendshape sum equation for all the frames in the video.

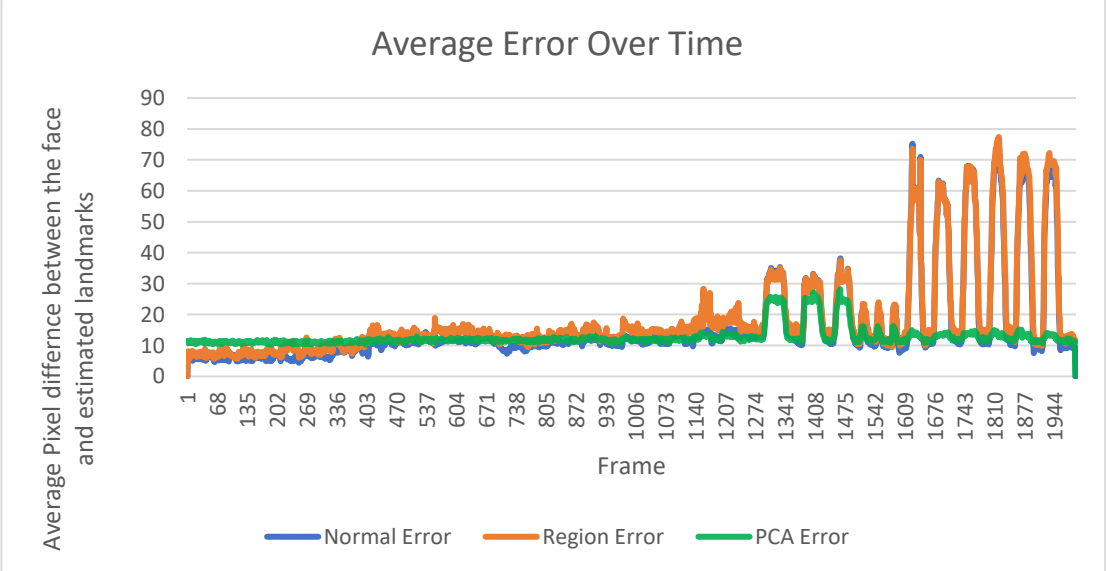


Figure 2 The average error per frame in time video 1

This testing method gave us the results that the All-in-One method has an average error of 10 displacement error per landmark. The region based method however has a slightly higher average error of 11.4, and the PCA method has an average error of 12.1 but has less variance in the error between frames. This means that the All-in-One method finds the better result for the blendshape expression vector.

An interesting observation of the error data is that it has huge spike in the error at the end of the testing videos, this indicates that the methods are bad at finding the correct expression vector for the expressions at the end of the video when given to the system. The expression at this point of the video are the left and right rotations and tilts of the head, the left and right rotation only give a slight rise in the error, where are the tilts of the head heavily increase the error in the blendshape expression vector. You can also observe that the serve increase in error does not happen when using the PCA method, with the highest average error for a frame being 28 which is significantly less than the max errors observed in the other methods. Even when removing these spikes from the data the errors are only slightly reduced. Even though the final expression isn't estimated very well by the system when using 2D landmark positions, when transferring over to the 3D model, it seems to match to the facial expression in the video. The only problem with rigid transformations blendshapes expressions is that occasionally the system can't the correct expression vector when you combine the rigid transformations with the non-rigid ones.

A common theme is that although the region based method outputs a slightly higher error than the other all-in-one method, when applying the expression vector to the 3D blendshape model, it often produces a better animation of the facial expression compared to the other method. The only downside is that region based method also causes the final animation to occasionally have very sudden in jumps in the current expression being shown, but this might be fixable by altering some of the arguments to the bobyqa function, but mainly changing the trust region radiuses that are used.

6.2 Performance Analysis

Other than the accuracy of the result, I also require the chosen method to find the solution to send as little time computing the expression vector. I have recorded the error of each method on each frame as well as recorded the time it took for the computation of the expression vector to finish. The performance of the system was recording using a Surface book pro 4 that has an Intel i5-6300U 2.4Ghz CPU with 4 gigabytes of ram available, the system was also running in the x86 architecture using the streaming SIMD extensions 2 enhanced instruction set that should speed up some of the computations.

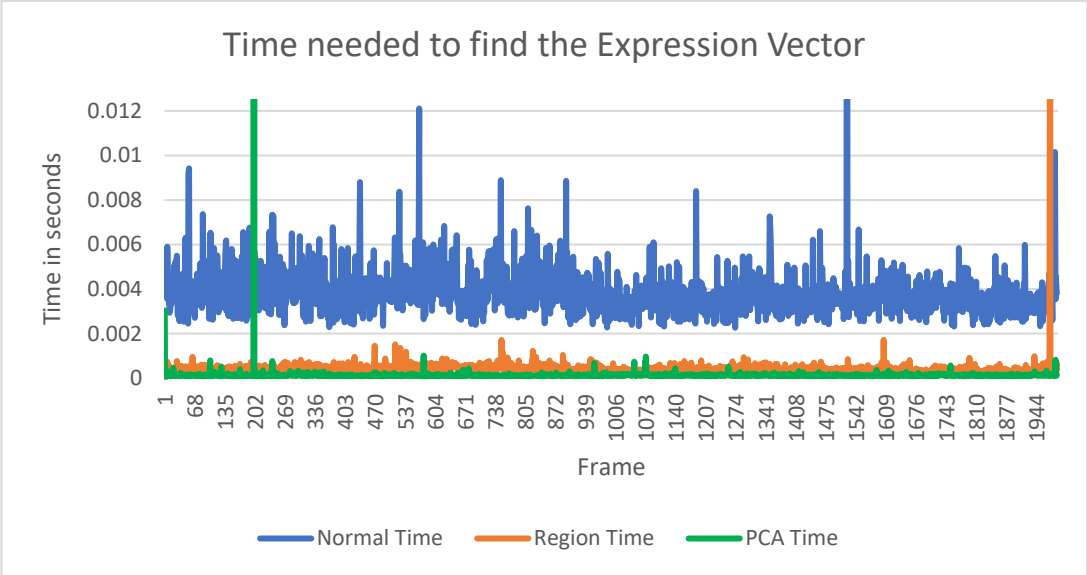


Figure 3. Time to compute the expression vector on each frame of test video 1

As is shown in the figure above, the PCA method has the quickest computation of all the methods with an average computation time of 0.134281ms and is followed by the region based method with an average time of 0.426 ms and then the All-in-One method being the slowest of the methods with an average time of 3.76 ms. This is an interesting discovery as it is the opposite of the results found in the accuracy testing, this means choosing the best method is much harder as all the methods I have implemented are very similar to each, each with slightly different advantages and disadvantages.

As the system isn't just solving the expression vector, it is important to look at the performance of the system when considering the computation of finding the 68 facial landmarks and rendering of the model. In the current implementation of the system it renders 2 models and runs all of the methods for finding the expression vector. It achieves this in an average of 5-10 ms per frame, which is equivalent to 100 to 200 frames per second. This is good performance for the animations results we achieve. It is a lot faster than the other methods for facial animation that I researched in the literature review, meaning that this system should still run if it was ported to a mobile phone. I believe I get this kind of performance compared to methods in the literature review as I am only solving for a relatively small set of blendshapes compared to their larger blendshape models that are a more based on the FACS model.

Another important feature of the performance of a computer program is the amount of ram is used when the programs is running. By minimizing the amount of ram that is used by a program it means it can be used on computer systems that have a limited amount of ram. The current implementation currently uses around 200MB of ram once it has loaded all the data from the pre-process step and the OBJ file used to define the blendshape model, even though this amount ram need is not excessive, it would help if it used less so it could be ported to more ram redistricted devices.

6.3 PCA Results

Even though the PCA method gave the largest error of all of the methods, it manages to consistently match the desired facial expression without much change in the error. It also manages to do this in the shortest time out of all the methods. This would make it ideal for the primary method for the facial animation. This would be the case if the expression vector of the eigenvector blendshape produced a final animated face akin the other methods, but this isn't the case for the current implementation of the system. Instead it produces a face shape that is very unlike the original expression models given to the system. This is what is most likely happening due to an error when the system calculates the expression vector for each of the eigenvectors, which are then used in turn to define the eigenvector blendshape model. This result is disappointing as the PCA method would have been an excellent choice in the method of find the expression vector. I have left the PCA calculation in the source code for the project, so that anyone who want to check the result of the PCA method are able to do so.

6.4 Additional Results

To test how much of an effect the L1 norm has on the final output of the all-in-one method and region method, I inputted test video 1 into the system with the L1 norm disabled to obtain the average error and time it now took to complete. The results of this test were that by adding the L1 norm to the minimisation function the system produces a very similar error for the all-in-one method for when it could find the best expression vector within the number of max iterations given to the bobyqa function, however when it couldn't the program crashed, which happened at frame 108 in the video. The region method also showed the same behaviour but crashing at a slightly later point in the video. This shows that the L1 norm is very useful by the fact that it makes the expression as sparse as we want it, but it also helps reduce the number of iterations needed to find the optimal expression vector, which would increase the computational cost of finding it.

During the implementation and testing stages of the project, I observed some interesting occurrences that happened in my facial animation system. The first being the fact that some of the expression models used were used to define the blendshape model are rarely seen in the final animation even when the person using the software is do that expression, although this is an undesired feature as I wanted the system to be able to do any of the expressions that were used in it to be visible. I believe this is a consequence of the overall method that I decided on using to solve the facial animation. This occurs as some of the 2D landmark shapes of the expressions are very similar to the neutral position or to some of the more common expression that appear in the final animation. I was also able to find that the user would need to slightly exaggerate their facial expression to obtain the desired expression in the animation, this could be a side effect of the images used in the training data as it displays some exaggerated facial expression, that is slightly uncommon in most people's normal facial expressions.

The final observation I have made is that facial animation is limited in its ability to show the correct expressions by the landmark tracker used on the webcam feed. This is as any mistake in the tracked points will also cause an error when finding the expression vector. It also means that whenever the landmark positions fail to be located it means the facial animation system won't do anything until the landmarks are found again.

Conclusions

In conclusion, in this project I have managed to create a facial animation system, capable of animating a 3D face model by only using a RGB webcam and a person's face. This was the main goal of the project I aimed to complete. I have also managed to complete the secondary objective of developing the system so that it produces an accurate result, while also only needing a little computation, meaning it can work fast in real-time. Even though the final system doesn't work as well as the researched system, as this was a topic that I had barely covered in my studies before taking on the project, I believe I have produced a good result given the amount of information I had to learn for my project. Even though the system produces a good result it can still be improved on, by conducting more research and trying to fix the problems with the current implementation.

There were many parts of the project that didn't go as planned, such as the PCA method used to find the expression vector not working as expected when applying it to the blendshape model. Another such problem that occurred, that hindered the project was the problems in the function that found the rotation between the current face shape and the base example shape, causing me to evaluate what I had to get to the end of the project without using this function. It would have been an interesting experiment to check the difference in the results if the rotation function did work.

There are many directions in which my finished project could go. This could be done by me or someone else if any further work was to be done on it. The first main area I would investigate would be to extend the size of the training data by both the number of people it includes and the total number of expressions that are included within it. It would also be interesting to see if a fixed eigenvector blendshape model would produce better animations than current methods used in the system. A further avenue of research within this topic area would be to find a way so that the expression vector solvers are less dependent on the tracked 2D landmark, meaning it could still work even when errors occur or perhaps when the 2D landmarks can't be found.

If I was to redo the project again, knowing the information and techniques I know now, it is likely that I would go about the project in a different manner, focusing more on written aspect and making sure I spend more time on the coding of the project in first semester and the winter break. As I now have a greater understanding of how to code a larger system than I had before, and have the writing skills to write a full dissertation to accompany any large software. I will be able to use this information for any project I work on in the future.

Bibliography

Beeler, T., Hahn, F., Bradley, D., Bickel, B., Beardsley, P., Gotsman, C., Sumner, R.W. and Gross, M., High-quality passive facial performance capture using anchor frames. In: ACM Transactions on Graphics (TOG), 2011. ACM, p. 75.

Bouaziz, S., Wang, Y. and Pauly, M., 2013. Online modeling for realtime facial animation. *ACM Transactions on Graphics (TOG)*, 32(4), p. 40.

Bradley, D., Heidrich, W., Popa, T. and Sheffer, A., High resolution passive facial performance capture. In: ACM Transactions on Graphics (TOG), 2010. ACM, p. 41.

Cao, C., Bradley, D., Zhou, K. and Beeler, T., 2015. Real-time high-fidelity facial performance capture. *ACM Transactions on Graphics (TOG)*, 34(4), p. 46.

Cao, C., Hou, Q. and Zhou, K., 2014. Displaced dynamic expression regression for real-time facial tracking and animation. *ACM Transactions on Graphics (TOG)*, 33(4), p. 43.

Cao, C., Weng, Y., Lin, S. and Zhou, K., 2013. 3D shape regression for real-time facial animation. *ACM Transactions on Graphics (TOG)*, 32(4), p. 41.

Kazemi, V. and Sullivan, J., One millisecond face alignment with an ensemble of regression trees. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2014. pp. 1867-1874.

King, D.E., 2009. Dlib-ml: A machine learning toolkit. *Journal of Machine Learning Research*, 10(Jul), pp. 1755-1758.

Lasseter, J., Principles of traditional animation applied to 3D computer animation. In, 1987. ACM, pp. 35-44.

Li, H., Weise, T. and Pauly, M., Example-based facial rigging. In: ACM Transactions on Graphics (TOG), 2010. ACM, p. 32.

Liu, Y., Xu, F., Chai, J., Tong, X., Wang, L. and Huo, Q., 2015. Video-audio driven real-time facial animation. *ACM Transactions on Graphics (TOG)*, 34(6), p. 182.

Pearson, K., 1901. LIII. On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11), pp. 559-572.

Powell, M.J., 2009. The BOBYQA algorithm for bound constrained optimization without derivatives. *Cambridge NA Report NA2009/06, University of Cambridge, Cambridge*.

Ravikumar, S., Davidson, C., Kit, D., Campbell, N., Benedetti, L. and Cosker, D., Reading Between the Dots: Combining 3D Markers and FACS Classification for High-Quality Blendshape Facial Animation.

Smith, L.I., 2002. A tutorial on principal components analysis. *Cornell University, USA*, 51(52), p. 65.

Weise, T., Bouaziz, S., Li, H. and Pauly, M., Realtime performance-based facial animation. In: ACM Transactions on Graphics (TOG), 2011. ACM, p. 77.

Weng, Y., Cao, C., Hou, Q. and Zhou, K., 2014. Real-time facial animation on mobile devices. *Graphical Models*, 76(3), pp. 172-179.

Xiong, X. and De la Torre, F., Supervised descent method and its applications to face alignment. In: Proceedings of the IEEE conference on computer vision and pattern recognition, 2013. pp. 532-539.

Appendix A

Design Diagrams

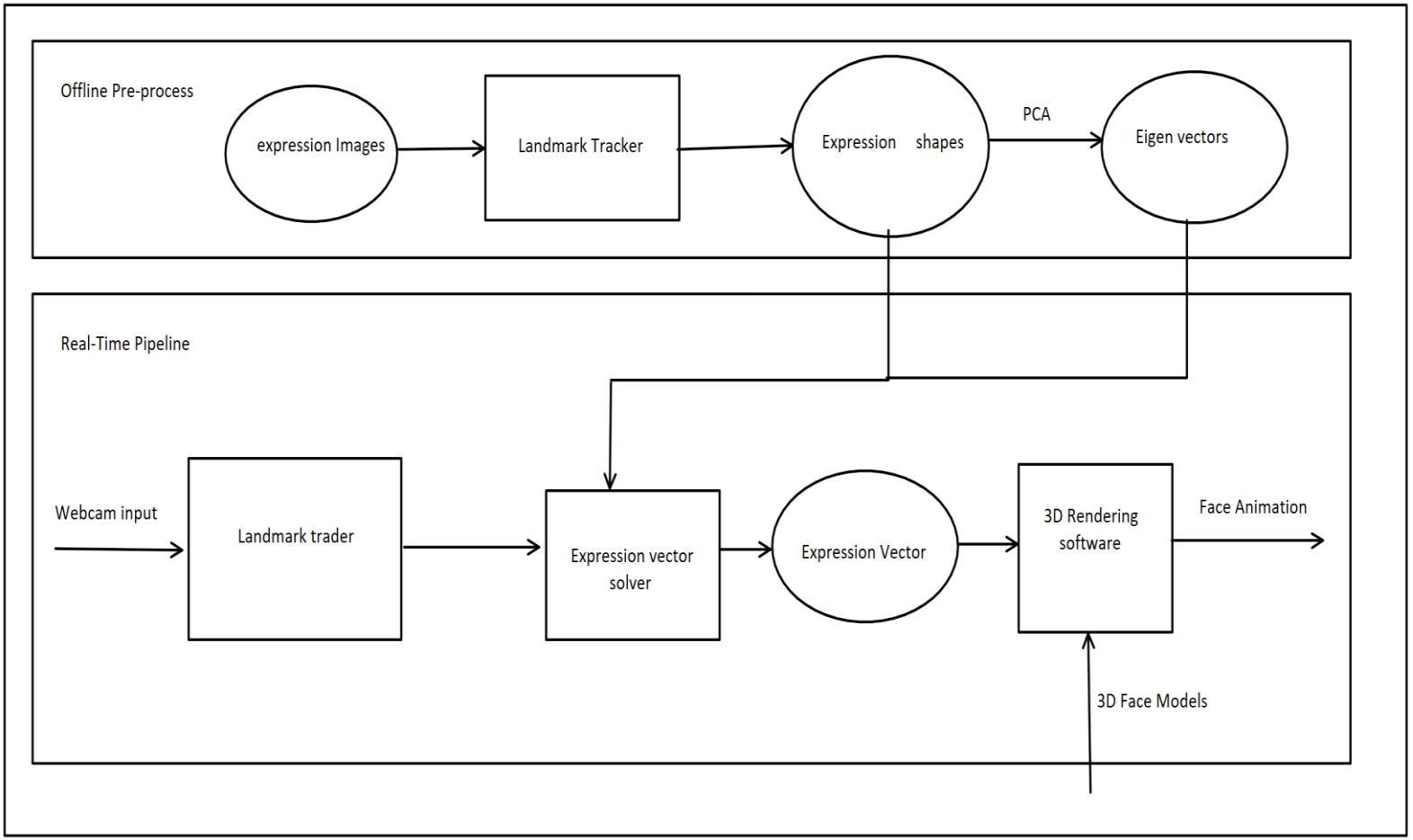


Figure 4 Design diagram for the offline and real-time processes

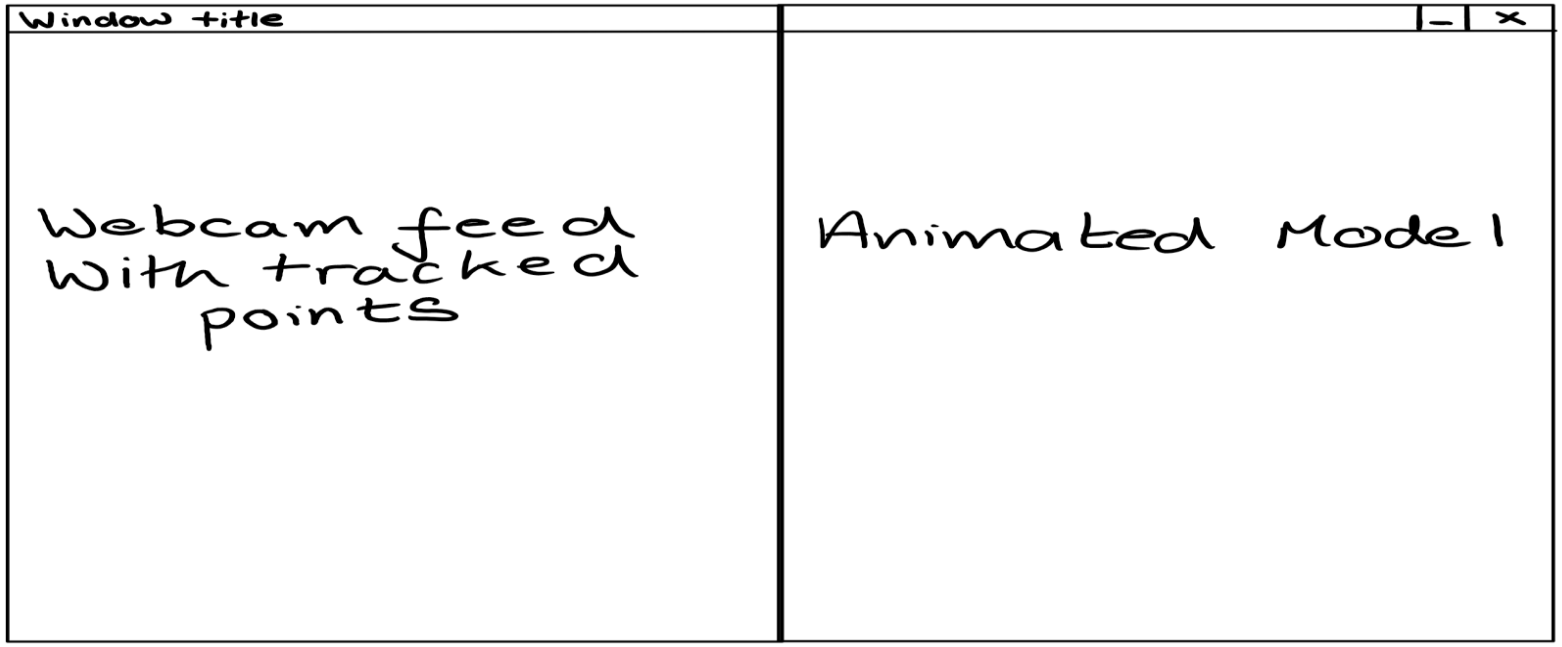
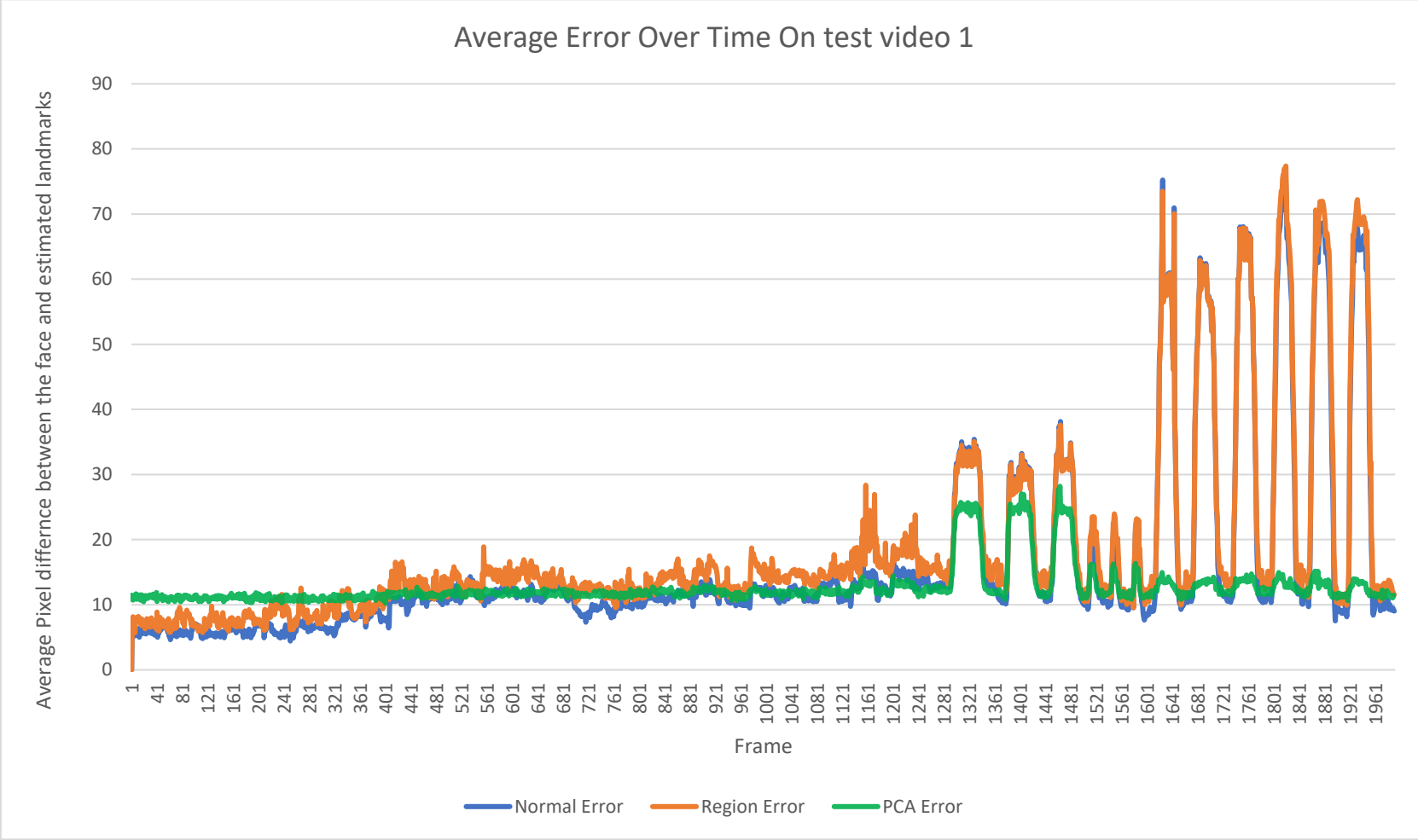
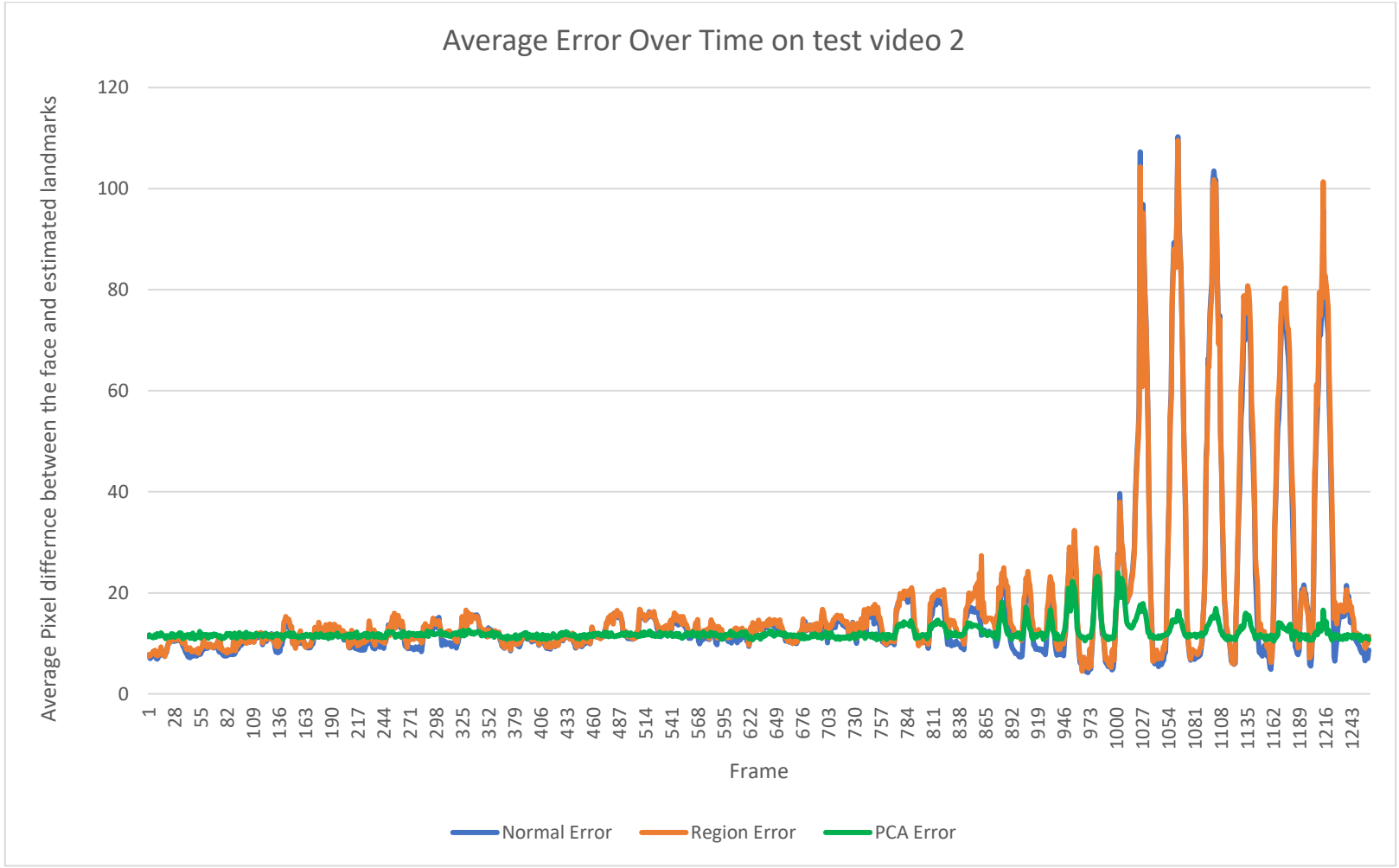


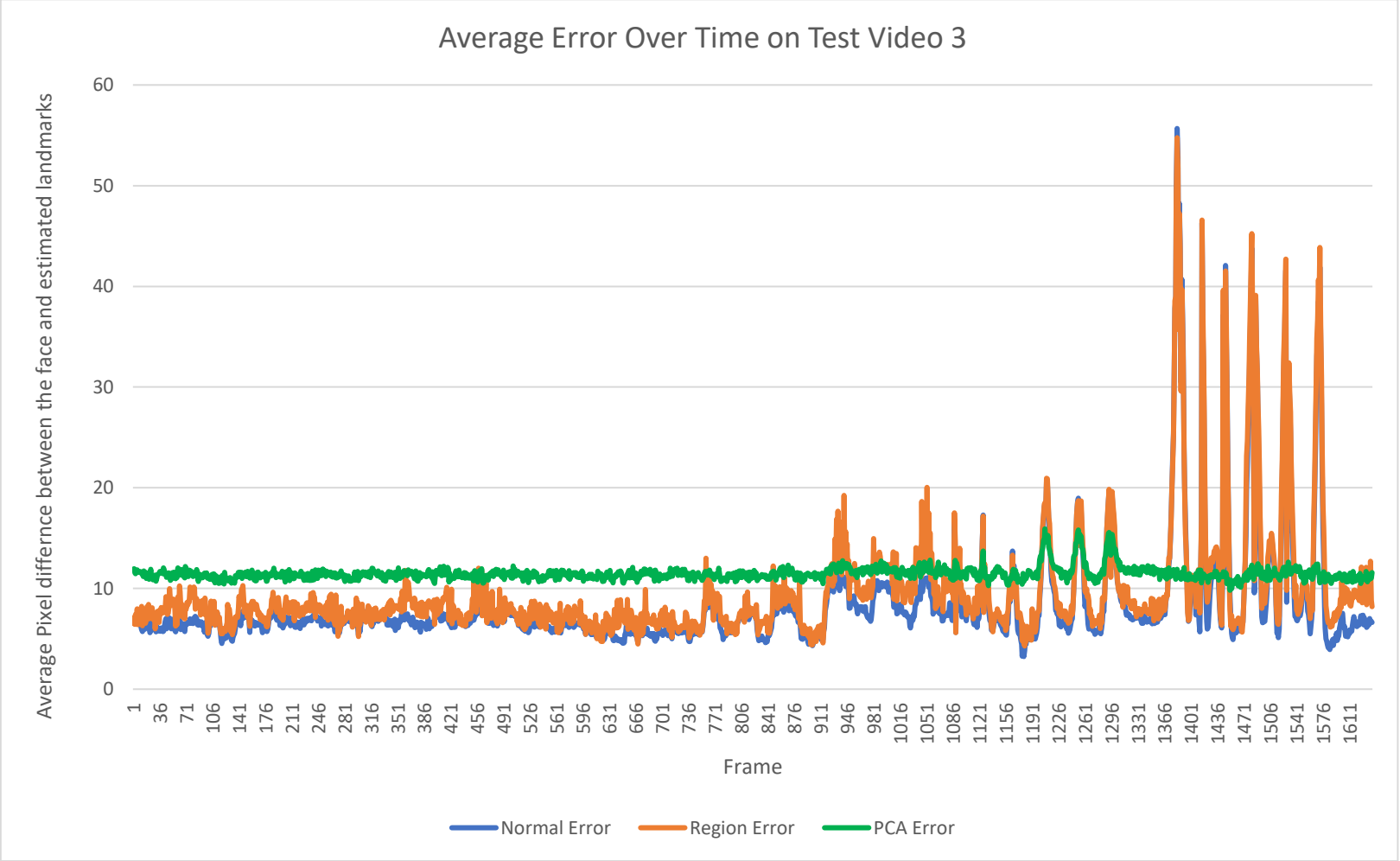
Figure 5 Design diagram showing the simple design of the window to display the webcam and the animated model

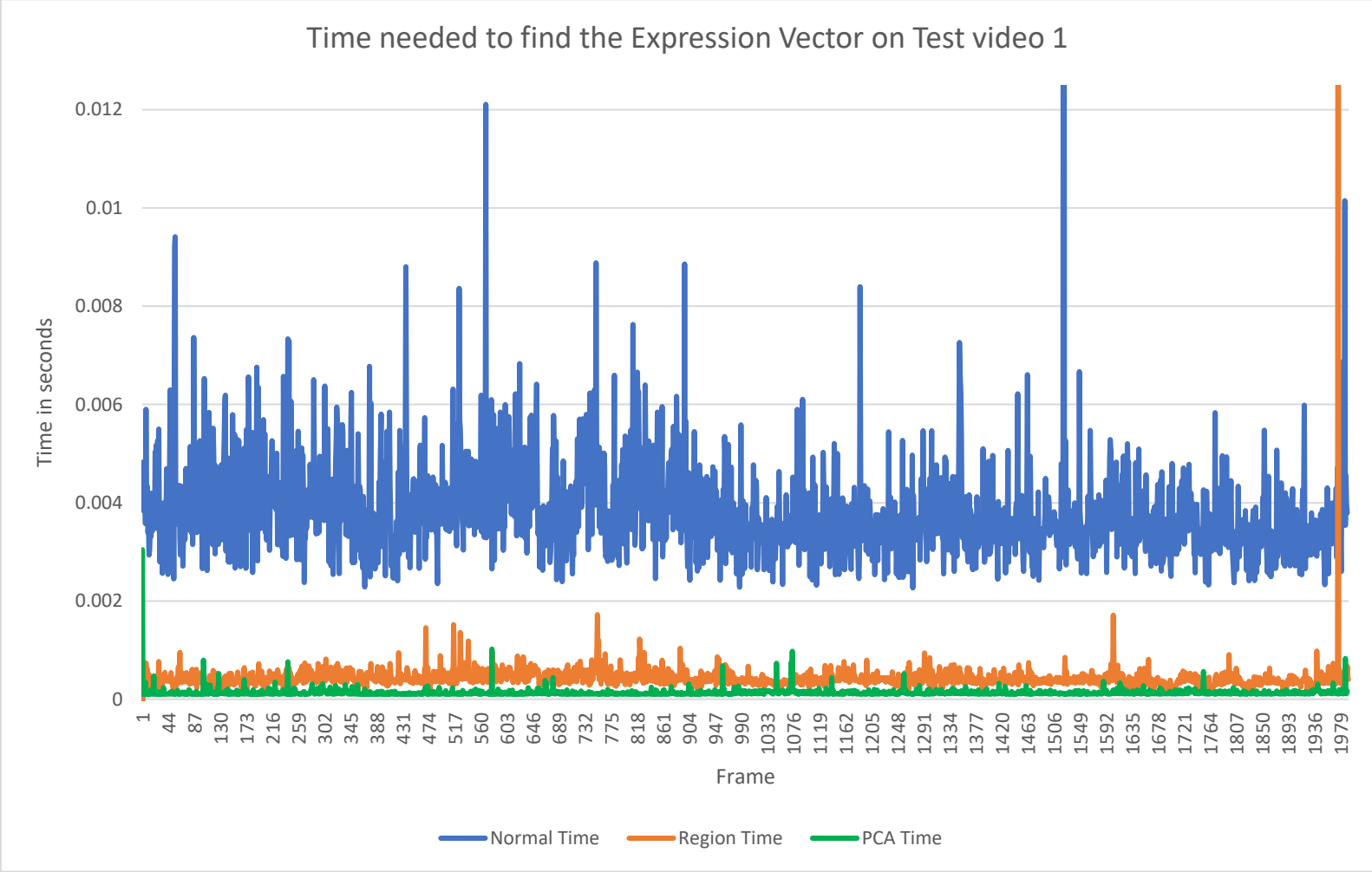
Appendix B

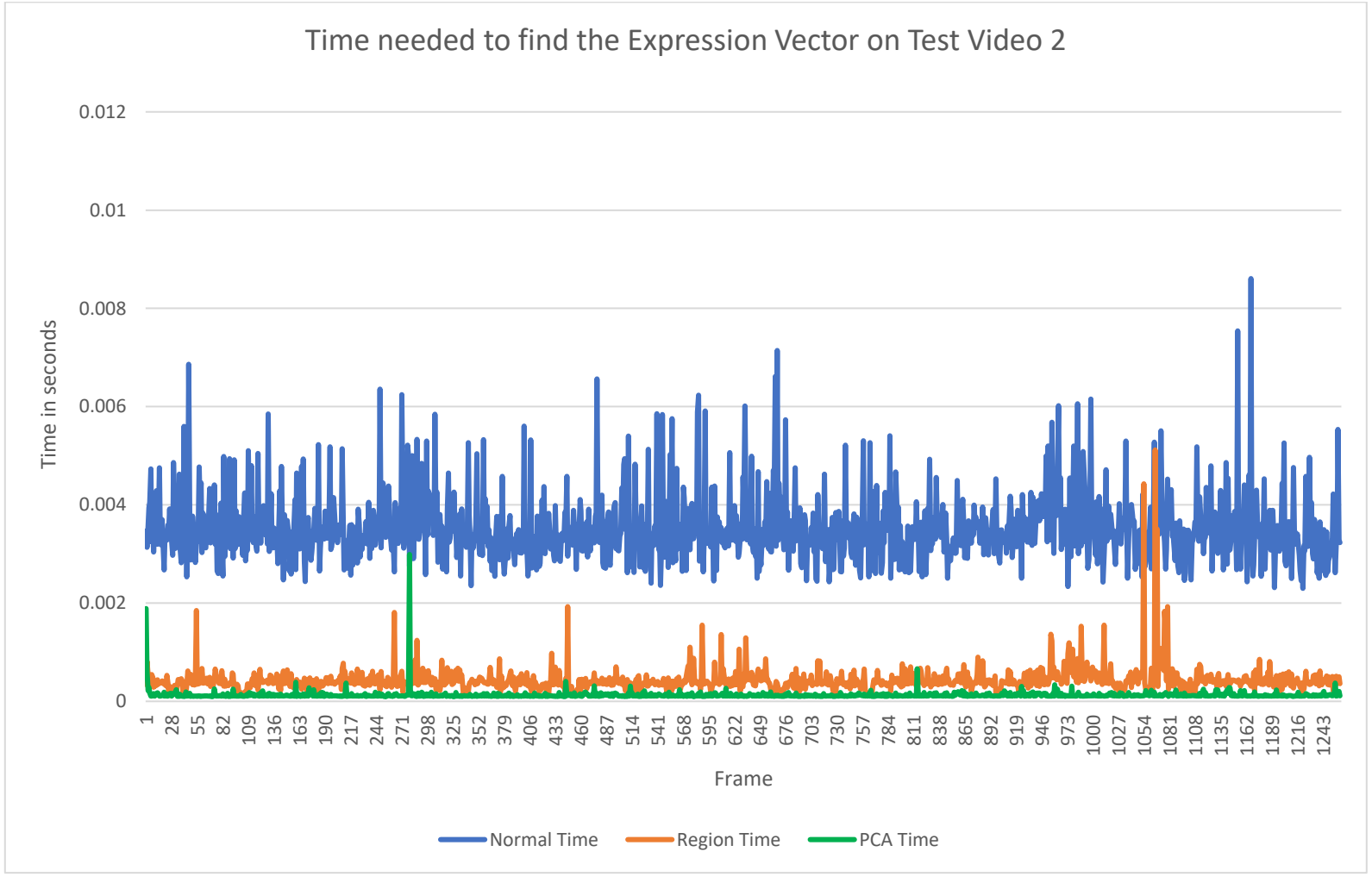
Raw results output

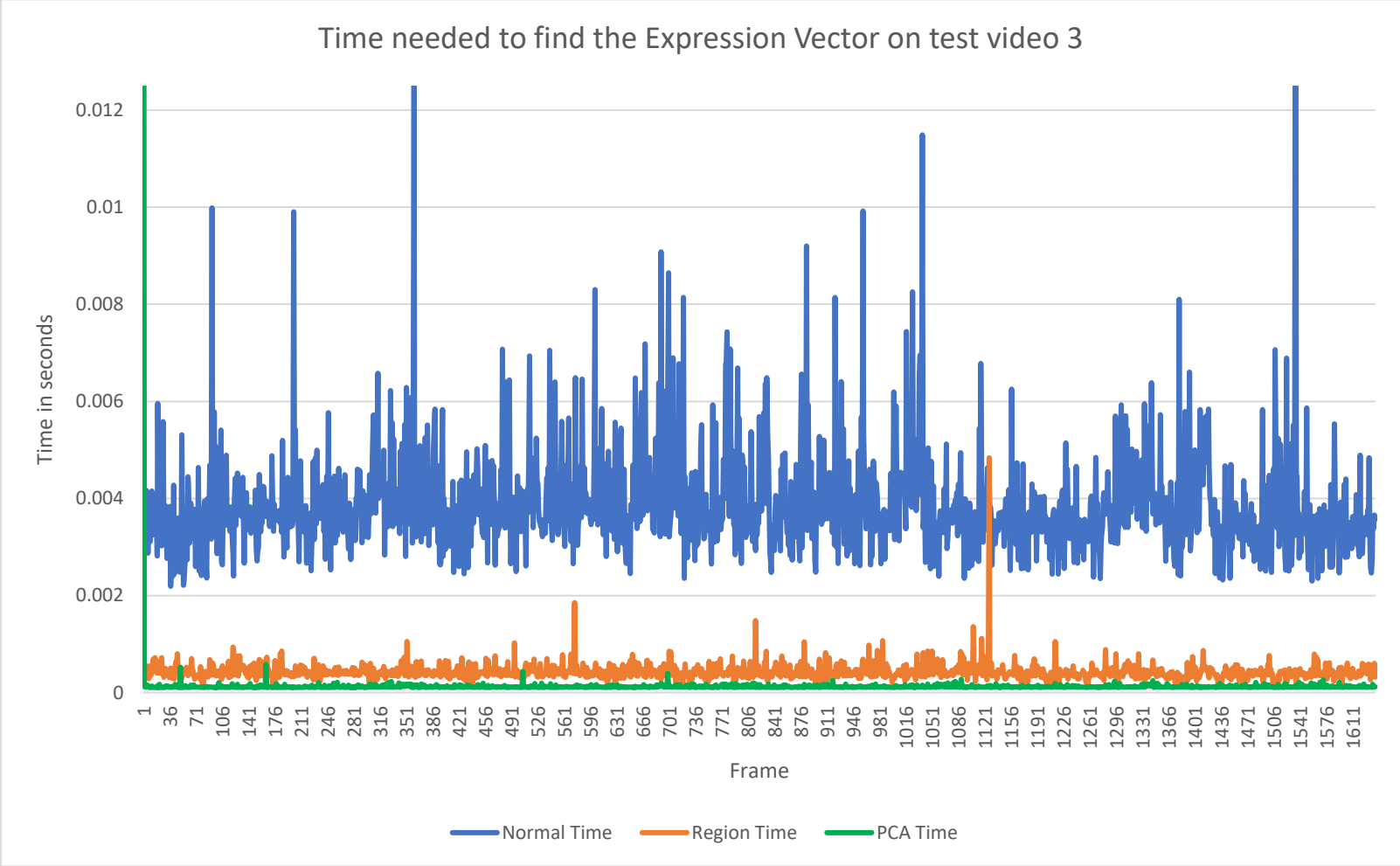

Average Error Over Time On test video 1
Average Pixel differnce between the face and estimated landmarks
90
80
70
60
50
40
30
20
10
0
1
41
81
121
161
201
241
281
321
361
401
441
481
521
561
601
641
681
721
761
801
841
881
921
961
1001
1041
1081
1121
1161
1201
1241
1281
1321
1361
1401
1441
1481
1521
1561
1601
1641
1681
1721
1761
1801
1841
1881
1921
1961
Frame
Normal Error
Region Error
PCA Error


Average Error Over Time on test video 2
Average Pixel differnce between the face and estimated landmarks
120
100
80
60
40
20
0
1
28
55
82
109
136
163
190
217
244
271
298
325
352
379
406
433
460
487
514
541
568
595
622
649
676
703
730
757
784
811
838
865
892
919
946
973
1000
1027
1054
1081
1108
1135
1162
1189
1216
1243
Frame
Normal Error
Region Error
PCA Error


Average Error Over Time on Test Video 3
Average Pixel differnce between the face and estimated landmarks
60
50
40
30
20
10
0
1
36
71
106
141
176
211
246
281
316
351
386
421
456
491
526
561
596
631
666
701
736
771
806
841
876
911
946
981
1016
1051
1086
1121
1156
1191
1226
1261
1296
1331
1366
1401
1436
1471
1506
1541
1576
1611
Frame
Normal Error
Region Error
PCA Error


Time needed to find the Expression Vector on Test video 1
Time in seconds
0.012
0.01
0.008
0.006
0.004
0.002
0
1
44
87
130
173
216
259
302
345
388
431
474
517
560
603
646
689
732
775
818
861
904
947
990
1033
1076
1119
1162
1205
1248
1291
1334
1377
1420
1463
1506
1549
1592
1635
1678
1721
1764
1807
1850
1893
1936
1979
Frame
Normal Time
Region Time
PCA Time


Time needed to find the Expression Vector on Test Video 2
Time in seconds
0.012
0.01
0.008
0.006
0.004
0.002
0
1
28
55
82
109
136
163
190
217
244
271
298
325
352
379
406
433
460
487
514
541
568
595
622
649
676
703
730
757
784
811
838
865
892
919
946
973
1000
1027
1054
1081
1108
1135
1162
1189
1216
1243
Frame
Normal Time
Region Time
PCA Time


Time needed to find the Expression Vector on test video 3
Time in seconds
0.012
0.01
0.008
0.006
0.004
0.002
0
1
36
71
106
141
176
211
246
281
316
351
386
421
456
491
526
561
596
631
666
701
736
771
806
841
876
911
946
981
1016
1051
1086
1121
1156
1191
1226
1261
1296
1331
1366
1401
1436
1471
1506
1541
1576
1611
Frame
Normal Time
Region Time
PCA Time

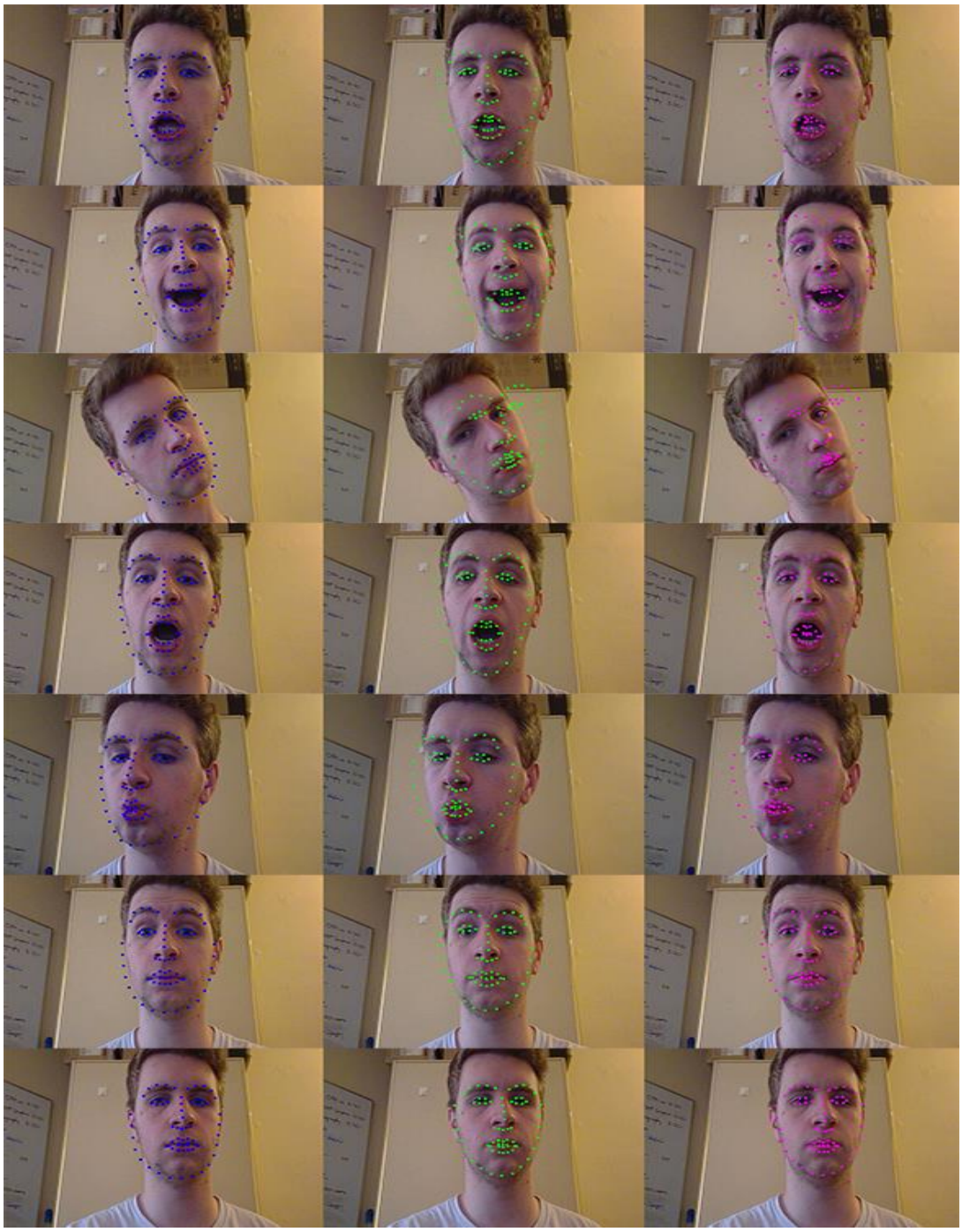


Figure 6. Difference between tracked points and calculated points, left tracked, middle All-in One method, right region method

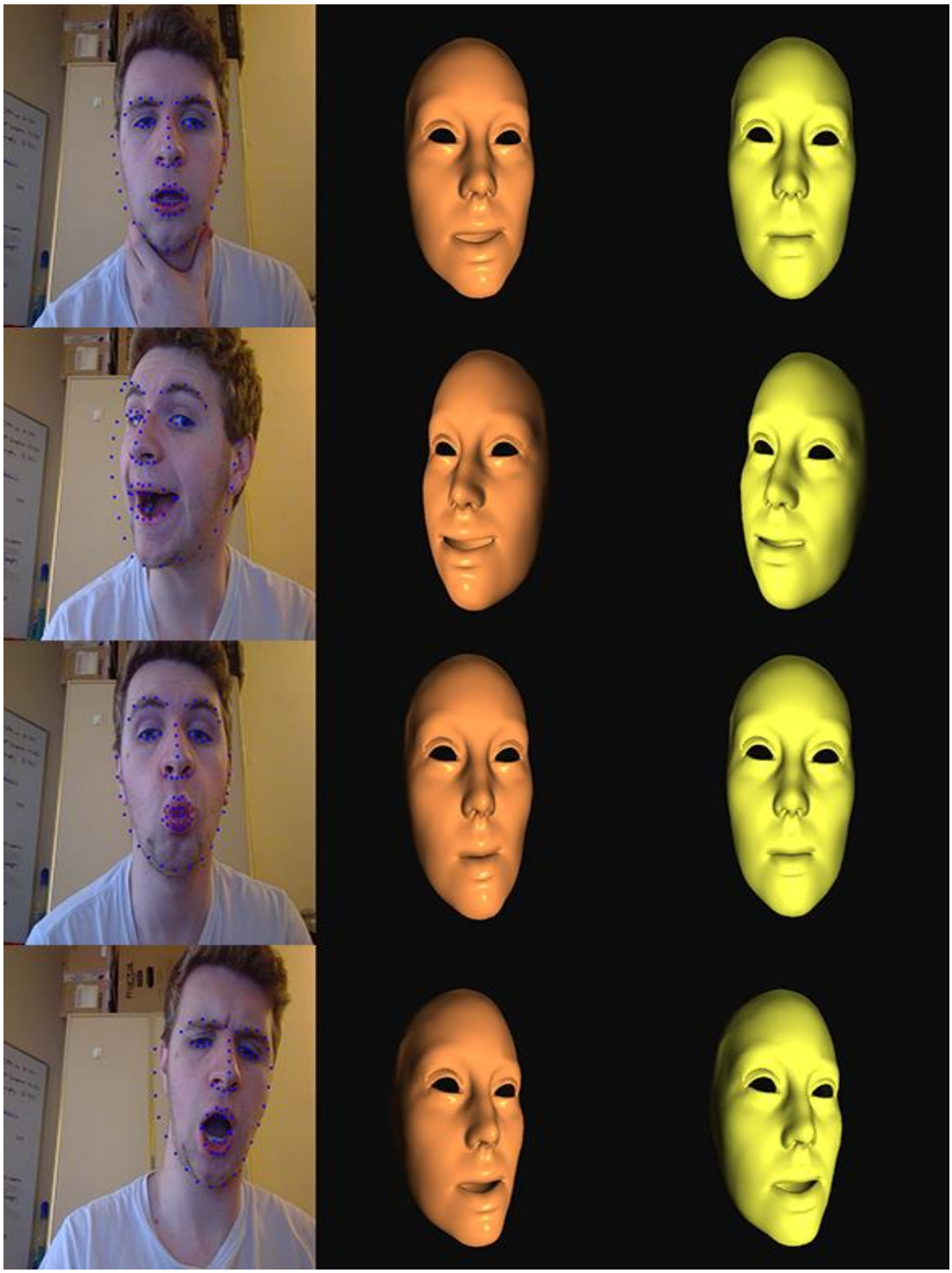


Figure 7. The tracked landmarks with the final animation, left model using the all-in-one expression vector and the right using the one found in the region solve.

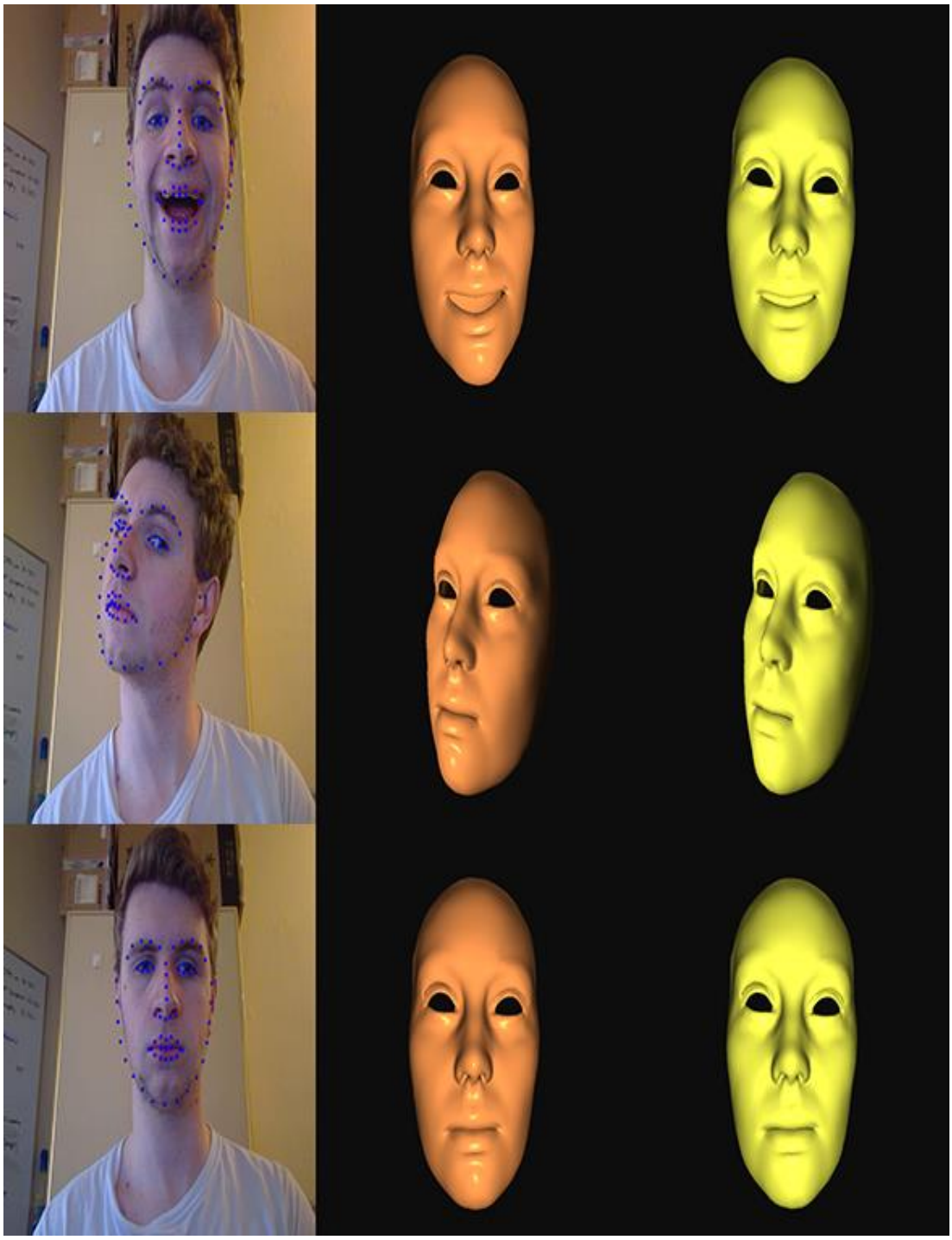


Figure 8. More examples of the final output of the blendshape sum

Appendix C

Code

A.1.1 BlendShape.h

Used To define a blendshape model object.

```
class Blendshape
{
public:
	std::vector<Vertex> baseModel;
	std::vector<std::vector<Vertex>> blendshapes;
	/*Functions */
	//Define the Blenshape for a given base shape and a path for the
	first blenshapes
	Blendshape(GLchar* path, std::vector<Vertex> baseModel) {
		this->baseModel = baseModel;
		this->loadModel(path);
	}
	//Define Blendshape with a base shapes
	Blendshape(std::vector<Vertex> baseModel) {
		this->baseModel = baseModel;
	}
	//Add a new model to the blendshape model
	void addShape(GLchar* path) {
		this->loadModel(path);
	}
	void addShape(std::vector<Vertex> blend) {
		this->loadModel(blend);
	}
	//Get the number of Vertices used in the blendshape
	int getNumberVertices(int i) {
		return blendshapes[i].size();
	}
	//Function to retrieve a specific blendshape model
	std::vector<Vertex> getBlendShape(int i) {
		return blendshapes[i];
	}
private:
	/* Model Data */
	std::string directory;
	/* Functions */

//Load Model from file using Assimp Importer
	void loadModel(std::string path){
		Assimp::Importer import;
		const aiScene* scene = import.ReadFile(path,
aiProcess_Triangulate | aiProcess_FlipUVs | aiProcess_GenNormals);

		if (!scene || scene->mFlags == AI_SCENE_FLAGS_INCOMPLETE ||
!scene->mRootNode) {
			std::cout << "ERROR::ASSIMP::" <<
import.GetErrorString() << std::endl;
			return;
		}
		this->directory = path.substr(0, path.find_last_of('/'));

		this->processNode(scene->mRootNode, scene);

	}
	void processNode(aiNode* node, const aiScene* scene) {
		// Process all the node's meshes (if any)
		for (GLuint i = 0; i < node->mNumMeshes; i++) {
			aiMesh* mesh = scene->mMeshes[node->mMeshes[i]];
			this->blendshapes.push_back(this->processMesh(mesh,
scene));
		}

		for (GLuint i = 0; i < node->mNumChildren; i++) {
			this->processNode(node->mChildren[i], scene);
		}
	}
	//Define a Model from another
	void loadModel(std::vector<Vertex> blend) {

		//Remove the base model from the model given
		for (unsigned int i = 0; i < blend.size(); i++) {
			glm::vec3 vector;
			vector.x = blend[i].Position.x -
baseModel[i].Position.x;
			vector.x = blend[i].Position.y -
baseModel[i].Position.y;
			vector.x = blend[i].Position.z -
baseModel[i].Position.z;
			blend[i].Position = vector;
		}

		blendshapes.push_back(blend);
	}
	std::vector<Vertex> processMesh(aiMesh* mesh, const aiScene* scene)
```

```
	{
		assert(baseModel.size() == mesh->mNumVertices);
		std::vector<Vertex> vertices;

		for (GLuint i = 0; i < mesh->mNumVertices; i++) {
			Vertex vertex;
			// Process vertex positions
			// Define the vertices as the change from the mean
			glm::vec3 vector;
			vector.x = mesh->mVertices[i].x -
baseModel[i].Position.x;
			vector.y = mesh->mVertices[i].y -
baseModel[i].Position.y;
			vector.z = mesh->mVertices[i].z -
baseModel[i].Position.z;
			vertex.Position = vector;
			//Normals
			vector.x = mesh->mNormals[i].x;
			vector.y = mesh->mNormals[i].y;
			vector.z = mesh->mNormals[i].z;
			vertex.Normal = vector;
			//Texture
			if (mesh->mTextureCoords[0]) {
				glm::vec2 vec;
				vec.x = mesh->mTextureCoords[0][i].x;
				vec.y = mesh->mTextureCoords[0][i].y;
				vertex.TexCoords = vec;
			}
			else
				vertex.TexCoords = glm::vec2(0.0f, 0.0f);

			vertices.push_back(vertex);

		}

		return vertices;
	}

};
```

A.1.2 SourceMain.cpp

The main file which does all the work of the real-time software, the loading of the pre-processed data.

```
/*Real-Time Facial Animation for Untrained Users, Final Year Project By Lander King
*/
//Imports skipped in Apdendix
#define	FACE_DOWNSAMPLE_RATIO 2
using namespace dlib;
typedef matrix<double, 0, 1> column_vector;
GLuint screenWidth = 1800, screenHeight = 900;
const float  PI_F = 3.14159265358979f;
//Storage for current face shape, Example Faces, and Estimated Face Shapes
full_object_detection currentFaceShape;
std::vector<point> shape2;
std::vector<dlib::point> regionSolve;
std::vector<full_object_detection> exampleShapes;
//Storage for Transformation matrices
double Transform[4] = {1,1,0,0};
matrix<double> Rotation(2, 2);
//Storage for PCA data
cv::PCA pca;
cv::Mat PCAEigenVectors, PCAMean;
std::vector<column_vector> EigModels;
#ifndef BIGVISION_RENDER_FACE_H_
#define BIGVISION_RENDER_FACE_H_
```

```
void printColumnVector(const column_vector& m) {
	std::cout << "Vector = ";
	for (int i = 0; i < m.size(); i++) {
		std::cout << m(i);
		std::cout << ", ";
	}
	std::cout << std::endl;
}
point calcWeightedShape(const column_vector& m, int i) {
	assert(m.size() == (exampleShapes.size()));
	point currentPoint;
	currentPoint.x() = (Transform[0] * exampleShapes[0].part(i).x());
	currentPoint.y() = (Transform[1] * exampleShapes[0].part(i).y());
	for (unsigned int j=0; j < exampleShapes.size() ; j++) {
		currentPoint.x() += m(j) * (Transform[0] * (exampleShapes[j].part(i).x() - exampleShapes[0].part(i).x()));
		currentPoint.y() += m(j) * (Transform[1] * (exampleShapes[j].part(i).y() - exampleShapes[0].part(i).y()));
	}
	//Apply Rotation and translation
	double temp = (currentPoint.x() * Rotation(0, 0)) + (currentPoint.y() * Rotation(0, 1)) + Transform[2];
	currentPoint.y() = (currentPoint.x() * Rotation(1, 0)) + (currentPoint.y() * Rotation(1, 1)) + Transform[3];
	currentPoint.x() = temp;
	return currentPoint;
}
point calcRotatedShape(const column_vector& m, int i) {
	assert(m.size() == 4);
	point currentPoint;
	currentPoint.x() = (Transform[0] * exampleShapes[0].part(i).x());
	currentPoint.y() = (Transform[1] * exampleShapes[0].part(i).y());
	int limit = exampleShapes.size();
	int j = limit - m.size();
	//Check if  the current point is one associated with the jawline or nose region
	for (int k=0; k < m.size(); j++,k++) {
		currentPoint.x() += m(k) * (Transform[0] * (exampleShapes[j].part(i).x() - exampleShapes[0].part(i).x()));
		currentPoint.y() += m(k) * (Transform[1] * (exampleShapes[j].part(i).y() - exampleShapes[0].part(i).y()));
	}
	//Apply Rotation and translation
	double temp = (currentPoint.x() * Rotation(0, 0)) + (currentPoint.y() * Rotation(0, 1)) + Transform[2];
	currentPoint.y() = (currentPoint.x() * Rotation(1, 0)) + (currentPoint.y() * Rotation(1, 1)) + Transform[3];
	currentPoint.x() = temp;
	return currentPoint;
}
point calcRegionShape(const column_vector& m, int i) {
	assert(m.size() == (exampleShapes.size())-4);
	point currentPoint;
	currentPoint.x() = regionSolve[i].x();
	currentPoint.y() = regionSolve[i].y();
```

```
		//Check if  the current point is one associated with the eyebrow region
		if (i > 17 && i<= 26)
		{
			for (int j = 0; j < 6; j++) {
				currentPoint.x() += m(j) * (Transform[0] * (exampleShapes[j].part(i).x() - exampleShapes[0].part(i).x()));
				currentPoint.y() += m(j) * (Transform[1] * (exampleShapes[j].part(i).y() - exampleShapes[0].part(i).y()));
			}
		}
		//Check if the current Point is associated with the mouth region
		else if (i<=16 ||(i >= 48 && i <= 68 )) {
			for (int j = 6; j < m.size(); j++) {
				currentPoint.x() += m(j) * (Transform[0] * (exampleShapes[j].part(i).x() - exampleShapes[0].part(i).x()));
				currentPoint.y() += m(j) * (Transform[1] * (exampleShapes[j].part(i).y() - exampleShapes[0].part(i).y()));
			}
		}
		//Apply Rotation and translation
		double temp = (currentPoint.x() * Rotation(0, 0)) + (currentPoint.y() * Rotation(0, 1));
		currentPoint.y() = (currentPoint.x() * Rotation(1, 0)) + (currentPoint.y() * Rotation(1, 1));
		currentPoint.x() = temp;
		return currentPoint;
}
```

```
double calcDiffL2NormSquared(point l, point m) {
		double xdif = l.x() - m.x();
		double ydif = l.y() - m.y();
		ydif = ydif*ydif;
		xdif = xdif*xdif;
		double difLength = sqrt(ydif + xdif);
		return difLength;
}
double calcL1Norm(const column_vector &m) {
		double l1 = 0;
		for (int i = 0; i < m.size(); i++) {
			l1 += abs(m(i));
		}
		return l1;
}

double minFunction(const column_vector& m) {
		double result = 0.0;
		//Total L2 Norm of Difference between Estimated Points and Observed Points
		for (unsigned int i = 0; i < currentFaceShape.num_parts(); i++) {
			point weightedPoint = calcWeightedShape(m, i);
			result += calcDiffL2NormSquared(currentFaceShape.part(i), weightedPoint);
		}
		//Add L1 Norm to normallaize result(make sparse)
		result += calcL1Norm(m);
		return result;}
```

```
double rotationMinFunction(const column_vector& m) {
	double result = 0.0;
	//Total L2 Norm of Difference between Estimated Points and Observed Points
	for (unsigned int i = 0; i < currentFaceShape.num_parts(); i++) {
		point weightedPoint = calcRotatedShape(m, i);
		result += calcDiffL2NormSquared(currentFaceShape.part(i), weightedPoint);
	}
	//Add L1 Norm to normallaize result(make sparse)
	result += calcL1Norm(m);
	return result;
}
double regionMinFunction(const column_vector& m) {
	double result = 0.0;
	//Total L2 Norm of Difference between Estimated Points and Observed Points
	for (unsigned int i = 0; i < currentFaceShape.num_parts(); i++) {
		point weightedPoint = calcRegionShape(m, i);
		result += calcDiffL2NormSquared(currentFaceShape.part(i), weightedPoint);
	}
	//Add L1 Norm to normallaize result(make sparse)
	result += calcL1Norm(m);
	return result;
}
double PCAsolvefunction(const column_vector& m) {
	double result = 0.0;
	int eigenId = EigModels.size();
	double xdif, ydif;
//Solve for the Total of the L2 Norm of Difference between Estimated Points and Observed Points
	//Solve for mean
	if (eigenId == 0) {
		for (int i = 0; i < PCAEigenVectors.cols / 2; i++) {
			point weightedPoint = calcWeightedShape(m, i);
			xdif = PCAMean.at<double>(2 * i) - weightedPoint.x();
			ydif = PCAMean.at<double>((2 * i) + 1) - weightedPoint.y();
			ydif = ydif*ydif;
			xdif = xdif*xdif;
			result += sqrt(ydif + xdif);
		}
	}
	//Solve for Eigen Vectors
	else {
		for (int i = 0; i < PCAEigenVectors.cols / 2; i++) {
			point weightedPoint = calcWeightedShape(m, i);
			xdif = PCAEigenVectors.at<double>(eigenId, 2 * i) + PCAMean.at<double>(2 * i) - weightedPoint.x();
			ydif = PCAEigenVectors.at<double>(eigenId, (2 * i) + 1) + PCAMean.at<double>((2 * i) + 1) - weightedPoint.y();
			ydif = ydif*ydif;
			xdif = xdif*xdif;
			result += sqrt(ydif + xdif);
		}
	}
```

```
		//Add L1 Norm to normallaize result(make sparse)
		result += calcL1Norm(m);
		return result;
}
full_object_detection moveToTop(full_object_detection shape, rectangle r) {
		//Re-align the detected face to the Top Corner
		for (unsigned long i = 0; i < shape.num_parts(); ++i) {
			shape.part(i).x() = shape.part(i).x() - r.left();
			shape.part(i).y() = shape.part(i).y() - r.top();
		}
		return shape;
}
std::vector<point> getFinalShape(const column_vector& m) {
		std::vector<point> FinalShape;
		point Point;
		for (unsigned long i = 0; i < exampleShapes[0].num_parts(); i++) {
			Point.x() =(Transform[0] * exampleShapes[0].part(i).x());
			Point.y() = (Transform[1] * exampleShapes[0].part(i).y());
			for (unsigned int j = 0; j < exampleShapes.size(); j++) {
				Point.x() = Point.x() + (m(j) * (Transform[0] * (exampleShapes[j].part(i).x()-exampleShapes[0].part(i).x())));
				Point.y() = Point.y() + (m(j) * (Transform[1] * (exampleShapes[j].part(i).y()-exampleShapes[0].part(i).y())));
			}
			double temp = (Point.x() * Rotation(0, 0)) + (Point.y() * Rotation(0, 1));
			Point.y() = (Point.x() * Rotation(1, 0)) + (Point.y() * Rotation(1, 1)) + Transform[3];
			Point.x() = temp + Transform[2];
			FinalShape.push_back(Point);
		}
		return FinalShape;
}
std::vector<point> getFinalPcaShape(const cv::Mat &m) {
		std::vector<point> FinalShape;
		point Point;
		for (unsigned long i = 0; i < exampleShapes[0].num_parts(); i++) {
			Point.x() = (Transform[0] * PCAMean.at<double>(2 * i));
			Point.y() = (Transform[1] * PCAMean.at<double>((2 * i) + 1));
			for (int j = 0; j < PCAEigenVectors.rows; j++) {
				Point.x() = Point.x() + (m.at<double>(j) * PCAEigenVectors.at<double>(j,2*i));
				Point.y() = Point.y() + (m.at<double>(j) * PCAEigenVectors.at<double>(j, (2 * i)+1));
			}
			double temp = (Point.x() * Rotation(0, 0)) + (Point.y() * Rotation(0, 1));
			Point.y() = (Point.x() * Rotation(1, 0)) + (Point.y() * Rotation(1, 1)) + Transform[3];
			Point.x() = temp + Transform[2];
			FinalShape.push_back(Point);
		}
		return FinalShape;
}
std::vector<full_object_detection> initial_example_face_matrix(shape_predictor &sp, frontal_face_detector &detector) {
		//Load Example Facial Expressions
		const int nImages = 16;
```

```
		array2d<rgb_pixel> images[nImages];
		load_image(images[0], "images/Neutral.png");
		load_image(images[1], "images/BrowLL.png"); //Lowered Left Eyebrow
		load_image(images[2], "images/BrowRL.png"); //Raised Left Eyebrow
		load_image(images[3], "images/BrowLR.png"); //Lowered Right Eyebrow
		load_image(images[4], "images/BrowRR.png"); //Raised Right Eyebrow
		load_image(images[5], "images/Suck.png");
		load_image(images[6], "images/PullOpen.png");
load_image(images[7], "images/PullL.png");
		load_image(images[8], "images/PullR.png");
		load_image(images[9], "images/Depress.png");
		load_image(images[10], "images/Pucker.png");
		load_image(images[11], "images/Stretch.png");
		load_image(images[12], "images/RotationL.png");
		load_image(images[13], "images/RotationR.png");
		load_image(images[14], "images/tiltL.png");
		load_image(images[15], "images/tiltR.png");
		// Make the image larger so we can detect small faces.
		std::vector<full_object_detection> shapes;
		for (unsigned long i = 0; i < nImages; i++) {
			//pyramid_up(images[i]);
			// Now tell the face detector to give us a list of bounding boxes
			// around all the faces in the image.
			std::vector<rectangle> dets = detector(images[i]);
			if (dets.size() == 0) {
				pyramid_up(images[i]);
				dets = detector(images[i]);
			}
			// Now we will go ask the shape_predictor to tell us the pose of
			// each face we detected.
			for (unsigned long j = 0; j < dets.size(); ++j)
			{
				full_object_detection shape = sp(images[i], dets[j]);
				shapes.push_back(shape);
			}
		}

		std::cout << "Shapes Size = " + shapes.size() << std::endl;


		return shapes;


}


void draw_polyline(cv::Mat &img, const dlib::full_object_detection& d, const int start, const int end, bool
isClosed = false)
{
		std::vector <cv::Point> points;
		for (int i = start; i <= end; ++i)
		{
			points.push_back(cv::Point(d.part(i).x(), d.part(i).y()));
```

```
		}
		cv::polylines(img, points, isClosed, cv::Scalar(255, 0, 0), 2, 16);


}
void render_face(cv::Mat &img, const dlib::full_object_detection& d, const std::vector<point> e, const std::vector<point> f)
{
		//Draw the landmarks ontop of the image
		for (unsigned int i = 0; i < d.num_parts(); i++) {
			cv::circle(img, cv::Point(d.part(i).x(), d.part(i).y()), 1, cv::Scalar(255, 0, 0), 2, 16); // Tracked Landmarks
			cv::circle(img, cv::Point(e[i].x(), e[i].y()), 1, cv::Scalar(0, 255, 0), 2, 16); // All-in-One Landmarks
			cv::circle(img, cv::Point(f[i].x(), f[i].y()), 1, cv::Scalar(255, 0, 255), 2, 16); // Region Landmarks
		}
}
#endif // BIGVISION_RENDER_FACE_H_
rectangle findLandmarkRectangle(full_object_detection &d) {
//Finds the Rectangle that bounds the detected landmarks exluding the points representing the Eyebrows
		float minX = d.part(0).x(), minY = d.part(0).y(), maxX = d.part(0).x(), maxY = d.part(0).y();
		for (unsigned long j = 1; j < d.num_parts(); ++j) {
			if (!(j >= 16 && j <= 27)) {
				if (d.part(j).x() <= minX) minX = d.part(j).x();
				if (d.part(j).x() >= maxX) maxX = d.part(j).x();
				if (d.part(j).y() <= minY) minY = d.part(j).y();
				if (d.part(j).y() >= maxY) maxY = d.part(j).y();
			}
		}
		rectangle r(minX, maxY, maxX, minY);
		return r;
}
rectangle findLandmarkRectangle(std::vector<point> &d) {
//Finds the Rectangle that bounds the detected landmarks exluding the points representing the Eyebrows
		float minX = d[0].x(), minY = d[0].y(), maxX = d[0].x(), maxY = d[0].y();
		for (unsigned long j = 1; j < d.size(); ++j) {
			if (!(j >= 16 && j <= 27)) {
				if (d[j].x() <= minX) minX = d[j].x();
				if (d[j].x() >= maxX) maxX = d[j].x();
				if (d[j].y() <= minY) minY = d[j].y();
				if (d[j].y() >= maxY) maxY = d[j].y();
			}
		}
		rectangle r(minX, maxY, maxX, minY);
		return r;
}
void findTransform(dlib::rectangle &currentRec, dlib::rectangle &baseRec) {
		//Finds the scale/ transform and rotation needed to map the estimated face onto the current face
		double currentWidth = (currentRec.right() - currentRec.left());
		double currentHeight = (currentRec.top() - currentRec.bottom());
		double baseWidth = (baseRec.right() - baseRec.left());
		double baseHeight = (baseRec.top() - baseRec.bottom());
		double currentMidx = currentRec.left() + (currentRec.width() / 2);
		double currentMidy = currentRec.bottom() + (currentRec.height() / 2);
```

```
		double baseMidx = (currentWidth / baseWidth) * (baseRec.left() + (baseRec.width() / 2));
		double baseMidy = (currentHeight / baseHeight) * (baseRec.bottom() + (baseRec.height() / 2));
		Transform[0] = currentWidth / baseWidth;  // X Scale
		Transform[1] = currentHeight / baseHeight; // Y Scale
		Transform[2] = currentMidx - baseMidx; // X translation
		Transform[3] = currentMidy - baseMidy; // Y translation
}
void findRotation(full_object_detection &face) {
//Find the Rotation needed to match the Scaled Estimatated Face Shape onto the Viewed Face Shape Use Singular Value Decompostion. This is needed to to get a better estimation of the shape and to be able to add rotation in the xy-plane for the final model
		column_vector estPoint(2), currPoint(2);
		column_vector centerC(2), centerE(2);
		for (unsigned int i = 0; i < face.num_parts(); i++) {
			currPoint = face.part(i).x(), face.part(i).y();
			estPoint = (Transform[0] * exampleShapes[0].part(i).x()), (Transform[1] * exampleShapes[0].part(i).y());
			centerC += currPoint;
			centerE += estPoint;
		}
		centerC = centerC / face.num_parts();
		centerE = centerE / face.num_parts();
		matrix<double> H, U, W, V,rotNew;
		for (unsigned int i = 0; i < face.num_parts(); i++) {
currPoint = face.part(i).x(), face.part(i).y();
estPoint = (Transform[0] * exampleShapes[0].part(i).x()), (Transform[1] * exampleShapes[0].part(i).y());
			H += (estPoint - centerE) * trans(currPoint - centerC);
		}
		svd(H, U, W, V);
		rotNew = V * trans(U);
		if (!(det(rotNew) < 0) & isfinite(rotNew)) {
			Rotation = rotNew;
		}
		matrix<double> T = (-Rotation * centerE) + centerC;
		Transform[2] = T(0);
		Transform[3] = T(1);
		std::cout << Rotation << std::endl;
}
cv::Mat objectDetectionToVector(full_object_detection &d) {
		//Reformat Face Detection to a matrix so we can run PCA on it
		//Change ((x1,y1),(x2,y2),....) to (x1,y1,x2,y2,....)
		cv::Mat vec = (cv::Mat_<double>(1, 2) << d.part(0).x(), d.part(0).y());
		cv::Mat x;
		for (unsigned int i = 1; i < d.num_parts(); i++) {
			x = (cv::Mat_<double>(1, 2) << d.part(i).x(), d.part(i).y());
			cv::hconcat(vec, x, vec);
		}
		//std::cout << vec.size() << std::endl;
		return vec;
}
cv::Mat pcaFaceSolve(full_object_detection &face) {
		cv::Mat b = objectDetectionToVector(face);
		//Need to Scale Mean and EigenVectors to match current face shape
		cv::Mat scaledNorm = b;
```

```
		for (int i = 0; i < PCAMean.cols; i++) {
			if (i % 2 == 0) {
scaledNorm.at<double>(i) -= (Transform[0] * PCAMean.at<double>(i)) +Transform[2];}
			else {
scaledNorm.at<double>(i) -= (Transform[1] * PCAMean.at<double>(i)) +Transform[3];}
		}
		cv::Mat EigenTranspose;
		cv::transpose(PCAEigenVectors,EigenTranspose);
		//Project the scaled norm into
		cv::Mat p = scaledNorm * EigenTranspose;
		//std::cout << p.size() << std::endl;
		//std::cout << p << std::endl;
		return p;
}
void PCAsetup() {
		cv::Mat pcaMat = objectDetectionToVector(exampleShapes[0]), colVec;
		for (unsigned int i = 1; i < exampleShapes.size(); i++) {
			colVec = objectDetectionToVector(exampleShapes[i]);
			vconcat(pcaMat, colVec, pcaMat);
		}
		std::cout << pcaMat.size() << std::endl;
		pca = pca(pcaMat, cv::Mat(), CV_PCA_DATA_AS_ROW);
		cv::Mat eigenValues = pca.eigenvalues.clone();
		cv::Mat eigenVectors = pca.eigenvectors.clone();
		PCAMean = pca.mean.clone();
		double TotalEnergy = cv::sum(eigenValues)[0];
		double currentEnergy = 0;
		//std::cout << matrixSum(eigenValues) << std::endl;
		//std::cout << eigenVectors.size() << std::endl;
		//Find number of EigenVectors to keep to have 99.9999% Energy
		int EigenVecsToKeep = 0;
		while (currentEnergy / TotalEnergy <= 0.9999) {
			currentEnergy += abs(eigenValues.at<double>(EigenVecsToKeep));
			EigenVecsToKeep++;
		}
		//Store the desired EigenVectors
PCAEigenVectors = cv::Mat(EigenVecsToKeep, eigenVectors.cols, CV_64FC1, eigenVectors.data);
		eigenValues = cv::Mat(EigenVecsToKeep, eigenValues.cols, CV_64FC1, eigenValues.data);
		//Solve for new eigenVector Models
		for (int i = 0; i <= PCAEigenVectors.rows; i++) {
			column_vector sol(16);
			sol = 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0;
			//find optimized solution for Each EigenVector
			find_min_bobyqa(PCAsolvefunction, sol, 34, uniform_matrix<double>(16, 1, 0.0),
		uniform_matrix<double>(16, 1, 1.0), 0.075, 0.01, 30000);
			sol(0) = 1.0;
			//Repeat Optimization with reseted first element and smaller stoping criteria
			find_min_bobyqa(PCAsolvefunction, sol, 34, uniform_matrix<double>(16, 1, 0.0),
uniform_matrix<double>(16, 1, 1.0), 0.05, 0.005, 30000);
			EigModels.push_back(sol);
		}
```

```
		//Save to file
		cv::FileStorage fs("EigenVectors.xml", cv::FileStorage::WRITE);
		//fs << "pca" << pca;
		fs << "eigenMatrix" << PCAEigenVectors;
		fs << "PCAmean" << PCAMean;
		//fs << "PCAeigVals" << eigenValues;
		fs << "PCAMODEL" << EigModels;
		fs.release();
		serialize("PCAEigModels.dat") << EigModels;
}
void roundColumnVector(column_vector &m) {
		for (int i = 0; i < m.size(); i++) {
			m(i) = floor((m(i) *1000.0) + 0.5) / 1000.0;
		}
}
std::vector<Vertex> blendshapeSolve(Blendshape &blend,const column_vector &w) {
		std::vector<Vertex> blendModel;
		for (unsigned int i = 0; i < blend.baseModel.size(); i++) {
			Vertex newV = {blend.baseModel[i].Position , blend.baseModel[i].Normal,
blend.baseModel[i].TexCoords };
			for (int j = 1; j < w.size(); j++) {
				newV.Position += (blend.blendshapes[j-1][i].Position * (float)w(j));
			}
			blendModel.push_back(newV);
		}
		return blendModel;}
std::vector<Vertex> blendshapeSolvePca(Blendshape &blend, const column_vector &w) {
		std::vector<Vertex> blendModel;
		for (unsigned int i = 0; i < blend.baseModel.size(); i++) {
			Vertex newV = { blend.baseModel[i].Position , blend.baseModel[i].Normal,
blend.baseModel[i].TexCoords };
			for (int j = 1; j < w.size(); j++) {
				newV.Position += (blend.blendshapes[j-1][i].Position * (float)w(j));
			}
			blendModel.push_back(newV);
		}
		return blendModel;
}
column_vector matToColumn(const cv::Mat &m) {
		assert(m.rows == 1);
		column_vector vec(m.cols);
		for (int i = 0; i < m.cols; i++) {
			vec(i) = m.at<double>(i);
		}
		return vec;}
column_vector concatColVec(const column_vector &a, const column_vector &b) {
		column_vector concatVec(a.size() + b.size());
		for (int i = 0; i < concatVec.size(); i++) {
		if (i >= a.size()) {concatVec(i) = b(i - a.size());}
		else {concatVec(i) = a(i);}
		}
		return concatVec;}
```

```
double pcaError(std::vector<dlib::point> &f) {
	double result = 0;
	for (unsigned int i = 0; i < currentFaceShape.num_parts(); i++) {
		result += calcDiffL2NormSquared(currentFaceShape.part(i), f[i]);
	}
	return result;
}
int main(){
	try {
		Rotation = 1.0, 0.0, 0.0, 1.0;
		//Setup of OPenCv/Dlib Facial Feature Tracking
		cv::VideoCapture cap(1); //Change to 0 if 1 isn't working
		if(!cap.isOpened())
		{
			std::cerr << "Unable to connect to camera" << std::endl;
			return 1;
		}
		cap.set(CV_CAP_PROP_FRAME_WIDTH, 600);
		cap.set(CV_CAP_PROP_FRAME_HEIGHT, 600);
		frontal_face_detector detector = get_frontal_face_detector();
		shape_predictor pose_model;
		//load landmark detector
		deserialize("shape_predictor_68_face_landmarks.dat") >> pose_model;
		//check if example landmark shapes have already been saved, else load it.
		if (pose_model.num_parts() == 68) {
			if (doesFileExist("exampleShapes68.dat")) {
				std::cout << "read examples from file" << std::endl;
				deserialize("exampleShapes68.dat") >> exampleShapes;
			}
			else {
				std::cout << "loaded images" << std::endl;
				exampleShapes = initial_example_face_matrix(pose_model, detector);
				serialize("exampleShapes68.dat") << exampleShapes;
			}
		}
		//check if PCA has been done before.
		if (doesFileExist("EigenVectors.xml") && doesFileExist("PCAEigModels.dat")) {
			cv::FileStorage fs("EigenVectors.xml", cv::FileStorage::READ);
			fs["eigenMatrix"] >> PCAEigenVectors;
			fs["PCAmean"] >> PCAMean;
			fs.release();

			deserialize("PCAEigModels.dat") >> EigModels;
		}
		else {
			//Run PCA on Example Shapes and find solutions for the Orthogonal basis shapes
			PCAsetup();
		}
		//OpenGL SetUP
		glfwInit();
		glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
```

```
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
glfwWindowHint(GLFW_RESIZABLE, GL_FALSE);
GLFWwindow* window = glfwCreateWindow(screenWidth, screenHeight, "LearnOpenGL", nullptr, nullptr);
if (window == nullptr){
    std::cout << "Failed to create GLFW window" << std::endl;
    glfwTerminate();
    return -1;}
glfwMakeContextCurrent(window);
glewExperimental = GL_TRUE;
if (glewInit() != GLEW_OK){
    std::cout << "Failed to initialize GLEW" << std::endl;
    return -1;
glViewport(0, 0, screenWidth, screenHeight)
glEnable(GL_DEPTH_TEST)
//Load OpenGL Shaders
Shader modelShader("model.vs", "model.frag");
Shader basicShader("default.vs", "default.frag");
Shader lightShader("light.vs", "light.frag")
Model ourModel("models/Emily.obj");
//Mesh to draw in 2nd View Port
Mesh outputModel = ourModel.getMesh(0);
//Mesh to draw in 3rd view Port
Mesh outputModel2 = outputModel;
//Load Models for Blendshape Model
Blendshape blend("models/Brow_lower_l.obj", ourModel.getVertices(0));
blend.addShape("models/Brow_raise_l.obj");
blend.addShape("models/Brow_lower_r.obj");
blend.addShape("models/Brow_raise_r.obj");
blend.addShape("models/suck.obj");
blend.addShape("models/mouth_PullOpen.obj");
blend.addShape("models/mouth_Pull_l.obj");
blend.addShape("models/mouth_Pull_r.obj");
blend.addShape("models/mouth_LowerLipDepress.obj");
blend.addShape("models/pucker.obj");
blend.addShape("models/stretch.obj");
blend.addShape("models/rotateL.obj");
blend.addShape("models/rotateR.obj");
blend.addShape("models/tiltL.obj");
blend.addShape("models/tiltR.obj");
//Define PCA Blendshapes
Blendshape pcaBlend(blendshapeSolve(blend, EigModels[0]));
std::vector<Vertex> pcaBlendModel;
for (unsigned int j = 1; j < EigModels.size(); j++) {
    pcaBlendModel = blendshapeSolve(blend, EigModels[j]);
    pcaBlend.addShape(pcaBlendModel);
}
//Find Bounding Sphere of Model for the camera lookat vector and Position of the light Source
glm::vec3 min = ourModel.getMinValues();
glm::vec3 max = ourModel.getMaxValues();
```

```
glm::vec3 center = glm::vec3((max.x + min.x) / 2, (max.y + min.y) / 2, (max.z + min.z) / 2);
        float rad = glm::length(max - center);
        glm::vec3 lightPos(center.x, center.y, center.z + 3 * rad);
        //Define Rectangle for the Webcam to be mapped onto
        GLfloat vertices[] = {
            // Positions          // Colors           // Texture Coords
            1.0f,  1.0f, -1.0f,   1.0f, 0.0f, 0.0f,   1.0f, 1.0f,   // Top Right
            1.0f, -1.0f, -1.0f,   0.0f, 1.0f, 0.0f,   1.0f, 0.0f,   // Bottom Right
            -1.0f, -1.0f, -1.0f,   0.0f, 0.0f, 1.0f,   0.0f, 0.0f,   // Bottom Left
            -1.0f,  1.0f, -1.0f,   1.0f, 1.0f, 0.0f,   0.0f, 1.0f    // Top Left
        };
        GLuint indices[] = {
            0, 1, 3, // First Triangle
            1, 2, 3  // Second Triangle
        };
        //Buffer
        GLuint VBO, VAO, EBO;
        glGenVertexArrays(1, &VAO);
        glGenBuffers(1, &VBO);
        glGenBuffers(1, &EBO);
        //put vertices in the buffer
        glBindVertexArray(VAO);
        glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER,sizeof(indices),indices,
GL_STATIC_DRAW);
        //Position att
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat), (GLvoid*)0);
        glEnableVertexAttribArray(0);
        //Color Att
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat), (GLvoid*)(3 *
sizeof(GLfloat)));
        glEnableVertexAttribArray(1);
        {/Text Att
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat), (GLvoid*)(6 *
sizeof(GLfloat)));
        glEnableVertexAttribArray(2);
            glBindVertexArray(0);
            GLuint texture1;
            glGenTextures(1, &texture1);
            glBindTexture(GL_TEXTURE_2D, texture1);
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
            // Set texture filtering
            glTexParameteri(GL_TEXTURE_2D,  GL_TEXTURE_MIN_FILTER,
GL_LINEAR);
            glTexParameteri(GL_TEXTURE_2D,  GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
            // Load, create texture and generate mipmaps
            glBindTexture(GL_TEXTURE_2D, 0);
            /Define /Cube used for the light
            GLfloat cube[] = {..}
            // First, set the container's VAO (and VBO)
            GLuint VBO2, lightVAO;
```

```
glGenVertexArrays(1, &lightVAO);
glGenBuffers(1, &VBO2);
glBindBuffer(GL_ARRAY_BUFFER, VBO2);
glBufferData(GL_ARRAY_BUFFER, sizeof(cube), cube, GL_STATIC_DRAW);
glBindVertexArray(lightVAO);
// Position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), (GLvoid*)0);
glEnableVertexAttribArray(0);
glBindVertexArray(0);
float fov = 45.0f;
//Storage for the Output of the Blendshape Sum
std::vector<Vertex> BlendedPosition;
std::vector<Vertex> OldPosition;
float g_interp = 0;
//Get current time used for time analysis of the program
double lastTime = glfwGetTime(), startTime, endTime;
int nbFrames = -1;
column_vector expressionVector(16);
column_vector rotationVector(4);
column_vector expressVector(12);
expressionVector = 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0;
rotationVector = 0.0, 0.0, 0.0, 0.0;
expressVector = 1.0,0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0;
while (!glfwWindowShouldClose(window))
{//Calculate the time per frame
double currentTime = glfwGetTime();
nbFrames++;
if (currentTime - lastTime >= 1.0) {
    //printf("%f ms/frame\n", 1000.0 / double(nbFrames));
    nbFrames = -1;
    lastTime += 10;}
glfwPollEvents();
glClearColor(0.05f, 0.05f, 0.05f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);
BlendedPosition.clear();
//Solve for the Final Model
BlendedPosition = blendshapeSolve(blend, expressionVector);
outputModel.reshapeMesh(BlendedPosition);
for (int viewp = 0; viewp < 3; viewp++) {
//Split the Screen into 3 Sections 1st displays the Webcam2nd the Final
Blendshape Sum 3rd the PCA Blendshape Sum
    if (viewp == 0) {
    glViewport(0, 0, screenWidth / 3, screenHeight)}
    if (viewp == 1) {
glViewport(screenWidth / 3, 0, screenWidth / 3, screenHeight);}
    if (viewp == 2) {
glViewport(screenWidth* 2/3, 0, screenWidth /3, screenHeight);
    }
    glClear(GL_DEPTH_BUFFER_BIT);
    if (viewp == 0) {
    //Matrices To store current frame from the webcam
        cv::Mat temp;
        cv::Mat temp_small;
```

```
cap >> temp;
//Storage for the faces and face shapes detected
std::vector<rectangle> faces;
std::vector<full_object_detection> shapes;
rectangle r;


//resize webcam image to be small so the facial detection can be computed quicker
cv::resize(temp, temp_small, cv::Size(), 1.0 / FACE_DOWNSAMPLE_RATIO, 1.0 / FACE_DOWNSAMPLE_RATIO);
cv_image<bgr_pixel> cimg_small(temp_small);
cv_image<bgr_pixel> cimg(temp);
//Detect Faces every Frame
if(nbFrames%1==0 ) faces = detector(cimg_small);
//Find the Pose for a detected faces
for (unsigned long i = 0; i < faces.size(); ++i) {
if (i == 1) {
break;}
r = rectangle((long)(faces[i].left() * FACE_DOWNSAMPLE_RATIO) - 5
(long)(faces[i].top() * FACE_DOWNSAMPLE_RATIO) + 5,
(long)(faces[i].right() * FACE_DOWNSAMPLE_RATIO) + 5,
(long)(faces[i].bottom() * FACE_DOWNSAMPLE_RATIO) - 5
);


currentFaceShape = pose_model(cimg, r);
findTransform(findLandmarkRectangle(currentFaceShape), findLandmarkRectangle(exampleShapes[0]));
endTime = glfwGetTime();
//std::cout << "   Time Taken for Landmark tracking = " << endTime - startTime << std::endl;
startTime = glfwGetTime();
//Solve for expression Vector
expressionVector(0) = 1.0;
find_min_bobyqa(minFunction, expressionVector, 33, uniform_matrix<double>(16, 1, 0.0), uniform_matrix<double>(16, 1, 1.0), 0.075, 0.01, 30000);
expressionVector(0) = 1.0;
//Repeat Optimization with reseted first element and smaller stoping criteria
double error = find_min_bobyqa(minFunction, expressionVector, 33, uniform_matrix<double>(16, 1, 0.0), uniform_matrix<double>(16, 1, 1.0), 0.05, 0.005, 30000);
//Testing
endTime = glfwGetTime();
//std::cout << "All-in-One method" << std::endl;
//std::cout << "Average Error = " << error / exampleShapes[0].num_parts();
//std::cout << "   Time Taken = " <<  endTime-startTime << std::endl;
roundColumnVector(expressionVector);
shape2 = getFinalShape(expressionVector)
//Solve for PCA exprssion Vectors
startTime = glfwGetTime();
cv::Mat pcaSolve = pcaFaceSolve(currentFaceShape);
std::vector<dlib::point> shapePCA= getFinalPcaShape(pcaSolve);
endTime = glfwGetTime();
//Testing
//std::cout << "PCA method" << std::endl;
error = pcaError(shapePCA);
std::cout << "Average Error = " << error / exampleShapes[0].num_parts();
std::cout << "   Time Taken = " << endTime - startTime << std::endl;
```

```
//Solve for Region based expression Vector
startTime = glfwGetTime();
column_vector regionExpressionVector(16);
//Solve for Tilting and rotation
find_min_bobyqa(rotationMinFunction, rotationVector, 9, uniform_matrix<double>(4, 1, 0.0), uniform_matrix<double>(4, 1, 1.0), 0.075, 0.01, 30000);
regionExpressionVector = concatColVec(expressVector, rotationVector);
regionSolve = getFinalShape(regionExpressionVector);
//Solve for facial expression
error = find_min_bobyqa(regionMinFunction, expressVector, 15, uniform_matrix<double>(12, 1, 0.0), uniform_matrix<double>(12, 1, 1.0), 0.075, 0.01, 30000);
regionExpressionVector = concatColVec(expressVector, rotationVector);
//Testing
endTime = glfwGetTime();
//std::cout << "Region method" << std::endl;
//std::cout << "Average Error = " << error / exampleShapes[0].num_parts();
//std::cout << "   Time Taken = " << endTime - startTime << std::endl;
regionSolve = getFinalShape(regionExpressionVector);
//render_face(temp, currentFaceShape, shape2, regionSolve); // Show the Tracked Landmarks, All in One Landmarks and PCA landmarks
render_face(temp, currentFaceShape, shape2,regionSolve); // Show the Tracked Landmarks, All in One Landmarks and Region Landmarks
shapes.push_back(currentFaceShape);
*//Solve for the PCA BlendShapes
BlendedPosition.clear();
BlendedPosition = blendshapeSolvePca(pcaBlend, matToColumn(pcaSolve));
outputModel2.reshapeMesh(BlendedPosition);
*///Solve for the Region Based Blendshapes
BlendedPosition.clear();
BlendedPosition = blendshapeSolve(blend, regionExpressionVector);
outputModel2.reshapeMesh(BlendedPosition);}
//Apply current frame as a texture for the Rectangle in the first viewport
cv::flip(temp, temp, 0);
        glBindTexture(GL_TEXTURE_2D, texture1);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB,temp.cols, temp.rows, 0, GL_BGR, GL_UNSIGNED_BYTE, temp.ptr());
glGenerateMipmap(GL_TEXTURE_2D);
basicShader.Use();
glActiveTexture(GL_TEXTURE0);
glUniform1i(glGetUniformLocation(basicShader.Program, "ourTexture1"), 0);
glBindVertexArray(VAO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
glBindVertexArray(0);
glBindTexture(GL_TEXTURE_2D, 0);
if (viewp == 1) {
/Draw the Final Blendshape Sum
modelShader.Use();
GLint objectColorLoc = glGetUniformLocation(modelShader.Program, "objectColor");
GLint lightColorLoc = glGetUniformLocation(modelShader.Program, "lightColor");
GLint lightPosLoc = glGetUniformLocation(modelShader.Program, "lightPos");
GLint viewPosLoc = glGetUniformLocation(modelShader.Program, "viewPos");
glUniform3f(objectColorLoc, 1.0f, 0.6f, 0.31f);
glUniform3f(lightColorLoc, 1.0f, 1.0f, 1.0f);
glUniform3f(lightPosLoc, lightPos.x, lightPos.y, lightPos.z);
```

```
glUniform3f(viewPosLoc, center.x, center.y, center.z + 2 * rad)
float aspect = (float)screenWidth / (float)(2 * screenHeight);
glm::mat4 view = glm::lookAt(glm::vec3(center.x, center.x, center.x + 2 * rad), center, glm::vec3(0.0f, 1.0f, 0.0f));
glm::mat4 projection = glm::perspective(fov, aspect, 0.1f, 100.0f);
glUniformMatrix4fv(glGetUniformLocation(modelShader.Program, "projection"), 1, GL_FALSE, glm::value_ptr(projection));
glUniformMatrix4fv(glGetUniformLocation(modelShader.Program, "view"), 1, GL_FALSE, glm::value_ptr(view));

glm::mat4 model;
//model = glm::translate(model, glm::vec3(0.0f, -1.75f, 0.0f)); // Translate it down a bit so it's at the center of the scene
model = glm::rotate(model, -0.15f, glm::vec3(1.0f, 0.0f, 0.0f));
model = glm::scale(model, glm::vec3(1.0f, 1.0f, 1.0f));	// It's a bit too big for our scene, so scale it down
glUniformMatrix4fv(glGetUniformLocation(modelShader.Program, "model"), 1, GL_FALSE, glm::value_ptr(model));
outputModel.Draw(modelShader);
lightShader.Use();
glUniformMatrix4fv(glGetUniformLocation(lightShader.Program, "projection"), 1, GL_FALSE, glm::value_ptr(projection));
glUniformMatrix4fv(glGetUniformLocation(lightShader.Program, "view"), 1, GL_FALSE, glm::value_ptr(view));
model = glm::scale(model, glm::vec3(0.2f));	//small cube light
glUniformMatrix4fv(glGetUniformLocation(lightShader.Program, "model"), 1, GL_FALSE, glm::value_ptr(model));
glBindVertexArray(lightVAO);
glDrawArrays(GL_TRIANGLES, 0, 36);
glBindVertexArray(0);
}
				if (viewp == 2) {
//Same as viewp=1 but with the second blendshape mesh and light colour code is skipped}
			}
			glfwSwapBuffers(window);
		}
		glDeleteVertexArrays(1, &VAO);
		glDeleteBuffers(1, &VBO);
		glDeleteBuffers(1, &EBO);
		glfwTerminate();
		return 0;
	}
	catch (std::exception& e)
	{
		std::cout << e.what() << std::endl;
	}
}
```