Real-Time Facial Motion Capture System

Master's Thesis

Steven Twist

MSc Computer Animation and Visual Effects

NCCA Bournemouth University

20th August 2010

Contents

List of Figures

Chapter 1

Introduction

Figure 1.1: A still frame from *Avatar* (2009), illustrating the extremely high quality of the rendering. (Image from DaCosta 2010).

It is of no surprise that *Avatar* (2009) won the 2010 Academy Award for *Best Visual Effects* (Oscars 2010). For me, personally, *Avatar* is hugely inspirational on many levels: the quality of the rendering is stunning (see Figure 1.1); the character rigs, utilising full muscle simulations (Teo 2010), are the best I have ever seen; the stereoscopic 3D technology is used to such an art, creating a fully immersive experience, as opposed to being used for cheap gimmicks - James Cameron yet again pushing forward the development of digital technology (Fox 2010).

Despite the countless other achievements in visual effects made on *Avatar*, the aspect of the production process that intrigued me the most was the innovative use of the performance capture stage, as Leonard Teo (2010, p. 36) explains:

> "During their performance capture, actors' movements are retargeted to the CG characters which can then be viewed in real-time within a virtual set... Traditionally, a character's motions are captured and the digital environment added later in post-production. The new technique displays a visualisation of the characters, 3D environments and objects to the camera operator in real-time. "Basically, you're shooting the film on a motion-capture stage in a way that's similar to a live-action film," says [Dan] Lemmon[, FX supervisor, Weta Digital]. "You're filming things, capturing the scene as you're seeing it through a monitor in real-time.""

I was captivated by the possibilities of the technology, especially the value a real-time facial motion capture system could bring to any production requiring facial animation.

As with *Avatar*, by allowing a director and/or actor to see, in real-time, the motion capture driven CG characters, you give them full control over the performance. Incredibly useful for a project of such scope as *Avatar*, but in my opinion just as useful on smaller productions. Allowing the director to give immediate feedback based on the CG face, as opposed to just the actor's face, allows them to know if an expression is over or under exaggerated, for example. This makes the motion capture process even more interactive, speeding up the time between iterations on a particular shot because you no longer need to wait for an offline calculation to convert the input data into animation.

In the world of video games, if a game developer was to directly integrate the real-time facial motion capture technology into their rendering engine, they could perhaps see exactly how the motion capture would look in the final product (with the final facial rigs, shaders, lighting etc.), whilst directing the actor. Again, this speeds up the time between iterations on a particular shot, allowing the developer to quickly know whether the captured data will meet their requirements or not.

Moving further 'outside the box', whilst the current technology is not best suited to this purpose (due to the use of obtrusive head-mounted hardware, and markers being painted on the actor's face - see Chapters 2 and 3), the possibilities for analysing the motion of a video game player's face, in real-time, and utilising that information as part of the game mechanics (perhaps in a manner similar to Microsoft's "Kinect" technology (Emery 2010)), could lead to interesting developments in the video game industry.

Thus, the goal for this Master's project was to develop a real-time facial motion capture system. The system would involve both hardware and software development, and must be capable of analysing a live video stream of an actor's face in order to animate any CG face (not just one of close resemblance to the actor) in real-time. In addition, unlike existing professional systems (a similar, non-real-time system from Meta Motion (2010), for example, costs just under $10,000) the hardware would need to be developed on a tight budget.

This thesis outlines the design and implementation of such a real-time facial motion capture system.

1.1 Thesis Overview

Chapter 2: Previous Work: An overview of existing facial motion capture techniques.

Chapter 3: Design: Presents, in detail, the program architecture and algorithm design, including any relevant mathematics. This chapter initially presents an overview of the entire system, and then details each module in its own section.

Chapter 4: Implementation: Details the techniques used to implement each of the modules presented in Chapter 3. Similarly to that chapter, this chapter is partitioned into one section per module.

Chapter 5: Results and Analysis Assesses the performance of the developed system in three key areas: temporal, cost and visual. Analyses any shortcomings in each of these areas, and suggests steps for improving on these shortcomings.

Chapter 6: Future Work Suggests extensions to the project, in addition to recapping any improvements suggested in Chapter 5.

Chapter 2

Previous Work

There have been many different approaches to facial motion capture in its short history (Pighin 2006).

2.1 Mesh Reconstruction Techniques

Some techniques reconstruct a highly detailed 3D mesh of the actor's face at every frame. "Universal Capture" (Borshukov et al. 2005) is one such example, developed for *The Matrix Reloaded* (2003). In Universal Capture, optical flow is used to track the motion of pixels in video data from a number of camera angles. Combined with a high-resolution, neutrally-posed base mesh, and triangulation techniques, the 3D motion of the vertices in the base mesh, across each frame, can be calculated. An alternative approach (Ma et al. 2008) uses structured light patterns to analyse and compute the 3D form of the face at a given frame, along with a normal map of the face for capturing the finer details.

Both of these approaches, whilst very effective, are drastically inappropriate for this project. Firstly, both techniques require expensive hardware setups. Borshukov et al. (2005, p. 1) describe the "sophisticated arrangement of Sony/Panavision HDW-F900 cameras and computer workstations" required for Universal Capture. Ma et al. (2008) requires specialist projectors to generate the structured light patterns. Such hardware was out of the scope of this Master's project, rendering both techniques inappropriate.

In addition, both techniques generate an animated 3D duplicate of the actor's face. Such techniques are extremely useful for films such as *The Matrix Reloaded* (2003) where digital doubles of lead actors were required, and where such doubles needed to match the real actors as closely as possible (Borshukov et al. 2005). However, the goal of this project is to develop a facial motion capture system flexible enough to animate any face, regardless of the resemblance to the actor (Chapter 1), and for this reason also, both techniques are inappropriate.

2.2 Other Techniques

Broadly speaking, other common approaches to facial motion capture can be categorised as tracking the face as a series of points (or 'markers') in space, and analysing these markers to drive the motion of the animated model (Pighin 2006).

There are a variety of techniques which can be used to calculate the marker locations, and an equally large number of techniques which can be used to generate the motion of the animated model from these marker locations.

2.2.1 Calculation of Marker Locations

Perhaps the most obvious technique for calculating the marker locations is to track physical markers on the actor's face, known as "marker-based motion capture" (ibid, p. 4). The tracking of physical markers is yet another broad research topic. Some systems track the marker locations in 3D space by using infrared cameras to detect infrared light reflecting off of the markers. Such a system was used by Sony Pictures Imageworks on *The Polar Express* (2004) (Bennett 2005) and *Monster House* (2006) (Havaldar 2006). Other approaches use a more simplistic analysis of a 2D video to track the marker locations in 2D space (Barker 2005).

Another technique for calculating the marker locations is "appearance-based face-tracking" (Pighin 2006, p. 2), as is used in ImageMetrics' commercial face tracking system (ibid.). Appearance-based tracking is a marker-less approach to tracking the face, which uses a "principal component (PCA) model... trained on variations in face shape [to generate a] ...set of invisible 'virtual' markers on the face." (ibid., p. 2).

It was decided that a 2D marker-based approach would be most appropriate for the scope of the project. Tracking markers in 2D requires a simple hardware setup (as will later be discussed), and can be achieved with efficient algorithms (Barker 2005), well suited for the real-time nature of this project.

2.2.2 Analysis of Marker Locations

Once the marker locations have been acquired, whether through marker-based or appearance-based tracking, they then need to be analysed to generate the motion of the animated model.

One approach is to generate a face rig which has a corresponding set of control points for each marker location. When the control points are moved, the underlying face model deforms. By moving each control point to the 3D location of the corresponding marker, the face model deforms to match the actor's face. One such example of this approach is implemented by Bickel et al. (2008).

However, for the control points to simply be repositioned to the exact 3D location of the corresponding markers, the face model needs to closely resemble the actor's face. Given that the goal of this project is to develop a facial motion capture system flexible enough to animate any face, regardless of the resemblance to the actor (Chapter 1), this technique is inappropriate.

In order to be able to drive any CG face model with an actor's face, one needs to break the connection between the geometry of the actor, and the geometry of the animated model. Sagar (2006, p. 1) describes how Weta Digital implemented such a system for *King Kong* (2005):

> "The technique used for King Kong breaks the direct geometric connection by first fitting the motion capture data to an expression space, which is then mapped to another expression space defined by animation control presets on the creature facial puppet.
>
> The expression space is effectively a Gorilla specific superset of Ekman and Friesen's Facial Action Coding System (FACS) which groups facial muscles which work as units. It represents an effective alphabet of facial expression; any expression can be formed by combinations of the individual elements.
>
> Expression space has several benefits. It provides a transcription of the communicative content of the performance, and is ideal to map in an art-directable way to a topologically different character. It provides an intuitive format for motion editing and making performance changes. Because the face is constrained in its motion, fitting to an expression space significantly reduces noise and allows for intelligent filtering.".

Whilst the above description might sound quite complicated, the core technique is actually very simple. Rather than analysing the motion of the markers to determine how the geometry

has changed, Weta Digital analyse the motion of the markers to determine how the expression of the face has changed. They do this by determining which muscles in the face are acting to cause the displacement in the marker locations (Teo 2010). On the animated model, a similar set of muscles are driven by this data, which results in the animated face deforming to show the same expression, regardless of the geometry.

The approach used for *King Kong* was used as a basis for the facial motion capture on *Avatar*, to "retarget the motion data onto faces that don't match directly - in this instance, the Na'vi" (Teo 2010, p. 33).

Havaldar (2006) mentions that a muscle-based system was also used on *Monster House*(2006), and similarly to Sagar's (2006) approach on *King Kong*, it is based heavily on the Facial Action Coding System (Ekman et al. 2002).

Choe et al. (2001) describes one approach to determining the "muscle actuation parameters" (i.e. how much each muscle is firing) based on the location of the markers. A heavily related approach considers a facial expression to be represented as a series of blendshapes, rather than muscle deformations, and calculates the blendshape weights, rather than the muscle actuation parameters (Chuang et al. 2002, and Joshi et al. 2006).

In both approaches, because the muscle actuation/blendshape weight parameters are calculated based on the actor's face, and then applied to the animated face, they break the connection between the geometry of the actor's face and the geometry of the animated model. Thus, these approaches are ideal for this project, and it was decided to use a blendshape based system for mapping the marker data to a set of blendshape weights to drive the animated model.

2.2.3 Helmet Mounted Cameras

Figure 2.1: Helmet mounted cameras were used to aid the motion capture process in *Avatar* (2009). (Images from Fox 2010).

Havaldar (2006) gives an excellent overview of the entire facial motion capture pipeline used by Sony Pictures Imageworks on *Monster House* (2006). Their pipeline tracks a large number of markers in 3D space, and as a result, a number of problems occur, each requiring quite complicated solutions.

Firstly, occlusions in the marker data (despite a large number of cameras), where one performer blocks the markers of another performer from the view of the cameras, result in incomplete data sets for some frames.

Secondly, to solve for facial motion, the overall rigid transformation of the head has to be determined. The inverse of this transformation must then be applied to the facial markers

in order to analyse just the facial motion.

Finally, each marker requires labelling so that each marker can be uniquely identified. The process of analysing the marker positions to determine their labels is fairly complex.

All three of these above complications can be completely avoided by using a "Helmet Mounted Camera set-up" (Firestone 2008). In such a setup, one or more cameras are attached to a helmet, and aimed at the actor's face. With the cameras so close to the actor's face, the markers never become occluded by other objects, solving problem number one. Because the camera is rigidly attached to the actor's head, the relative position of the markers in the video footage are unaffected by rigid transformations of the head, solving problem number two. Removing occlusions and rigid head transformations simplifies the process of labelling each of the markers, solving the final problem.

In addition, by utilising a helmet mounted camera setup, each actor's facial data is kept separated, allowing for many actors to be captured at once, without risk of confusion between actors (ibid.).

ImageMovers Digital developed a helmet mounted camera setup with Vicon for *A Christmas Carol* (2009) which utilised 4 cameras to track facial markers in 3D (Firestone 2008). A similar setup, nick-named the "headrig" was used on *Avatar* (2009) (see Figure 2.1) (Firestone 2008):

> ""It also employs a skullcap helmet, but uses a single camera mounted from the side like a headset microphone," [Glenn Derry] says.
>
> While lacking the multipositional perspective for tracking, as well as the resolution of a four-camera system over a one camera system, it makes up for it with realtime facial motion solving and playback, which Derry explains is what is most important for their application, and provides them with the results they need in their streamlined, realtime workflow."

For *Avatar*, the headrig used a life cast of the actor's head, combined with a laser scan, to produce a tightly fitting scull cap (Discovery 2009). Whilst developing a custom scull cap for an actor is beyond the scope of this project, for the benefits listed above it was decided to develop and use a basic helmet mounted camera as the input device for this project.

In addition to simplifying the process of facial motion capture on *Avatar*, helmet mounted cameras proved invaluable for providing video reference of the actor's faces, as Andy Jones, animation director at Weta Digital explains:

> "Jim [Cameron] shot a ton of HD reference of his actors and that ended up being the saving grace for the animation process. Once the facial solve came out of motion capture, we would submit side-by-side renders of the real actor and his avatar/Na'vi counterpart, and tweak and adjust the facial animation to get every last nuance into the performance." (Teo 2010, p. 33)

Chapter 3

Design

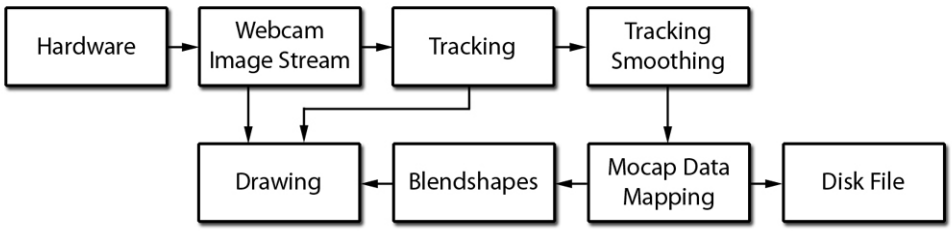


Figure 3.1: Diagram illustrating the architecture of our facial motion capture system.

Figure 3.1 illustrates the overall architecture of our facial motion capture system. Facial motion capture can be thought of as a pipeline, beginning with some form of input data from a hardware device, and ending with the generation of animation data in some form.

The facial motion capture pipeline can be subdivided into smaller modules. Each module encapsulates a specific task, receiving certain data as input, processing that data, and outputting new data for the consumption of other modules.

The architectural and algorithmic design of each module is discussed in its own section, below. Chapter 4 contains similar sections for each module, detailing the techniques used to implement each of the modules in our facial motion capture system.

3.1 Hardware

The entry point to any motion capture system is some form of hardware device which captures relevant data from the actor, and streams it to the motion capture software (Pighin 2006). In the case of our facial motion capture system, our hardware device is a video camera aimed at the actor's face. This video camera must be capable of outputting a continual live video stream to a desktop PC. For our current discussion of system design, the format of the video stream is unimportant.

Section 2.2.3 introduced the idea of helmet mounted cameras, and described the benefits that they bring to a facial motion capture system. Our facial motion capture system will utilise such a helmet mounted camera setup, where the video camera filming the actor's face is rigidly attached to the actor's head, via a helmet.

This gains us all the benefits mentioned in Section 2.2.3, namely: the removal of marker occlusions, due to the proximity of the camera to the actor's head; the removal of any rigid transformations of the actor's head from the relative transformations of the facial markers in the video feed; and the subsequent ease of marker labelling.

The hardware setup used by our facial motion capture system has a significant impact on the rest of the pipeline. By using a single video camera, the software component of the pipeline receives only 2D image data of the face. Thus all other operations performed on that data will occur in two dimensions. The decision to utilise a single camera, as opposed to a multi-camera setup, was made for three reasons:

- Most facial motion occurs in a 2D plane, with relatively little facial motion occurring in a forwards-backwards direction, in terms of the orientation of the actor's head (Ekman et al. 2002).
- To reduce overall hardware complexity, and thus costs, ensuring that the hardware could be built on a tight budget (which is one of the goals of this project (Chapter 1)).
- To allow for real-time facial motion capture solving by reducing the amount of input data (Firestone 2008), a key goal of this project (Chapter 1).

3.2 Webcam Image Stream

The *Webcam Image Stream* module receives video data from the video camera hardware as a series of frames. The format of this data will be dependant on the hardware used. This module has a simple purpose: to take each of the input video frames, and convert them to a RGB bitmap data format for subsequent processing by the *Tracking* module (Section 3.3). In addition, the *Webcam Image Stream* module draws the RGB frames to the screen, such that the video feed of the actor's face is displayed as part of the user interface.

3.3 Tracking

The *Tracking* module receives an RGB bitmap frame of video data from the *Webcam Image Stream* module (Section 3.2), and is responsible for detecting and tracking a set of facial markers in that frame, outputting the 2D location of each facial marker. In addition, the *Tracking* module draws the marker locations over the video feed of the actor's face to allow the end user to assess the quality of the tracking data.

The *Tracking* module was implemented for a previous project, and thus its design and implementation are covered in detail in that project's report (Twist 2010). That report has been reprinted, for convenience, in Appendix A of this thesis.

3.4 Tracking Smoothing

High-frequency, low-amplitude noise (known as "shake") in the marker data is introduced as a result of both the tracking process (Twist 2010) and due to physical shake in the camera, introduced by small reverberations in the hardware setup caused by the motion of the actor's head and jaw.

This shake was ignored until the rest of the pipeline was implemented, at which time it was determined that the shake had a detrimental impact on the animation quality (as later mentioned in Section 5.3).

Thus, it was decided to pass the tracked marker data through a *Tracking Smoothing* module, which acts as a low-pass filter (Wikipedia 2010c), removing this high frequency noise from the motion signal of the markers. For every frame of marker data received by the *Tracking Smoothing* module, a frame of smoothed marker data is output to the *Motion Capture Data Mapping* module (Section 3.6).

3.5 Blendshapes

The next stage in the facial motion capture pipeline after *Tracking Smoothing* (Section 3.4) is *Motion Capture Data Mapping* (Section 3.6). However, it is important to understand exactly the type of facial rig that needs to be driven by the motion capture data, in order to understand the requirements of the *Motion Capture Data Mapping* module. Therefore, we will first discuss the *Blendshapes* module.

A key goal of this facial motion capture system is to animate any CG face, not just one of close resemblance to the actor (Chapter 1). As discussed extensively in Section 2.2.2, in order to animate any CG face one needs to break the connection between the geometric nature of the actor's face, and the geometry of the animated model (Sagar 2006). As earlier discussed (Section 2.2.2), there are two main techniques currently being used in production to achieve such a detachment: muscle-based (Choe et al. 2001, Havaldar 2006, and Sagar 2006), and blendshape-based (Chuang et al. 2002, and Joshi et al. 2006).

Both techniques are heavily related. The muscle-based approach analyses the motion capture data to determine which facial muscles are acting, and then drives the corresponding muscles in the facial rig (Sagar 2006). The blendshape-based approach analyses the motion capture data to determine which blendshapes are acting, and then drives the corresponding blendshapes in the facial rig.

Blendshape-based facial rigs are arguably much more simplistic than muscle-based facial rigs, given that they consist of a simple linear combination of poses (Section 3.5.1). Given the scope of this project, it was decided to use a blendshape-based facial rig, as we have previous experience working with blendshapes and have access to existing blendshape-based facial rigs for use with this project. In addition, it is our understanding that blendshapes are computationally less expensive than muscle simulations (again, due to the simplicity of the underlying mathematics), and thus would be better suited to a real-time application.

3.5.1 Blendshape Theory

A blendshape-based facial rig consists of a rest pose, and a number of target poses (Ritchie et al. 2005). All poses are provided by the artist as topologically identical models.

However, rather than simply storing the true locations of every vertex in the target pose mesh, a target blendshape is constructed by calculating the vector offset of each vertex in the target pose from the corresponding rest pose vertex (Lorach 2007):

$$\mathbf{B}_i = T_i - R_i \tag{3.1}$$

where T_i is the position of vertex i in the target pose, R_i is the position of vertex i in the rest pose, and $\mathbf{B}_i$ is the vector offset of vertex i.

In this way, if one took the rest pose and added to each vertex the corresponding offset vector of a particular target blendshape, the resultant model would be that of the target pose initially used to create that blendshape.

Each target blendshape has an associated weight parameter. Given a set of weights, the resultant model is calculated as the weighted sum of all the blendshapes added to the rest pose (Joshi et al. 2006):

$$M_i = R_i + \sum_{j=1}^{n} w_j \mathbf{B}_{ji} \tag{3.2}$$

where M_i is the resultant position of vertex i, w_j is the weight parameter for blendshape j, $\mathbf{B}_{ji}$ is the vector offset of vertex i in blendshape j, and n is the number of blendshapes.

We can rewrite this as:

$$M_i = R_i + \mathbf{O}_i \tag{3.3}$$

where $\mathbf{O}_i$ is the vector offset applied to vertex i:

$$\mathbf{O}_i = \sum_{j=1}^{n} w_j \mathbf{B}_{ji} \tag{3.4}$$

Rather than representing our mesh as a vector of 3-tuples, where each vertex is a single component in that vector, and where v is the number of vertices in our mesh, i.e.:

$$\mathbf{M} = \begin{bmatrix} [x_1, y_1, z_1] \\ [x_2, y_2, z_2] \\ ... \\ [x_v, y_v, z_v] \end{bmatrix}$$

Let us represent our mesh as a vector of scalar values, where the x, y and z coordinates of a vertex are consecutive components of that vector, and where m is the number of components in the vector (i.e. $m = v$ * the dimensionality of the vertices; in this case, $m = v * 3$):

$$\mathbf{M} = [x_1, y_1, z_1, x_2, y_2, z_2, ..., x_v, y_v, z_v]^T \tag{3.5}$$

If we also represent each blendshape $\mathbf{B}_j$ (i.e. offset between the target pose and rest pose), the resultant vector offset $\mathbf{O}$, and the rest pose mesh $\mathbf{R}$ in a similar format, equations 3.3 and 3.4 can now be rewritten as vector equations:

$$\mathbf{M} = \mathbf{R} + \mathbf{O} \tag{3.6}$$

where $\mathbf{M}$, $\mathbf{R}$ and $\mathbf{O}$ are all vectors containing m components, and $\mathbf{O}$ is defined as:

$$\mathbf{O} = \sum_{j=1}^{n} w_j \mathbf{B}_j \tag{3.7}$$

In other words, our vector offset is defined as the weighted sum of each of our blendshape vectors $\mathbf{B}_j$. The importance of rewriting our blendshape equation in this way will become apparent in Section 3.6.1.

Taking these equations one step further, we can actually rewrite equation 3.7 as a matrix equation:

$$\mathbf{O} = \mathbf{B} * \mathbf{w} \tag{3.8}$$

where $\mathbf{w}$ is a n-vector of weight parameters (recall that n is the number of blendshapes), and $\mathbf{B}$ is a m (rows) by n (columns) matrix (recall that m is the number of vertices multiplied by the dimensionality of our vertices), where each column of $\mathbf{B}$ represents one of the blendshape vectors. The importance of rewriting our blendshape equation in this way will become apparent in Section 3.6.2.

Hopefully it is apparent that when all weight parameters are set to a value of 0.0, the resultant model is that of the rest pose. When the weight parameter of any one blendshape is set to 1.0, the resultant model is that of the corresponding target pose. By varying the weights of each blendshape, one can increase or decrease the influence of each target pose over the resultant model. A set of target poses are generated by an artist in an attempt to allow any facial expression to be created by selecting the appropriate set of weight parameters (Ritchie et al. 2005). We will be revisiting the topic of an "ideal" set of target poses at many points throughout this thesis.

Thus, the *Blendshapes* module requires a set of weights (1 weight per blendshape) as input, passed to it from the *Motion Capture Data Mapping* module (Section 3.6). It will take those weights and calculate the resultant head model, which is finally drawn to the screen in order for the end user to see the results of the motion capture process.

3.6 Motion Capture Data Mapping

The last of our modules to discuss, the *Motion Capture Data Mapping* module receives a set of marker locations for a given frame from the *Tracking Smoothing* module (Section 3.4) and must determine from those locations the appropriate set of weights to pass to the *Blendshapes* module (Section 3.5) such that the blendshape facial rig represents our actor's facial expression.

In Section 2.2.2 the concept of an "expression space" was introduced (Sagar 2006), whereby the motion capture data (i.e. marker locations) must be mapped to an expression space such that we can isolate the expression of the actor's face from the underlying geometry of the actor's face. If we can then map the data from this expression space to our facial rig parameters (i.e. our blendshape weights), we can transfer the expression of the actor onto our animated model.

How one defines this expression space, as well as the transformation of the marker data into that space, and the subsequent transformation from that space to the facial rig parameters, is possibly the most complicated part of a motion capture system of this nature. Our approach is inspired by Sagar (2006) and Havaldar (2006), who describe the use of the Facial Action Coding System (Ekman et al. 2002), along with introducing at a conceptual level the idea of an expression space mapping, as outlined above, but never detail the specifics. To understand our approach to defining this "expression space", and our subsequent solution to calculating the blendshape weights from a given set of marker locations, we must first introduce some basic vector space theory.

3.6.1 Vector Space Theory

The mathematics of vector spaces are quite involved (Comninos 2006). However, for our purposes, it is enough to understand the basic idea of a vector space. To ease our understanding, we will consider a simple 2D space (i.e. the XY plane) as an example of a vector space.

A vector space is a set of vectors, where the addition of any two vectors in the set results in a third vector that is also a member of the set, and the multiplication of any vector in the set with any scalar value results in a second vector that is also a member of the set (Comninos 2006). The XY plane *is* a vector space. It is a set of all the 2D vectors (i.e. any possible combination of x and y coordinates). Mathematically speaking:

> "The vectors of the two-dimensional (2D) Euclidean space E^2 are 2-tuples that are members of the set $\mathbf{S}^2 = \mathbb{R}^2$ " (Comninos 2006, p. 156)

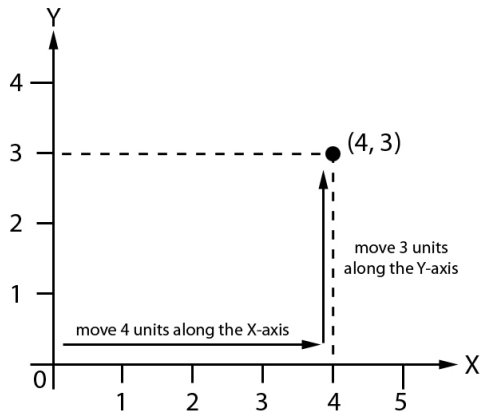


Figure 3.2: Diagram illustrating the coordinate system for navigating 2D space.

We know that we can represent any position in 2D space with an x coordinate and a y coordinate. We learn from a young age that the x coordinate instructs us how far we need to move along the X-axis, and the y coordinate how far we need to move along the Y-axis (see Figure 3.2). But what does this really mean, mathematically speaking?

Let **i** be a unit vector parallel to the X-axis (1, 0), and **j** be a unit vector parallel to the Y-axis (0, 1). The x and y coordinates are, mathematically speaking, the scalar values we need to multiply **i** and **j** by, respectively, to get the position vector of the desired location in space. Given the desired location (4, 3), we must multiply **i** by 4, and **j** by 3, to get the position vector of that location (see Figure 3.3):

$$\begin{aligned}\begin{bmatrix} x_i \\ y_i \end{bmatrix} &= 4\begin{bmatrix} 1 \\ 0 \end{bmatrix} + 3\begin{bmatrix} 0 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 4 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 3 \end{bmatrix} \\ &= \begin{bmatrix} 4 \\ 3 \end{bmatrix}\end{aligned}$$

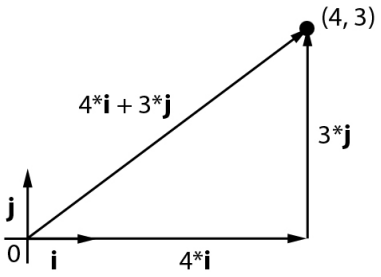


Figure 3.3: Diagram illustrating one possible set of basis vectors for 2D Euclidean space.

In terms of vector spaces, **i** and **j** are the *basis vectors* of our 2D vector space. By linearly combining these vectors, we can define any other vector in the vector space:

$$\mathbf{v} = x * \mathbf{i} + y * \mathbf{j}$$

Extending this idea to a m-dimensional space:

$$\mathbf{v} = s_1 * \mathbf{v}_1 + s_2 * \mathbf{v}_2 + ... + s_m * \mathbf{v}_m \text{ (Comninos 2006, p. 156)}$$

Which could equally be written:

$$\mathbf{v} = \sum_{j=1}^{m} s_j \mathbf{v}_j$$

The above equation bears a striking resemblance to our vector blendshape equation (equation 3.7). The offset that gets applied to the rest pose is calculated as the linear combination of each of the blendshapes. Much as the x coordinate acts as a multiplier of **i**, a blendshape weight acts as the multiplier of the corresponding blendshape.

Thus, we can think of our blendshapes as defining a basis for a vector space, and our weights as the coordinates of some location in that vector space. We earlier discussed in Section 3.5.1 that, given an "ideal" set of target poses, by selecting the correct weights we can generate any facial expression. Thus, we can think of our blendshape vector space as an "expression space" (Sagar 2006, p. 1). Any point in that multi-dimensional space, as specified by the weight parameters, gives us a particular facial expression.

But what constitutes an "ideal" set of target poses? Our target poses define the basis of our expression space. Mathematically speaking, there are two important properties that a set of basis vectors must possess, that we have yet to discuss. Namely: The basis vectors must fully span the vector space, and the basis vectors must be linearly independent (Comninos 2006).

Spanning Basis Vectors

> "The set that contains all possible linear combinations of the vectors in $\mathbf{V}_m$ is called the *span* of $\mathbf{V}_m$" (Comninos 2006, p. 157)

Mathematically speaking, the span of our basis vectors must represent the entire vector space (i.e. by linearly combining our basis vectors we must be able to generate every possible vector belonging to our vector space). If this property is not achieved, the basis vectors do not truly define a basis (Comninos 2006). Whilst mathematically the term 'basis' and 'basis vectors' refers to a set of vectors that represent the entire vector space, to simplify our discussions throughout the rest of this thesis, we are going to relax the definition to simply mean the vectors we choose to represent our vector space, whether they fully represent the space or not.

With our 2D example we can clearly see the problems encountered if our basis does not fully represent the vector space. Figure 3.4 illustrates a basis consisting of a single vector, **i**. The linear combination of our basis simply results in scaling **i**, or in other words, we can only represent vectors that are parallel to **i**. We cannot possibly represent points in space that lie off of the X-axis, because we have no way of 'travelling in the Y direction' (so to speak).

Extending this concept to three dimensions. If we have a 3D vector space, and 2 basis vectors **i** = (1, 0, 0) and **j** = (0, 1, 0), we can only represent points on the 2D plane of z = 0. We have no way to represent points that lie off of this plane, because we have no way of 'travelling in the Z direction' (see figure 3.5).

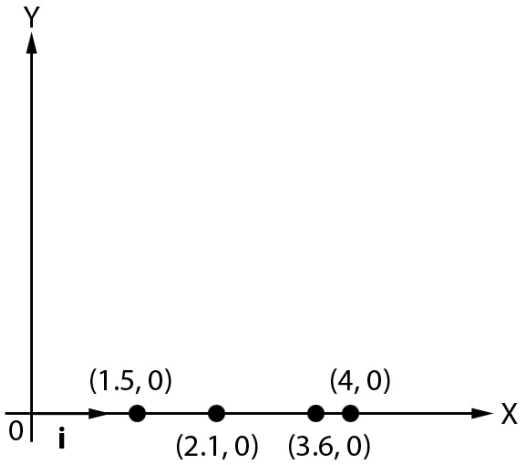


Figure 3.4: Diagram illustrating a basis that does not fully represent 2D Euclidean space.

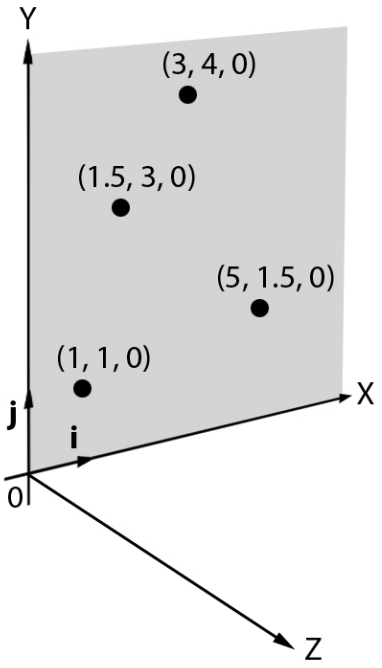


Figure 3.5: Diagram illustrating a basis that does not fully represent 3D Euclidean space.

As we have just seen, if our choice of basis vectors do not fully represent a vector space, we are unable to represent every point within that space. Applying this understanding to our expression space, if our choice of basis vectors do not fully represent that vector space, we are unable to represent every possible facial expression (because every point in our expression space represents a different facial expression).

Quite obviously we wish to be able to represent every facial expression the actor is capable of. In order to achieve this, we must choose a basis that fully represents our expression space. In other words, our choice of blendshapes for the 3D model must be able to combine to form any facial expression.

Linearly Independent Vectors

> "Given a set of vectors $\mathbf{V}_m = \{\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_m\}$ from the vector space $\mathbf{V}$, we say that the vectors in $\mathbf{V}_m$ are *linearly independent*... if the equation $s_1 * \mathbf{V}_1 + s_2 * \mathbf{V}_2 + ... + s_m * \mathbf{V}_m = \mathbf{0}$ only has a solution when $s_1 = s_2 = ... = s_m = 0$. We say that the vectors in $\mathbf{V}_m$ are *linearly dependent* if the equation $s_1 * \mathbf{V}_1 + s_2 * \mathbf{V}_2 + ... + s_m * \mathbf{V}_m = \mathbf{0}$ has a solution when at least one of the scalars s_1, s_2,...,s_m is non-zero. Thus, a set $\mathbf{V}_m$ from a vector space $\mathbf{V}$ is said to be linearly dependent if it contains at least one vector $\mathbf{v}_i$ that can be written as a linear combination of the remaining (m - 1) vectors in the set $\mathbf{V}_m$." (Comninos 2006, p. 157)

Mathematically speaking, our set of basis vectors must be linearly independent. That is to say that none of the basis vectors can be written as a linear combination of the other basis vectors. If this property is not achieved, the basis vectors do not truly define a basis (Comninos 2006). Whilst mathematically the term 'basis' and 'basis vectors' refers to a linearly independent set of vectors, to simplify our discussions throughout the rest of this thesis, we are going to relax the definition to simply mean the vectors we choose to represent our vector space, whether they are linearly independent or linearly dependent.

When the basis of a vector space is linearly independent, every point in that space has a unique definition. Only a single set of coefficients for the basis vectors will result in that point. We can see this with our 2D Euclidean space example. We know that, given $\mathbf{i}$ = (1, 0) and $\mathbf{j}$ = (0, 1), a point at (4, 3) can only be represented by the position vector $4 * \mathbf{i} + 3 * \mathbf{j}$.

However, let us introduce a third basis vector, $\mathbf{v}$ = (0.4, 0.3). The set of basis vectors $\{\mathbf{i}, \mathbf{j}, \mathbf{v}\}$ are linearly dependent. $\mathbf{v}$ can be written as a linear combination of $\mathbf{i}$ and $\mathbf{j}$:

$$\mathbf{v} = 0.4 * \mathbf{i} + 0.3 * \mathbf{j}$$

If we use our new set of basis vectors $\{\mathbf{i}, \mathbf{j}, \mathbf{v}\}$ to represent a point at (4, 3), we no longer have a unique definition of that point. We could get the point (4, 3) using any of the following coefficients, for example, and they're just a few of the possible combinations:

$$\begin{aligned}
\begin{bmatrix} 4 \\ 3 \end{bmatrix} &= 4.00 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + 3.00 \begin{bmatrix} 0 \\ 1 \end{bmatrix} + 0.00 \begin{bmatrix} 0.4 \\ 0.3 \end{bmatrix} \\
&= 0.00 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + 0.00 \begin{bmatrix} 0 \\ 1 \end{bmatrix} + 10.0 \begin{bmatrix} 0.4 \\ 0.3 \end{bmatrix} \\
&= 2.00 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + 1.50 \begin{bmatrix} 0 \\ 1 \end{bmatrix} + 5.00 \begin{bmatrix} 0.4 \\ 0.3 \end{bmatrix}
\end{aligned}$$

(see Figure 3.6)

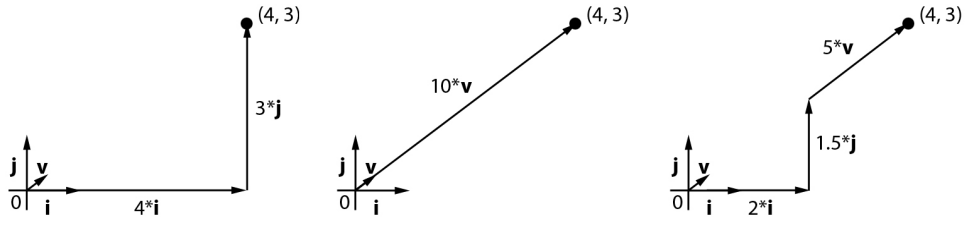


Figure 3.6: Diagram illustrating a few possible combinations of a linearly dependent basis that result in the same 2D point.

If the set of basis vectors are linearly dependent, every point in the vector space has multiple definitions. Applying this understanding to our expression space, where every point in that space represents a different facial expression, if the basis of that space is linearly dependent, many different combinations of coefficients could result in the same facial expression.

Earlier we said those coefficients can be thought of as our blendshape weights. If that is the case, the same facial expression in our expression space could be represented by many different combinations of blendshape weights.

This results in unpredictable behaviour when transforming a facial expression into our expression space, and back out into blendshape weights. Given the same input expression, we could get a variety of different output results. Even if visually the animated face represents the same expression as the actor, the underlying weights could be shifting dramatically between blendshapes every frame, making the data extremely hard to modify after the motion capture process.

As will later become clear (Section 3.6.2), there is actually a separation between the basis vectors of our expression space, and the blendshapes of the animated model (though up until now it is perfectly fine to consider them one and the same). Therefore, it is highly unlikely that every possible combination of weights, even if they mathematically generate the same point in the expression space, will actually generate the same visual result in the animated face. Because of this, drastically shifting weights could result not only in unclean data, but also in visual discontinuities in the facial animation (see Section 5.3).

Thus, we need to define a basis for our expression space that is linearly independent. In other words, our choice of blendshapes should be such that no blendshape can be written as the linear combination of the other blendshapes.

Facial Action Coding System (FACS)

As previously discussed, for facial animation, regardless of whether motion capture is used or not, having a set of blendshapes that can combine to form any facial expression is important.

A lot of research has gone into facial rigging over the years, and there is a strong consensus as to the set of primitive blendshapes required to be able to achieve a full range of facial expressions (Ritchie et al. 2005). The vast majority of facial rigs used in production utilise a set of blendshapes based on the Facial Action Coding System (Ekman et al. 2002), whether those rigs will be animated by hand (Ritchie et al. 2005), or will be driven by motion capture (Sagar 2006 and Havaldar 2006).

> "Ekman and Friesen's Facial Action Coding System (FACS)... groups facial muscles which work as units. It represents an effective alphabet of facial expression; any expression can be formed by combinations of the individual elements." Sagar (2006, p. 1).

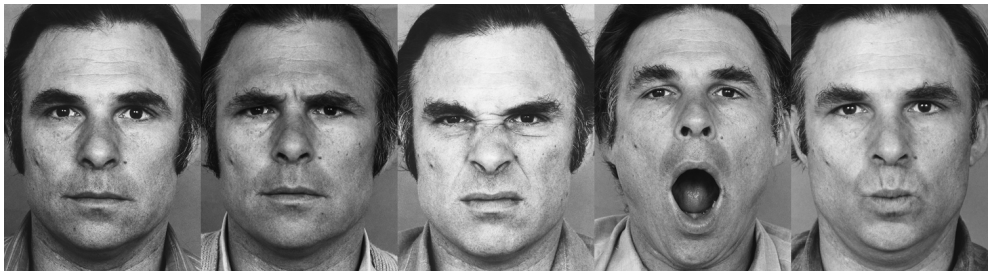


Figure 3.7: Examples of FACS Action Units. From left to right: neutral pose, "Brow Lowerer", "Nose Wrinkler", "Mouth Stretch", and "Lip Pucker". (Terminology and images from Ekman et al. 2002)

FACS describes a number of "Action Units" (Ekman et al. 2002) which are a "way of breaking down the face into workable portions. Dr. Ekman's facial action units describe the precise movement of each individually recognizable muscle group in the human face." (Ritchie et al. 2005, p. 101).

Examples of action units include "Action Unit 4 - Brow Lowerer... [which] lowers the eyebrow[,]... pulls the eyebrows closer together... [and] produces vertical wrinkles between the eyebrows" (Ekman et al. 2002, p. 17), "Action Unit 9 - Nose Wrinkler... [which] pulls the skin along the sides of the nose upwards towards the root of the nose causing wrinkles to appear along the sides of the nose and across the root of the nose" (ibid., p. 93), "Action Units 25, 26, 27 - Lips Part, Jaw Drop, Mouth Stretch" (ibid., p. 103) which describe stages involved in opening the mouth, and "Action Unit 18 - Lip Pucker... [which]shapes the lips as if the person is pronouncing the vowel oo in the word 'fool'." (ibid., p. 233) (see Figure 3.7).

Any facial expression can be created by combining varying intensities of the different action units (Ekman et al. 2002, Ritchie et al. 2005, Sagar 2006, and Havaldar 2006). By basing our set of blendshapes on FACS, we can represent any facial expression by selecting the correct blendshape weights. Thus, if these blendshapes form the basis vectors of our expression space, we can say that, approximately, the basis fully represents our expression space.

FACS divides the muscle motions into groups of independent action, which can act independently with different degrees of intensity to form a given facial expression (Ekman et al. 2002). Because the underlying muscle motions are independent, to get a specific facial pose requires a specific set of muscles to act with specific intensities.

A large part of the FACS training manual (Ekman et al. 2002) discusses how one can identify which action units are present in an expression, based on the visual characteristics of the face alone (i.e. without having knowledge of what the underlying muscles are doing). We therefore make the assumption that the resulting visual movement of the facial features directly corresponds to the underlying muscle movements, and is therefore approximately independent for each action unit (please note: this assumption might not be valid, however, due to the use of a sparse set of markers, as is later discussed in Section 5.3).

Therefore, if the set of FACS action units form our blendshapes (and thus form our basis vectors), and we know each of these motions to be independent, we can say that our basis vectors are approximately linearly independent.

Thus, we use a set of blendshapes based on FACS for our motion capture system, in an attempt to define a basis for our expression space that is both fully representative, and linearly independent.

In addition to forming the basis for our expression space, FACS is incredibly useful for understanding where the different facial muscles are, and in which direction they act (see Figure 3.8).

This was used as a strong basis for determining the required location of our facial markers,

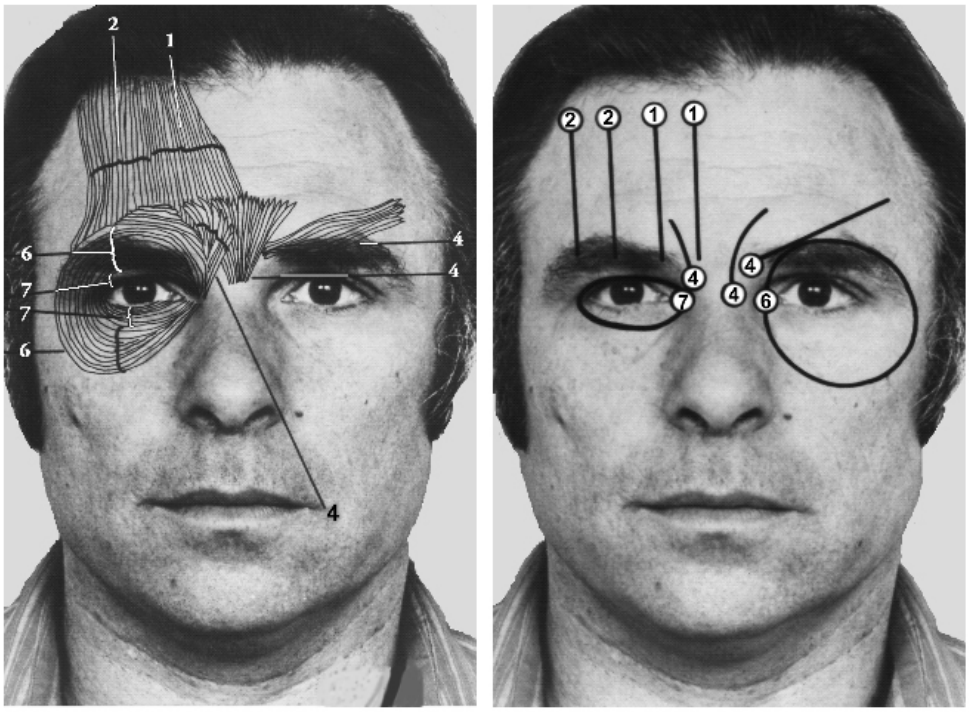


Figure 3.8: Example of a FACS muscle diagram, illustrating the locations and actions of the underlying facial muscles responsible for the FACS Action Units (Image from Ekman et al. 2002)

as the markers needed to be placed on areas of the face that were clearly affected by the underlying muscles.

Video reference of the marker locations used on *Avatar* (2009) (Discovery 2009, Fox 2010, and Mob Scene 2009) was also used, in addition to the FACS muscle diagrams, as inspiration for the placement of our markers.

3.6.2 Solving for Blendshape Weights

By using our blendshape weights directly as the coordinates of the expression space, there is a simple mapping from expression space to our facial rig. However, we still need to transform our marker data into that expression space to determine those blendshape weights (i.e. to determine the expression space coordinates).

To do this, we need to analyse our current facial expression, and determine the intensities with which each FACS action unit is acting. These intensities will give us the coefficients of each of those action units, which we know to be equivalent to the expression space coordinates of our current facial expression.

We earlier showed that the blendshape equation can be written as a matrix equation (Equation 3.8), to recap:

$$\mathbf{O} = \mathbf{B} * \mathbf{w}$$

where $\mathbf{w}$ is a n-vector of weight parameters, $\mathbf{B}$ is a m (rows) by n (columns) matrix, where each column of $\mathbf{B}$ represents one of the blendshape vectors, and $\mathbf{O}$ is the resultant m-vector offset to be applied to the rest pose.

Now, let us speculate that we have been provided with the current animated model for the current facial expression, **M** (even though we obviously don't have this, as it's what we're eventually trying to generate).

Given the rest pose **R**, along with **M**, we can calculate **O** by rearranging equation 3.6:

$$\mathbf{O} = \mathbf{M} - \mathbf{R}$$

Thus, if we have the current animated model for the current facial expression, allowing us to calculate **O**, we can solve the blendshape matrix equation (Equation 3.8) to find our blendshape weights:

$$\mathbf{w} = \mathbf{B}^{-1} * \mathbf{O} \tag{3.9}$$

There are two problems, however.

Problem 1: We don't have the animated model

As earlier noted, we obviously don't have the current animated model for the current facial expression, **M**, because that's what we're trying to generate from our actor's facial expression. Without **M**, we cannot calculate **O**, and thus cannot solve for the blendshape weights, **w**.

What we have instead are the positions of each of our tracked markers for the current facial expression. This information could be written as a vector in a similar format to **M**. The only difference being that the dimensionality of each of our vertices is 2, not 3, because the markers are tracked in 2D, and the number of vertices is substantially fewer than our animated model.

Thus, instead of defining our basis vectors (which get stored in matrix **B** of Equation 3.8) in terms of the blendshapes of the animated model, we define them in terms of the marker locations on the actor's face. We generate an equivalent set of 'blendshapes' using the marker locations, as opposed to vertex locations of the animated model, for every blendshape in the animated model's facial rig.

This is achieved through a configuration stage, where the actor trains the system on each of these 'blendshapes'. Firstly, the actor poses their face in a rest pose, and the location of all the markers are stored. This gives us the rest pose vector, **R**. Then, for each blendshape belonging to the animated model's facial rig, the actor is required to pose their face in the same expression. Again, each of these poses, or 'blendshapes', are captured, and the marker locations processed just as the vertices of a target pose mesh are processed to generate a blendshape - namely, the rest pose is subtracted from each target pose to generate a blendshape offset vector. All of these blendshape vectors together give us the blendshape matrix, **B**.

On any given frame, we have the current location of the markers, **M**. As earlier discussed, with **M**, **R** and **B** we can solve for **w**, giving us the blendshape weights.

Our earlier described expression space is therefore defined in terms of the actor's facial markers, not the animated model. However, because there is a direct one-to-one correspondence between the blendshapes in the facial rig and the 'blendshapes' in our blendshape matrix trained on the actor's face, we can take the weights vector **w**, as solved for the actor's face, and apply that directly to the blendshape rig. It is this division, between the blendshape matrix trained on the actor's face and the blendshape matrix for the animated model, that allows any facial model to be animated with our system, regardless of the resemblance to the actor's face (which was one of our goals for this project (Chapter 1)). In addition, the direct correspondence means that all our earlier discussions regarding vector space theory, and how we define the expression space (i.e. the importance of FACS), still apply to our new expression space defined in terms of marker locations.

Our approach was inspired by Havaldar (2006), who describes Sony Pictures Imageworks' facial motion capture pipeline on *Monster House* (2006):

> "On Monster House, the facial system made use of FACS as a foundation to base the capture and retarget [of] motion captured data on the characters faces. Prior to acting, each actor went through a calibration phase where the actor is made to perform all the action units." (Havaldar 2006, p. 5)

> "we capture mocap frames of the actor that relate to his facial expression corresponding to a FACS pose. Some of these poses are broken into left and right sides in order to capture the asymmetry that an actor face might undergo. Every incoming frame of the actor is then analyzed in the space of all these FACS captured frames. As such, these action unit-triggered poses correspond to facial basis vectors, and each ones activation needs to be computed for an incoming data frame... The activations get transferred onto a digital face, which has been rigged using a facial muscle system." (ibid., p. 10)

This "calibration phase" (ibid., p. 5) can be seen in Figure 3.9.

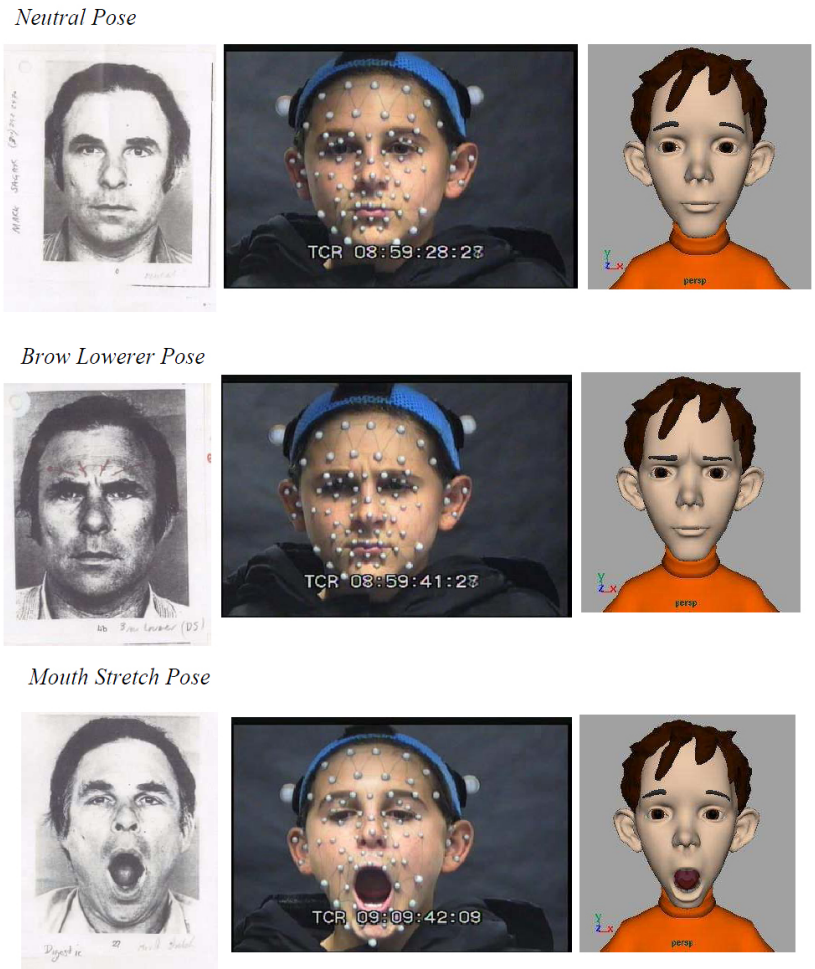


Figure 3.9: Images from Havaldar (2006) illustrating the "calibration phase" (ibid., p. 5) for the facial motion capture pipeline on *Monster House* (2006).

Whilst Havaldar's descriptions are brief, conceptual, and do not detail the specifics of the implementation, we certainly believe our approach to be similar to Havaldar's, and thus we hope to achieve comparable results to Havaldar. However, whereas Havaldar describes the use of a muscle-based facial rig, we use a blendshape-based facial rig, as discussed in Section 3.5. We believe that since our blendshape-based facial rig is based on FACS, which is itself based on the underlying muscles (Ekman et al. 2002), this difference in approach to facial rigging should not greatly impact the effectiveness of the motion capture solution.

Problem 2: Equation 3.9 has no direct solution

Given the above solution to problem 1, we still cannot solve Equation 3.9, because there is no direct solution to that equation. Matrix $\mathbf{B}$ is not necessarily square, and even if it was square, it might not be possible to find a true inverse (or even a pseudo inverse) due to the data often being sparse, and containing a large number of zero values (Comninos 2006).

Instead, we must use an iterative algorithm to estimate the values for the weights vector $\mathbf{w}$ based on $\mathbf{O}$, the current marker locations expressed as an offset from the rest pose.

Choe et al. (2001), Chuang et al. (2002), and Joshi et al. (2006) consider the problem in a linear least squares sense, such that we estimate $\mathbf{w}$ by minimising:

$$||\mathbf{O} - \mathbf{B} * \mathbf{w}_e|| \tag{3.10}$$

where $\mathbf{w}_e$ is our estimated weights vector.

In other words, we estimate the weights vector such that the vector offset of the markers from the rest pose, generated by $\mathbf{B} * \mathbf{w}_e$, is as close as possible (given the number of iterations of the algorithm) to the observed offset of the markers from the rest pose, $\mathbf{O}$.

There are many different algorithms for solving Equation 3.10, as referenced by Choe et al. (2001), Chuang et al. (2002) and Joshi et al. (2006). We decided to use Lawson and Hanson's well known "Non-negative Least Squares" (NNLS) algorithm (Lawson and Hanson 1995) for a number of reasons.

Firstly, this algorithm was used to solve a similar problem by Chuang et al. (2002), who appeared to get pleasing results with their facial motion capture system. In addition, Chuang et al. describe the problems encountered if negative weights are allowed. With negative weights, large positive weight values can often be generated for some blendshapes, and negative weights generated for other blendshapes to cancel those values out. This would result in a mathematically correct minimisation, but when applied to the animated model would result in a lot of distortion (i.e. an incorrect mapping from the actor to the animated model). Chuang et al. use NNLS because it is constrained to only allow positive weights. Thus, no negative weights can be generated to cancel out large positive weights, and the overall solution is much more stable, resulting in less distortion, and thus a better mapping (ibid.).

Disallowing negative weights also makes sense from an artistic perspective - when animating a blendshape based facial rig by hand (i.e. when not using motion capture), one would not expect to ever use negative weights, but rather would only use positive weights to pose the face in some expression other than the rest pose (Ritchie et al. 2005). Thus, if positive values are used exclusively when animating the blendshape weights manually, the same should be true for our motion capture solution.

Finally, unlike most other algorithms to solve Equation 3.10, the source code for a C implementation of NNLS is readily available (Turku PET Centre 2010). Given the scope of the project, it made sense for us to utilise existing source code where possible, as there was not time to implement our own algorithm to solve Equation 3.10.

The C implementation of NNLS (Turku PET Centre 2010) takes in a matrix ($\mathbf{B}$), an observation vector ($\mathbf{O}$) and calculates the required parameter vector ($\mathbf{w}$) to minimise Equation 3.10. It therefore lends itself well to our problem.

Data Output

Once calculated, the weights vector is output from the *Motion Capture Data Mapping* module to the *Blendshapes* module (Section 3.5) for deforming the 3D animated head. In addition, when in 'capture' mode, the weights vector for every frame is streamed out to a file on disk, such that the animation data, as driven by our motion capture system, is stored for future use in other applications (for example, to drive a final, rendered, animated face).

Chapter 4

Implementation

4.1 Hardware

4.1.1 Helmet Mounted Camera

As discussed in Section 3.1, our facial motion capture system utilises a single, helmet-mounted camera as the input device. One of our goals for this project was to develop such hardware on a tight budget (Chapter 1). With this in mind, our helmet mounted camera setup was constructed from the following components:

Components: Video Camera Device

The core component of our helmet mounted camera setup is the video camera device. The specifications of this device have a large impact on the rest of the facial motion capture pipeline, as they define the fidelity of the input data to the system. Both camera resolution and framerate affect the quality of the input data.

The video camera device has to be capable of high enough framerates to capture facial motion at a rate acceptable for its intended use. Given that film runs at 24 frames per second, PAL video at 25 frames per second, and NTSC video at 29.97 frames per second (Wikipedia 2010b), it was decided that a minimum acceptable frame rate was 30 frames per second. Whilst infrared based systems can often capture at higher framerates (Havaldar 2006), our research into affordable camera technology determined that 30 frames per second was the maximum frame rate achievable when capturing full colour images. Thus, we decided to use a video camera device capable of 30 frames per second.

The other factor that defines the fidelity of the input data to the facial motion capture system is the video resolution. Given an identical framing of the actor's face within the image, a high resolution image is capable of representing smaller motions in the facial markers than a lower resolution image. With this in mind, and given that *Avatar*'s (2009) headrigs used HD cameras (Teo 2010), we thoroughly researched affordable camera technology capable of high resolutions at frame rates of 30 frames per second.

Based on this research, we determined that the best video camera device for our requirements was the *Microsoft H5D-00003 LifeCam Cinema Webcam* (see Figure 4.1). Purchased for £44.99, the *Microsoft H5D-00003 LifeCam Cinema Webcam* supports resolutions of 1280x720 at 30 frames per second, and thus is ideal for our purposes.

The webcam nature of the *Microsoft H5D-00003 LifeCam Cinema Webcam* met the additional requirement, as specified in Section 3.1, of being able to output a continual live video stream to a desktop PC. In addition, the *Microsoft H5D-00003 LifeCam Cinema Webcam* is a USB device, and thus facilitates easy access to the video data through third party media APIs (Section 4.2).

It is briefly worth noting that the *Microsoft H5D-00003 LifeCam Cinema Webcam* allows for both automatic and manual calibration of white balance, exposure, and focus. Using

the included software, we disabled automatic calibration, and manually selected the desired settings. We found that using automatic calibration could cause inconsistent frame rates whenever the hardware recalibrated itself. Thankfully, the manual settings only need to be specified once for each work session, as the settings are remembered after the included software is closed, and our software launched.

Figure 4.1: The *Microsoft H5D-00003 LifeCam Cinema Webcam* - the video camera device of our helmet mounted camera setup.

Components: Helmet

The second most important component of our helmet mounted camera setup is the helmet. The helmet must be able to rigidly attach the video camera device to the actor's head. Therefore it must both securely grip the actor's head, and facilitate the attachment of the camera to the helmet structure.

As discussed in Section 2.2.3, *Avatar* (2009) manufactured scull caps for each actor based on a life-cast and a laser scan of the actor's head (Discovery 2009). Such custom scull caps are beyond the scope of this project. Thus, we decided to re-factor an existing helmet for our helmet mounted camera setup.

Bicycle crash helmets are designed to securely fit the rider's head, and the large variety of crash helmet designs allowed us to find a helmet which caused minimal obstruction to the actor's face, both in terms of visually occluding the actor's face from the camera, and preventing the actor from moving their face freely.

Thus, for our helmet mounted camera setup we use the *freestyle helmet* model of crash helmet, manufactured by *Hood* (see Figure 4.2). We found this helmet to securely fit the actor's head, with minimal obstruction to the actor's face, and to be manufactured from a material which allowed for easy attachment of the camera to the helmet structure. The helmet was purchased for £16.99.

Figure 4.2: The *freestyle helmet* by *Hood* - the helmet of our helmet mounted camera setup.

Components: Additional Components

In addition to the video camera device and the helmet, the following components were required for the helmet mounted camera setup's construction: 1m length of 10mm x 2mm aluminium bar (purchased for £3.66, and chosen for its strength, flexibility, and light-weight material), 60g (four 10g, and four 5g) lead weights to counter-balance the weight of the camera (purchased for £0.20), four 10mm length screws and three pot rivets (purchased for approximately £0.15).

Components: Total Cost

Thus, the total hardware cost for the helmet mounted camera setup was:

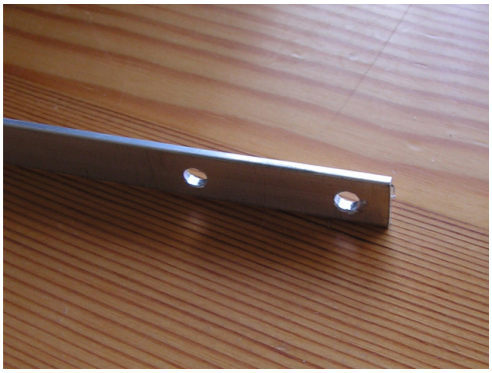
- *Microsoft H5D-00003 LifeCam Cinema Webcam* - £44.99
- *freestyle helmet* by *Hood* - £16.99
- 1m length of 10mm x 2mm aluminium bar - £3.66
- 60g (4 x 10g and 4 x 5g) lead weights - £0.20
- Four 10mm length screws and three pot rivets - £0.15

- **Total:** £65.99

Manufacturing Process

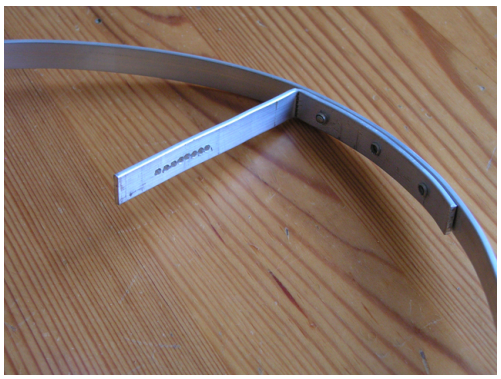
Assisted by David Owen (a mechanical engineer and family friend), the helmet mounted camera setup was manufactured in the following manner:

Firstly, two small holes, about an inch apart, were drilled into the aluminium bar at each end. These holes are later used to screw the aluminium bar to the helmet.



The aluminium bar was then bent into a large U-shape, such that it would wrap around from one side of the helmet to the other. Two small pilot holes were drilled into the helmet, and the aluminium bar screwed to the helmet using the inner most set of holes. The outer hole is currently left unscrewed to enable rotational freedom in the bar, allowing for flexibility in determining the camera angle during construction.


A small section of aluminium bar is bent into a right angle, and attached to the main aluminium bar with pot rivets. It is positioned such that the camera is centred on the actor's face (as will later become apparent when the camera is attached). A series of small holes are drilled into the bar to facilitate flexibility in determining the camera position during construction.



The included camera stand is removed from the *Microsoft H5D-00003 LifeCam Cinema Webcam*. This reveals a screw, previously used to attach the stand to the camera.

This screw is used to attach the camera, via one of the previously described small holes, to the aluminium bar. The hole is chosen to give the best framing of the actor within the video feed.

The angle of the camera, and the angle of the main aluminium bar are adjusted to give the best framing of the actor within the video feed. With the camera position and angle finalised, the screw holding the camera in place is firmly tightened. The main aluminium bar is locked in place by a second set of screws. The camera is now rigidly attached to the helmet.

Finally, a series of lead weights are attached to the rear of the helmet, to counter-weight the aluminium bar and camera. This balances the helmet on the actor's head for maximum comfort.

4.1.2 Other Hardware

In addition to the helmet mounted camera setup, the following hardware is utilised by our facial motion capture system:

Workstation Specifications

A desktop PC is used both to develop and execute the facial motion capture software. It has the following specifications:

- *Intel Core i5* Processor (quad-core, 2.66GHz clock speed)
- 8GB (4 x 2GB) *G.SKILL* DDR3 1333 RAM
- *Point of View* (*NVIDIA* chipset) GTX260 graphics card (216 Core EXO Edition)
- *Gigabyte* GA-P55A-UD3 Motherboard
- *Corsair* 750W HX Modular PSU
- *Microsoft Windows 7* 64-bit operating system.

We believe this to be a moderate performance workstation, not uncommon in the 3D industry. Perhaps the most important of these specifications is the graphics card. Our facial motion capture system utilises the GPU, where possible, to gain from it's parallel computing performance. Because of this, a powerful graphics card is required to execute our facial motion capture software in real-time.

Other Materials

Green face paint and blue liquid eyeliner are used to paint markers onto the actor's face. These materials were chosen primarily because they are dermatologically tested, and thus are safe to use on the skin of the actor's face. Secondly, they provided the strong saturation and matte colour required by the tracking algorithm (Twist 2010). In addition, they are both readily available and affordable (each were purchased for around £3).

A simple lighting setup was also purchased to aid in the tracking process, as detailed in our earlier report (Twist 2010). This lighting setup cost approximately £30.00.

4.2 Webcam Image Stream

The *Microsoft H5D-00003 LifeCam Cinema Webcam* (Section 4.1.1) provides a native video stream in YUY2 format. As outlined in Section 3.2, the *Webcam Image Stream* module is responsible for converting this YUY2 format to RGB bitmap format. In addition, it controls the overall logic loop of the application.

The data stream from the webcam is accessed using Microsoft's *Media Foundation API* (Microsoft 2010a). The Media Foundation API allows for asynchronous data transfer from a webcam device (Microsoft 2010b). Using the Media Foundation API, a separate thread is created, responsible for gathering a frame of video data from the webcam and executing a callback method once that data is ready for use within the application.

Within this callback method the next frame of data is requested (from the Media Foundation API), setting up an infinite loop, in which our callback method is called once per frame, as soon as a frame of video data is ready for use within the application. We use this callback method to track the status of the current frame of video data.

Within the main application, the availability of the current frame of video data is queried. All execution (except the processing of user input events) is suspended until that frame is ready for consumption. As soon as it becomes available, that frame of data is processed by

the entire facial motion capture pipeline (Chapter 3), and execution is again suspended until the next frame of data is ready.

In this manner, our main thread of execution has, in theory, the entire duration of a single frame to process the facial motion capture pipeline, rather than having to spend some of that time transferring the video data over USB. We found this to improve the overall performance of our software.

Conversion from YUY2 to RGB can be achieved with a very simple algorithm (Microsoft 2004). Every 32-bit piece of YUY2 data becomes two 32-bit pieces of RGB data, and can be processed independently (Wilson 2007). Thus, the conversion from YUY2 to RGB is a highly parallel task.

We implement this conversion algorithm on the GPU to exploit its highly parallel nature. The YUY2 data is transferred to the GPU, where it is then converted to RGB and stored as a RGB bitmap in GPU memory. Because subsequent processing on the RGB bitmap (both drawing, and tracking (Section 4.3)) occur on the GPU, we minimise the data transfer from the host memory to GPU memory (transferring only the initial YUY2 data), and thus avoiding unnecessary bottlenecks in our pipeline (NVIDIA 2009a and NVIDIA 2009b).

4.3 Tracking

As previously mentioned (Section 3.3), the *Tracking* module was implemented for a previous project, and thus its design and implementation are covered in detail in that project's report (Twist 2010). That report has been reprinted, for convenience, in Appendix A of this thesis.

4.4 Tracking Smoothing

As discussed in Section 3.4, the *Tracking Smoothing* module is responsible for filtering out high-frequency, low-amplitude noise from the motion signal of the markers.

We implement this low-pass filter using a simple, efficient moving average algorithm (Wikipedia 2010d). The 'smoothed' marker data for any given frame is calculated as the mean of the observed marker data over the previous n frames, where n is a configuration parameter of our system.

Given that the marker data is a single column vector (see Sections 3.5 and 3.6), we construct a matrix with n columns to store this data over n frames.

This matrix acts as a circular buffer (Wikipedia 2010a). Every frame, the index into that buffer is incremented. When the index reaches the end of the buffer it is reset to the beginning of the buffer. The marker data for the current frame is stored into the column of the matrix referenced by this index, thus overwriting the oldest observation in the buffer. The smoothed marker data for the current frame is calculated as the mean of the columns in the matrix.

Obviously this would cause problems for the first $n - 1$ frames, when the matrix hasn't been fully initialised. For these frames, only the first m columns of the matrix (where m represents the number of frames currently stored in the matrix) are averaged.

It is worth noting that whilst the moving average algorithm is highly parallel, and thus well suited to being implemented on the GPU, it was decided to implement the algorithm on the CPU. Given the ratio of the data transfer (the current observation would need to be transferred to the GPU, and the smoothed data would need to be transferred back from the GPU) to the number of operations on that data, and the fact that data transfer to the GPU is the main bottleneck in GPU execution (NVIDIA 2009a and NVIDIA 2009b), we feel that no performance would be gained by implementing the algorithm on the GPU. In other words, the time saved in executing the algorithm on the GPU, would be lost in transferring the data to and from the GPU.

4.5 Motion Capture Data Mapping

Section 3.6 discussed at length our approach to the motion capture data mapping process. In short, the actor is required to train the system on a number of 'blendshapes' (each of which correspond to a blendshape from the facial rig), which define the basis vectors for an "expression space" (Sagar 2006). Each blendshape is stored as a column in a matrix. Given this matrix, and the current position of the markers, we can solve for the blendshape weights.

We use the Non-negative Least Squares (NNLS) algorithm (Lawson and Hanson 1995), as implemented by the Turku PET Centre (2010), to solve this problem. Whilst using third party code gains us the advantage of not having to dedicate time to implementing and testing our own solution (which was beyond the scope of the project), it was not without its issues.

The NNLS implementation by Turku PET Centre takes in the above mentioned basis vector matrix and current observation vector, and outputs the blendshape weights. However, the basis vector matrix is modified as part of the algorithm. If this matrix is then used as the basis vector matrix for subsequent frames, the results will be incorrect.

To overcome this, a temporary copy of the matrix is created, and passed to the NNLS algorithm. The data from the true basis vector matrix is copied into this temporary matrix prior to executing the NNLS algorithm on each frame. The memory for this matrix is allocated once, at program start-up, in order to avoid the overhead of run-time memory allocation, and thus to achieve increased performance.

Whilst the NNLS algorithm is considered efficient (Lawson and Hanson 1995), the larger the input matrix and observation vector, the longer the algorithm takes to solve for the blendshape weights (ibid.). Due to this, it was decided to break the face into different regions, where possible, with each region being processed independently. Because each region would contain a subset of the full marker set, it would have both a smaller basis vector matrix and a smaller observation vector, and thus would reduce the computational cost of the NNLS algorithm.

We utilise two such facial regions (although the system supports any configuration of facial regions): the "upper face" (the eyebrows), and the "lower face" (everything except the eyebrows), which can be analysed and solved independently (Ekman et al. 2002).

During the configuration stage, the actor must train the system on each of the basis vectors ('blendshapes') that define the expression space (see Figure 4.3). A number of techniques were utilised to ease this process.

With each basis vector being based on a FACS action unit (Section 3.6), we felt that many actors could struggle to individually pose some of the action units. "Inner Brow Raiser" (ibid., p. 20) and "Outer Brow Raiser" (ibid., p. 22), for example, require individual control of the eyebrow muscles, which many people find hard to achieve (ibid.). Rather than forcing the actor to spend countless hours practicing and perfecting individual control of the facial muscles (as recommended by Ekman (ibid.)), our approach allows the actor to train the system on both the "Inner Brow Raiser" and "Outer Brow Raiser" at the same time. In addition, our approach allows the actor to train the system on both the left and right eyebrows, at the same time.

We achieve this through the use of 'Blendshape Groups', specified in configuration files. Rather than training the system on individual blendshapes, the actor trains the system on a number of Blendshape Groups, which consist of one or more blendshapes, and are associated with a facial region (as mentioned above).

When the actor poses for a certain Blendshape Group, the system is instructed to store the current marker locations as that pose. The rest pose vector is subtracted from the current locations to generate a basis vector (Section 3.6). This basis vector is not used for every blendshape belonging to the Blendshape Group, however. Instead, the configuration files, on a per-blendshape basis, specify multipliers for each of the markers. These multipliers can be thought of as weights; i.e. how much influence does that blendshape have over each marker. The basis vector is multiplied by these weights before being stored in the basis vector matrix.

Figure 4.3: A selection of poses from the facial motion capture configuration process.

In this way, a number of blendshapes can be configured with a single pose, by specifying the markers influenced by each blendshape. As a practical example, the "Inner Brow Raiser" for the left eyebrow would have weights of 1.0 for the markers of the inner left eyebrow and 0.0 for other markers in the "upper face" region. The "Outer Brow Raiser" for the left eyebrow would have weights of 1.0 for only the markers of the outer left eyebrow. Similarly for the inner and outer right eyebrow. The result of multiplying these weights with the marker locations would be the same, roughly speaking, as if the actor had been able to perform each of these blendshapes individually.

The weights can take any value, allowing a marker to share weighting between multiple blendshapes (as is often the case with markers in the centre of the face for asymmetrical actions).

In addition to the use of Blendshape Groups, a simple but effective visualisation of the current pose the actor is configuring also eases the configuration process. When in configuration mode, rather than evaluating the NNLS algorithm to generate the weights output from the *Motion Capture Data Mapping* module, the weights are generated based on the blendshapes being configured. All blendshapes not being configured are assigned a weight of 0.0. All blendshapes being configured are assigned an animated weight, which repeats the following sequence: increase from 0.0 to 1.0, hold briefly at 1.0, and then decrease back to 0.0.

This results in the animated model moving from a neutral pose, to the pose the actor is configuring, holding that pose at the extreme point, and then back to the neutral pose. We feel that this visual information is invaluable to the actor's understanding of what facial expression they should be performing. In addition, we feel by animating the pose from neutral to extreme, subtle poses can be clearly identified (more so than if just the extreme pose was shown, without animation), and the actor acquires an understanding for not just the final desired pose, but the motion their face should go through to get there (which we found often helps with achieving the correct pose).

4.6 Blendshapes

The blendshape algorithm (Section 3.5) is evidently a highly parallel task, allowing for each vertex in the mesh to be processed independently. We implement the blendshape algorithm on the GPU to exploit this highly parallel nature.

The rest pose model, and each of the target pose models, are loaded at program start up. The target pose models are processed to generate each of the blendshape vectors, as discussed in Section 3.5. This processing is performed by the CPU. Once generated, the information remains constant throughout the program execution, as only the blendshape weights will change on a frame to frame basis.

Thus, once generated, the rest pose model, and each of the blendshape vectors, are transferred to GPU memory. This is quite a slow process, as there is a lot of data to transfer (NVIDIA 2009a and NVIDIA 2009b), but it occurs only once, at program start up, and thus does not adversely affect the performance of the program.

The desired end result of the *Blendshapes* module is a deformed head model that can be drawn to the screen. We utilise OpenGL Vertex Buffer Objects (VBOs) (Ahn 2005) such that the model is kept entirely in GPU memory, with that memory modified by a GPU algorithm. This allows us to avoid unnecessary data transfer from the host memory to GPU memory (and vice-versa), and thus avoid unnecessary bottlenecks in our pipeline (NVIDIA 2009a and NVIDIA 2009b).

The GPU blendshape algorithm takes a weights vector as input on every frame, and calculates the deformed model from the rest pose and blendshape vectors, storing the deformed vertex locations and vertex normals into an OpenGL VBO.

For optimal performance, consecutive threads should access consecutive memory elements (NVIDIA 2009a and NVIDIA 2009b). As discussed in Section 3.5, our rest pose mesh, and all blendshape vectors, are stored such that a vertex's x, y and z components are consecutive elements in a vector (as opposed to each vertex being a single element in a vector).

For this reason, each vertex component (i.e. x, y **or** z) are treated as individual work-items, as opposed to a whole vertex (i.e. x, y **and** z). Each work-item is assigned a thread, responsible for processing the blendshape algorithm on that component. This equates to 1 thread per element of the vectors in Equations 3.6 and 3.7, and thus ensures that each consecutive thread processes consecutive memory elements, achieving optimal performance.

Chapter 5

Results and Analysis

5.1 Temporal Performance

A key goal of this project was to develop a facial motion capture system capable of animating a facial rig, in real-time, based on a live video stream of an actor's face (Chapter 1). Having chosen the video hardware for this project (Section 4.1.1), 'real-time' was defined as 30 frames per second (30Hz).

Our facial motion capture system is hugely successful in achieving this goal, running at a constant and steady 30Hz.

Using a high resolution CPU timer (Bolton, no date), which based on our CPU clock speed is accurate to 1/2666 of a second, we tested the performance of our facial motion capture system.

The entire motion capture pipeline (Chapter 3), from receiving a frame of YUY2 image data from the hardware device, to generating weights for a facial rig, and using those weights to deform and draw a model on the screen, takes on average 15.54ms per frame. Thus, in theory we could run these calculations at approximately 64Hz, easily meeting our requirement of 30Hz.

To run at 30Hz, all calculations for a single frame must take less than 1/30 of a second (33.333ms). Thus, our current implementation has approximately 17.79ms per frame of free calculation time, which could be dedicated to additional tasks. Examples of such additional tasks are mentioned later in this chapter, and in Section 6.1.

As discussed in our previous report (Twist 2010), the *Webcam Image Stream* and *Tracking* modules take approximately 5.08ms per frame to evaluate. The remaining 10.46ms of the evaluation time is divided up amongst the other modules in the following manner: *Tracking Smoothing* takes less than 0.01ms per frame to evaluate, as the calculations are extremely simple, and thus very efficient; *Motion Capture Data Mapping* (i.e. the evaluation of the NNLS algorithm for each of the facial regions) takes 2.84ms per frame to evaluate, on average; The *Blendshapes* module takes approximately 1.48ms per frame to evaluate; The remaining 6.14ms per frame is used by OpenGL to draw the various UI elements (including our face model) to the screen.

5.2 Cost Performance

Another key goal of this project was to develop the hardware for our facial motion capture system on a tight budget (Chapter 1). We believe that we were successful in achieving this goal.

Our helmet mounted camera setup cost £65.99 to manufacture (Section 4.1.1), and our basic lighting setup cost approximately £30.00. These are the only additional costs on top of a workstation of moderate specifications, which we believe to be common place in the 3D industry (Section 4.1.2). Even if one was to purchase a workstation for the sole purpose of

running our facial motion capture system, the workstation we use cost approximately £600 to purchase (in December 2009). Thus, even the combined cost of the workstation, helmet mounted camera setup, and lighting setup, is far less than a similar, non-real-time system from Meta Motion (2010), which costs just under $10,000, for example.

5.3 Visual Performance

Given that we were successful in achieving the goal of a real-time system, on a tight budget (Sections 5.1 and 5.2), we must now evaluate if we were successful in achieving our overall goal: to animate any CG face based on the performance of an actor (Chapter 1).

Evaluating the quality of facial animation is a very subjective task. We feel that the quality of animation generated by our facial motion capture system has both strengths and weaknesses.

Overall, the animation is very smooth, with few artefacts (i.e. no jittering or popping). We feel the overall essence of the actor's expression is captured well, and transferred to a CG face that does not bear close resemblance to the actor (i.e. has different facial proportions to the actor). Understanding the underlying mathematics and system design (Chapter 3), we know that any model utilising similar sets of blendshapes would result in approximately equivalent animation. Large motions in the actor's face, including eyebrow motions and overall changes in the shape of the mouth, are represented well by the animated face. Such results can be seen in Figure 5.1.

However, the system does have some weaknesses. We feel there is a lack of detail in differentiating subtle changes in expression. This is most noticeable in the lips, which are arguably the most expressive part of the face (Ekman et al. 2002). The lips of the animated model reflect the overall lip shape of the actor, but do not represent the subtle details, shapes, and configurations of the lips that give a performance its believability (see Figure 5.2).

This is very evident when watching the actor speak (and, unfortunately, is hard to see through the still images presented in this thesis - video demonstrations of the discussions made in this chapter are included with the project hand-in for a better illustration). When the actor speaks, the overall jaw motion and overall elliptical shape of the mouth (i.e. wide/narrow, open/closed) are captured well. One can tell that the animated model is being driven by the dialogue of the actor, mostly due to the jaw motion being timed perfectly with the dialogue.

However, the subtle details of the lip-sync are not present, and thus we feel one would not believe the animated model to truly be speaking those lines. We feel such details are essential to convincing facial animation, and thus we feel this weakness is something that must be addressed for this system to be considered 'production ready'.

Our facial motion capture system is based on the underlying concept of an "expression space" (Sagar 2006) (see Section 3.6). We believe that the concept of an expression space, and its applications to facial motion capture, can provide extremely good results, being used to great effect on *King Kong* (2005) (Sagar 2006), *Monster House* (2006) (Havaldar 2006), and *Avatar* (2009) (Teo 2010).

We therefore believe an "expression space" to be a good approach for a facial motion capture system, and thus are led to believe that there are two factors which could be contributing to the weaknesses previously mentioned:

- The definition of our expression space is not ideal.
- The definition of our expression space is ideal, but our transformation of the marker data into that expression space is not ideal.

Let us begin by discussing the second factor.

Figure 5.1: Illustrating a selection of 'good' results from the motion capture system. Broad expressions are reflected quite well in the animated model.

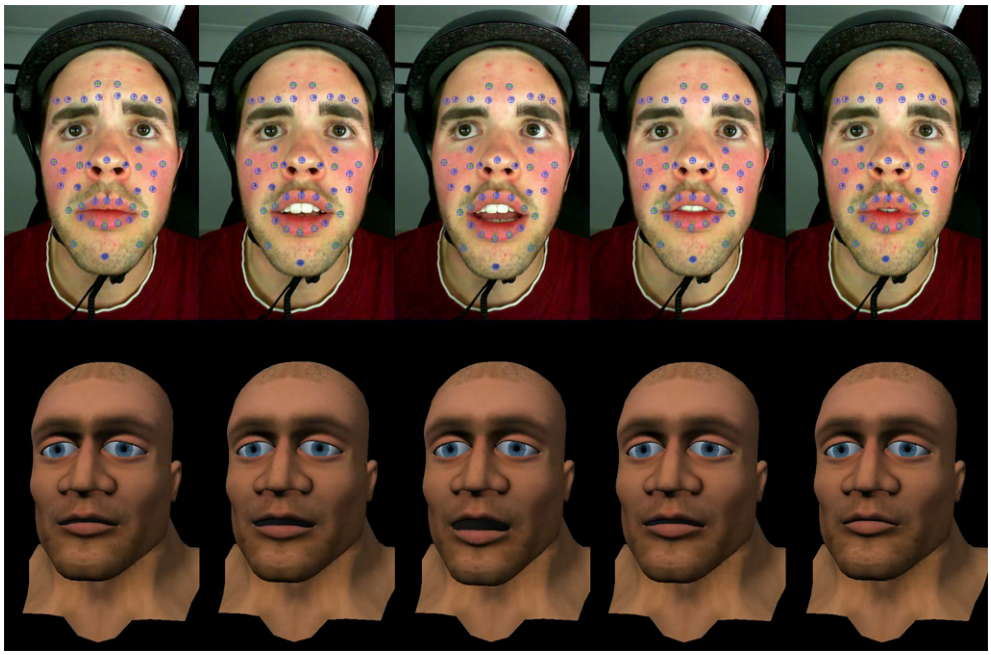


Figure 5.2: Illustrating a selection of 'poor' results from the motion capture system. Subtle changes in the lip motion are not reflected well in the animated model.

Theory: Non-ideal expression space transformation

If we assume our expression space to be ideal, then correctly transforming our marker data into that expression space should result in expression space coordinates (blendshape weights) which fully represent the actor's expression when applied to the animated model.

We know this to not be the case. Some of the details of the facial expression are lost in the mapping process, as discussed above. If our expression space is defined correctly, one would assume that this detail loss is caused by the transformation of the marker data into the expression space: i.e. given an input expression, the correct location in expression space is not being calculated.

We use the NNLS algorithm (Lawson and Hanson 1995) to solve for our expression space coordinates (blendshape weights), given the current marker locations (see Section 3.6). There are alternative algorithms to solve the same problem, a few of which are referenced by Choe et al. (2001) and Joshi et al. (2006). Thus, if the NNLS algorithm is resulting in a poor transformation, it is possible that one of these alternative algorithms might give better results.

We tested the mapping of the marker data into our expression space, using the estimated weights vector and the basis vector matrix to calculate a set of marker locations. We then compared these to the observed marker locations, testing the output data of the mapping process to see if it matched the input data.

We found the calculated marker locations to show quite a strong correlation to the observed marker locations. In other words, given an input expression and a basis vector matrix, the NNLS algorithm is calculating the weights required to best locate our input expression within our expression space. Thus, we believe this to indicate that the NNLS algorithm is performing as intended for minimising Equation 3.10. Therefore, whilst it is possible that alternatives to the NNLS algorithm may produce better results, we believe the weaknesses of our system are caused not by the transformation into our expression space, but instead by the definition of that expression space.

It is important to note, whilst discussing the transformation of marker data into our expression space, that smoothing the marker data (Sections 3.4 and 4.4) greatly improved the results of our facial motion capture system.

Any shake introduced into the marker data's motion signal (either from physical shake in the hardware, or due to the tracking algorithm) was removed by the *Tracking Smoothing* module. As the smoothed data is used by all modules further down the pipeline, we believe smoothing the data resulted in more accurate definitions of the basis vector matrix and per-frame observation vectors, and therefore resulted in a better mapping of the per-frame observation vectors into our expression space. In terms of visual impact, smoothing the data removed animation artefacts from the animated model (jittering and popping animation), and resulted in the animated model better reflecting the overall expression of the actor.

However, it unfortunately resulted in some loss of details. Subtle motions (i.e. high-frequency, low-amplitude motions) get treated as shake, and thus get filtered by the *Tracking Smoothing* module.

The number of samples used by the moving average calculations can be customised (Section 4.4). We experimentally selected the optimum value of 4 samples, after determining that 3 samples or less introduced too many artefacts in the animation (i.e. didn't remove all of the shake from the motion signal of the markers), and 5 samples or more reduced both the detail and responsiveness of the animation.

Theory: Non-ideal expression space definition

Given that we believe the transformation of the marker locations into the expression space to be functioning correctly, the weaknesses previously mentioned are most likely caused by the way in which our expression space is defined.

We earlier discussed the basics of vector spaces, addressing why it is essential for the basis vectors to be linearly independent and fully representative of the vector space (Section 3.6). We then proceeded to introduce FACS (Ekman et al. 2002) in relation to those concepts,

suggesting that FACS can provide us with an ideal set of basis vectors, and thus can define our expression space.

To investigate these claims, in terms of the visual results achieved with our facial motion capture system, we performed two simple tests using alternative blendshape sets to FACS (and thus alternative definitions of the expression space): phonemes and expressions.

Phonemes

Phonemes are atomic components of sound used to construct spoken language (International Phonetic Association 1999). The shape of the face, with focus on the shape of the lips, for each of these atomic components of sound can form a set of blendshapes. Some examples of phoneme based blendshapes can be seen in Figure 5.3.

Unlike FACS action units, which are primitive components of facial expression based on the underlying muscle groups (Ekman et al. 2002), phoneme based blendshapes have no such anatomical foundation. Because of this, many phonemes consist of facial expressions which are only subtly different from one another. We believe this to result in basis vectors which are linearly dependent. In other words, the basis vectors for certain phonemes can be constructed from the linear combination of the other phoneme basis vectors, especially when a sparse marker set is used to define these basis vector (an issue that will be discussed, in depth, later in this chapter).

Thus, a phoneme based set of blendshapes was chosen to test our theory that linear independence in the basis vectors of our expression space is important. The results that we observed illustrate some of the points made in Section 3.6.

Due to the linear dependence of the basis vectors, certain facial expressions have multiple definitions in our expression space (i.e. can be constructed from different combinations of blendshape weights). Small motions in the actor's face can result in dramatic shifts in weights between these linearly dependent blendshapes, which when applied to the animated model result in animation artefacts. For example, smooth motion on the actor's face can become jittering animation on the animated model, caused by the weights shifting back and fourth between a number of blendshapes. This can be partially seen in Figure 5.4, although it is much more clearly illustrated in the video that accompanies this project hand-in.

By using only a subset of the phoneme based blendshapes, which roughly correspond to FACS action units, all previously witnessed artefacts were resolved. This leads us to believe that the artefacts are caused by the blendshapes being too closely related, which we believe to correspond to the linear dependence of those blendshapes.

Expressions

Much like the phoneme example tests the theory of linearly dependent basis vectors, our next example tests the theory of not fully representing the expression space.

We chose a set of blendshapes, where each blendshape represents a full facial expression. These blendshapes can be seen in Figure 5.5. Please note: for this test we used only a single facial region containing the entire set of markers because the expression-based blendshapes do not allow for seperation of facial regions.

Using these blendshapes as our basis vectors, the results were as expected. If the actor posed their face in an expression close to any of the input expressions, the resultant animated face would be heavily influenced by that expression, and thus would appear to match our actor's facial expression (see the left-most two images of Figure 5.6). However, if the actor posed their face in some expression that the system was not trained on, the resultant expression of the animated model was a poor match to the actor (see the right-most two images of Figure 5.6).

Logically, this makes sense. The system is unable to isolate parts of the facial expressions it is trained on. Thus, it is unable to combine parts of one blendshape with parts of another blendshape to generate the desired facial expression. Instead, it has to use some multiple of the full expression.

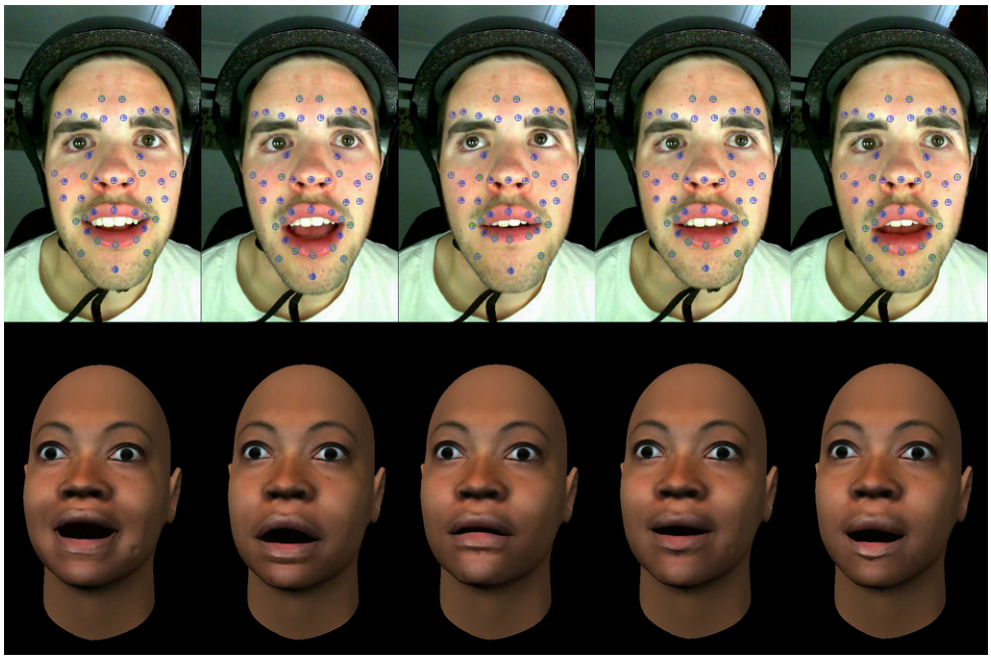


Figure 5.3: Illustrating a selection of the phonemes the system was trained on in configuration mode. From left to right: /E/, /e/, /f/ or /v/, /i/ and /n/.

Figure 5.4: Illustrating the results of using a phoneme set. Small changes in facial movement can result in large changes in blendshape weights due to linearly dependent basis vectors.

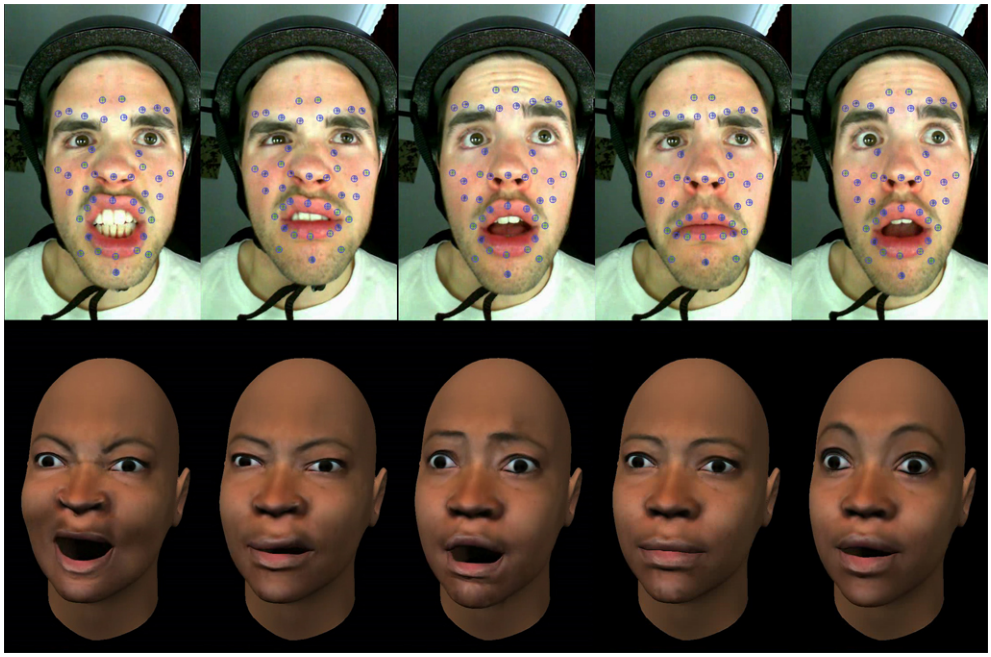


Figure 5.5: Illustrating the expressions the system was trained on in configuration mode. From left to right: anger, disgust, fear, sad and surprise.

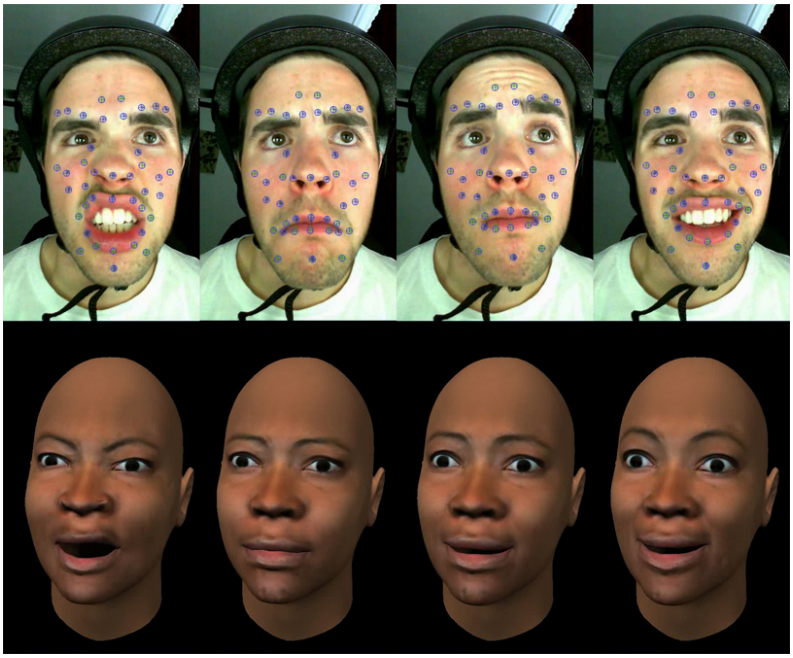


Figure 5.6: Illustrating the results of using an expression set. The left-most two images demonstrate inputting expressions close to the basis vector expressions, resulting in a good representation of those expressions with the animated model. The right-most two images demonstrate inputting expressions the system isn't trained on, resulting in a poor representation of those expressions with the animated model.

The problem, of course, is that each of these expressions do not combine together to form every other possible facial expression - far from it - and thus many facial expressions cannot be achieved by using these blendshapes.

As a result of the minimisation of Equation 3.10, the resultant expression is often a confused mix between two or more of the blendshapes. This is most clearly seen when the actor is talking. To try and find a point in expression space that represents the expression of the actor's mouth, some of the basis vectors with a similar mouth shape will receive weighting. However, those basis vectors also contain changes in the rest of the facial expression - most notably, the eyebrows. When the actor talks, and thus as the actor's lips move, even if the actor's eyebrows are stationary, the animated model's eyebrows are forced to move, by the fact that the eyebrow and lip motion cannot be separated.

Mathematically, when the observed marker locations are transformed into the expression space, they can only be represented with the basis vectors, and thus cannot be accurately represented. Instead, the best fit of the observed marker locations is found, which results in some position in expression space that does not closely represent any one portion of the facial expression. The position in expression space that represents our desired facial expression simply cannot be generated from the basis vectors. In other words, our basis vectors do not fully represent the expression space.

FACS

As hopefully illustrated by our phoneme and expression tests, it is of great importance to the visual results of the facial motion capture system that the basis vectors are both linearly independent and fully representative of the expression space.

Given the strengths of our visual results (as previously discussed), we believe that our FACS-based set of basis vectors do come close to both linear independence and full representation of the expression space. However, mathematically speaking it is unlikely that the blendshapes the system is trained on (and thus the basis vectors of our expression space) are truly linearly independent and fully representative of the expression space. This is likely the cause of the weaknesses in the visual results of our facial motion capture system.

This problem is most likely compounded by the use of a sparse marker set. Earlier (Section 3.6) we made the assumption that the perceived motion of the facial features is independent, because the underlying muscle motions are independent. However, given the sparse nature of the samples taken from the face (i.e. our marker locations), it is highly possible that this assumption is incorrect.

The lack of a third dimension when using a sparse marker set also makes it hard to distinguish certain facial motions. We found, for example, that "Action Unit 18 - Lip Pucker" (Ekman et al. 2002, p. 233) and "Action Unit 22... - Lip Funneler... [where] lips funnel outwards taking on the shape as though the person were saying the word flirt" to be indistinguishable when analysing the sparse marker set of the lips within our 2D video footage. The largest distinguishing factor is that Action Unit 22 pushes the markers forward, where-as Action Unit 18 does not, which is simply not detectible from our video footage.

The use of appearance-based (marker-less) techniques (Pighin 2006) would define a very different expression space, of much higher fidelity. This high fidelity of information perhaps would be independent for each underlying muscle motion, where our sparse marker set is not. A sparse marker set cannot detect patterns of wrinkles that occur with certain muscle actions, for example. An appearance based method could make use of such information to distinguish the muscle motions (ibid.).

Whilst *Avatar* (2009) certainly used markers (as is evident in Figure 2.1), which were tracked in 2D (as can briefly be seen in *Avatar: Creating the World of Pandora* (Fox 2010, 11 minutes, 48 seconds)), very little information is available regarding the underlying technology. Interestingly, James Cameron hints that the technology made use of more than just the facial markers:

> "And that proved to... be the holy grail approach to how to do CG faces. Not the stuff that they've done before, which is what we call marker-based" (Discovery 2009, 1 minute, 58 seconds)

> "The part that we had to create during the making of this film was the facial, image-based performance capture technology" (Mob Scene 2010, 3 minutes, 1 second)

Thus, we cannot say for certain whether or not a sparse 2D marker set was solely used on *Avatar* (2009) to distinguish all of the FACS action units.

Monster House (2006) used solely marker-based techniques, but had a much denser marker set, which was tracked in 3D (Havaldar 2006), perhaps better defining the expression space.

Whilst the low fidelity of the sparse 2D marker set used by our facial motion capture system may not contain the detail required to best define the expression space, it does lend itself well to a real-time system. Increasing the fidelity of the input information, perhaps with a dense marker set, using two or more cameras to generate a 3D track, or using appearance-based approaches, would also increase the complexity of processing such information and thus might be problematic for performing such calculations in real-time.

Further research into facial motion capture reveals that often Principle Component Analysis (PCA) is used (Chuang et al. 2002, Borshukov et al. 2006, Deng et al. 2006, Pighin 2006, and Bickel et al. 2008).

> "Principal component analysis (PCA) involves a mathematical procedure that transforms a number of possibly correlated variables into a smaller number of uncorrelated variables called principle components." (Wikipedia, 2010e).

Applying principal component analysis to our facial motion capture system could allow for generating a series of orthogonal basis vectors from our blendshapes, thus ensuring linear independence. Chuang et al. (2002) and Deng et al. (2006) use principal component analysis in exactly this way. Given a series of 'training' frames of marker data, they use principal component analysis to determine the orthogonal basis vectors that describe the expression space of those markers. Thus, the use of PCA could allow us to define a mathematically rigorous expression space, and thus could lead to significant improvements in our visual results.

Havaldar (2006, p. 13) describes an alternative approach to solving the problem of generated facial rig parameters not exactly matching the actor's facial expression:

> "Most of these problems can be corrected by a tuning system, which recalibrates the facial library based on feedback from an animator. At Imageworks, the artists and software engineers developed a Multidimensional Tuning System, which takes an artists input to reduce the effects of incorrect mathematical solves. In this system, [after the] FACS solve, the animator adjusts a few frames (typically five to ten, among the many thousands) to look correct. This is accomplished by modifying the weights of poses on a few culprit frames that have resulted from a FACS solve. The tuning system exports this changed data, analyzes it and creates a more optimized FACS library. The new library generated is based on the marker range of motion as well as the changed weighting and uses non-linear mathematical optimization tools. This changed library when used in a new solve now attempts to hit the weighting defined by the animator on the few tuned frames at the chosen poses and also incorporate this change programmatically on to various other frames causing the whole animation to look better."

Whilst no specifics of implementation are given, it is possible that a similar approach could fix the issues of our facial motion capture system. Further research would need to be performed to test this theory.

Chapter 6

Conclusion

6.1 Future Work

In Section 5.3 we discussed at length some of the areas we believe should be researched and evaluated to determine if they could improve the results of our current implementation. We believe principle component analysis to be the most promising of those areas, due to its extensive use in existing facial motion capture systems (Chuang et al. 2002, Borshukov et al. 2006, Deng et al. 2006, Pighin 2006, and Bickel et al. 2008). Therefore, if we continue the development of our facial motion capture system, this will be our next avenue of research.

Alternatively, Havaldar's suggestion of a "Tuning System" (Havaldar 2006, p. 13) sounds as though it could help improve the results of our facial motion capture system, and certainly warrants further research.

If these above two techniques do not to solve the issues experienced with our facial motion capture system, then one could look into the possibility of increasing the fidelity of the input data (using a dense marker set (Havaldar 2006), capturing 3D data (Firestone 2010), or perhaps using appearance-based (marker-less) techniques (Pighin 2006)).

If one can perform all the calculations of an appearance-based system in real-time it would certainly provide the highest fidelity of data, and thus could possibly generate the best results. In addition, the process of painting markers on the actor's face is time consuming and, as previously mentioned (Chapter 1), prohibitive to using the system as a general-purpose input device (e.g. for use as a video game controller). Thus, by using appearance-based techniques, and therefore removing the need for markers, the facial motion capture system would gain the secondary advantage of moving a step closer to becoming a general-purpose input device, which we feel could have exciting applications.

For our facial motion capture system to be considered 'production ready', in addition to improving the results of the current implementation, it would need to be able to track both eye and eyelid motion, and map that motion to the eye and eyelid rotations of the facial rig. As earlier discussed (Section 5.1), the spare processing time of each frame could be utilised for such calculations.

6.2 Conclusion

The goal of this Master's project was to develop a real-time facial motion capture system, capable of analysing a live video stream of an actor's face in order to animate any CG face (Chapter 1). In addition, the hardware had to be developed on a tight budget.

We believe that we were successful in achieving this goal. The system runs in real-time, at a constant and steady 30 frames per second (30Hz) (Section 5.1), and the hardware was developed on a tight budget (Section 5.2).

Our facial motion capture system is capable of animating any CG face based on a live video stream of the actor's face (Section 5.3). The visual quality of this animation has both strengths and weaknesses, as discussed at length in Section 5.3. We feel the strengths of the animation show great promise for future development of the system, and we discuss and outline possible areas of research to improve on the weaknesses of the system in Section 5.3 and Section 6.1.

Chapter 7

Bibliography

A Christmas Carol, 2009. Film. Directed by Robert ZEMECKIS. USA: Walt Disney Pictures.

AHN, S.H., 2005. *OpenGL Vertex Buffer Object (VBO)* [online]. Canada: Song Ho Ahn. Available from: http://www.songho.ca/opengl/gl_vbo.html [Accessed 5 July 2010].

Avatar, 2009. Film. Directed by James CAMERON. USA: Twentieth Century Fox Film Corporation.

BARKER, J., 2005. Tracking Facial Markers with an Adaptive Marker Collocation Model. *In: Proceedings of the 30th International Conference on Acoustics, Speech, and Signal Processing, March 18-23 2005 Philadelphia PA*. 665-669.

BENNETT, D., 2005. The Faces of "The Polar Express". *In: ACM SIGGRAPH 2005 Courses: Digital Face Cloning, July 31-August 4 2005 Los Angeles CA*. New York, NY, USA: ACM Press. Article No. 6.

BICKEL, B., LANG, M., BOTSCH, M., OTADUY, M.A. AND GROSS, M., 2008. Pose-Space Animation and Transfer of Facial Details. *In: SCA '08: Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, July 7-9 2008 Dublin Ireland*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association. 57-66.

BOLTON, D., no date. *How do I do High Resolution Timing in C++ on Windows?* [online]. About.com, NY. Available from: http://cplus.about.com/od/howtodothingsi2/a/timing.htm [Accessed 13th May 2010].

BORSHUKOV, G., PIPONI, D., LARSEN, O., LEWIS, J.P., TEMPELAAR-LIETZ, C., 2005. Universal Capture - Image-Based Facial Animation for "The Matrix Reloaded". *In: ACM SIGGRAPH 2005 Courses: Digital Face Cloning, July 31-August 4 2005 Los Angeles CA*. New York, NY, USA: ACM Press. Article No. 16.

BORSHUKOV, G., MONTGOMERY, J. AND WERNER, W., 2006. Playable Universal Capture: Compression and Real-time Sequencing of Image-based Facial Animation. *In: ACM SIGGRAPH 2006 Courses: Performance Driven Facial Animation, July 30-August 3 2006 Boston MA*. New York, NY, USA: ACM Press. Article No. 8.

CHOE, B., LEE, H. AND KO H.S., 2001. Performance-Driven Muscle-Based Facial Animation. *The Journal of Visualization and Computer Animation*, 12 (2), 67-79.

CHUANG, E. AND BREGLER, C., 2002. *Performance Driven Facial Animation using Blendshape Interpolation*. Stanford University Computer Science Technical Report CS-TR-2002-02. Palo Alto, CA, USA: Stanford University.

COMNINOS, P., 2006. *Mathematical and Computer Programming Techniques for Computer Graphics*. London: Springer.

DACOSTA, C., 2010. *Avatar's Up in the Air - Reviews* [online]. Corve DaCosta's Blog. Available from: http://corvedacosta.wordpress.com [Accessed 29 June 2010].

DENG, Z., CHIANG, P.Y., FOX, P. AND NEUMANN, U., 2006. Animating Blendshape Faces by Cross-Mapping Motion Capture Data. *In: I3D '06: Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games, March 14-17 2006 Redwood City CA*. New York, NY, USA: ACM Press. 43-48.

DICOVERY, 2009. *Avatar: Motion Capture Mirrors Emotions* [online]. Silver Spring, MD: Discovery Communications. Available from: http://news.discovery.com/videos/avatar-making-the-movie/ [Accessed 29 June 2010].

EKMAN, P., FRIESEN, W.V. AND HAGER, H.C., 2002. *Facial Action Coding System*. 2nd ed. Salt Lake City, UT, USA: Research Nexus division of Network Information Research Corporation.

EMERY, D., 2010. *Microsoft unveils Xbox 'Kinect' motion controller*. UK: BBC. Available from: http://news.bbc.co.uk/1/hi/10284289.stm [Accessed 29 June 2010].

FIRESTONE, H., 2008. *The Future of Motion Capture* [online]. Glendale, CA, USA: COP Communications, Inc. Available from: http://www.copcomm.com/ [Accessed 29 June 2010].

FOX, 2010. *Avatar: Creating the World of Pandora*. Los Angeles, CA, USA: Fox Movie Channel. Also available online from: http://3dup.com/news/view.vfx?nid=327 [Accessed 29 June 2010].

HAVALDAR, P., 2006. Sony Pictures Imageworks. *In: ACM SIGGRAPH 2006 Courses: Performance Driven Facial Animation, July 30-August 3 2006 Boston MA*. New York, NY, USA: ACM Press. Article No. 5.

INTERNATIONAL PHONETIC ASSOCIATION, 1999. *Handbook of the International Phonetic Association: A Guide to the use of the International Phonetic Alphabet*. Cambridge: Cambridge University Press.

JOSHI, P., TIEN, W.C., DESBRUN, M. AND PIGHIN, F., 2006. Learning Controls for Blend Shape Based Realistic Facial Animation. *In: ACM SIGGRAPH 2006 Courses: Performance Driven Facial Animation, July 30-August 3 2006 Boston MA*. New York, NY, USA: ACM Press. Article No. 17.

King Kong, 2005. Film. Directed by Peter JACKSON. USA: Big Primate Pictures.

LAWSON, C.L. AND HANSON, R.J., 1995. *Solving Least Squares Problems. Classics in Applied Mathematics*. 2nd ed. Philadelphia, PA, USA: SIAM.

LORACH, T., 2007. DirectX 10 Blend Shapes: Breaking the Limits. *In:* H. NGUYEN. *GPU Gems 3*. Reading, MA, USA: Addison-Wesley Professional. 53-68.

MA, W.C., JONES, A., HAWKINS, T., CHIANG, J.Y. AND DEBEVEC, P., 2008. A High-Resolution Geometry Capture System for Facial Performance. *In: ACM SIGGRAPH 2008 Talks: Smile for the Camera, August 11-15 2008 Los Angeles CA*. New York, NY, USA: ACM Press. Article No. 3.

META MOTION, 2010. *Standard Deviation 2D Face Tracker* [online]. San Francisco, CA, USA: Meta Motion. Available from: http://www.metamotion.com/hardware/face-trackers.htm [Accessed 29 June 2010].

MICROSOFT, 2004. *Converting Between YUV and RGB* [online]. Redmond, WA, USA: Microsoft. Available from: http://msdn.microsoft.com/en-us/library/ms893078 [Accessed 5 July 2010].

MICROSOFT, 2010a. *About Media Foundation* [online]. Redmond, WA, USA: Microsoft. Available from: http://msdn.microsoft.com/en-us/library/ms696274%28VS.85%29.aspx [Accessed 5 July 2010].

MICROSOFT, 2010b. *Asynchronous Callback Methods* [online]. Redmond, WA, USA: Microsoft. Available from: http://msdn.microsoft.com/en-us/library/ms704787%28VS.85%29.aspx [Accessed 5 July 2010].

MOB SCENE, 2009. *Avatar: Performance Capture Featurette* [online]. Mob Scene Creative + Productions. Available from: http://www.mobscene.com/avatar/ (video 03) [Accessed 29 June 2010].

Monster House, 2006. Film. Directed by Gil KENAN. USA: Columbia Pictures.

NVIDIA, 2009a. *OpenCL Programming Guide for the CUDA Architecture, Version 2.3.* Santa Clara, CA, USA: NVIDIA.

NVIDIA, 2009b. *NVIDIA OpenCL Best Practices Guide, Version 2.3.* Santa Clara, CA, USA: NVIDIA.

OSCARS, 2010. *Nominees & Winners for the 82nd Academy Awards* [online]. Beverly Hills, CA, USA: Academy of Motion Picture Arts and Sciences. Available from: http://www.oscars.org/awards/academyawards/82/nominees.html [Accessed 29 June 2010].

PIGHIN, F., 2006. Introduction. *In: ACM SIGGRAPH 2006 Courses: Performance Driven Facial Animation, July 30-August 3 2006 Boston MA.* New York, NY, USA: ACM Press. Article No. 1.

RITCHIE, K., CALLERY, J. AND BIRI, K., 2005. *The Art of Rigging: A Definitive Guide to Character Technical Direction with Alias Maya, Volume 1.* San Rafael, CA, USA: CG Toolkit.

SAGAR, M., 2006. Facial Performance Capture and Expressive Translation for King Kong. *In: ACM SIGGRAPH 2006 Sketches: Face to Face, July 30-August 3 2006 Boston MA.* New York, NY, USA: ACM Press. Article No. 26.

TEO, L., 2010. The Making of Avatar. *3D World* issue 125, January 2010, p. 30-36.

The Matrix Reloaded, 2003. Film. Directed by Andy WACHOWSKI and Larry WACHOWSKI. USA: Warner Bros. Pictures.

The Polar Express, 2004. Film. Directed by Robert ZEMECKIS. USA: Castle Rock Entertainment.

TURKU PET CENTRE, 2010. *nnls.c File Reference* [online]. Finland: Turku PET Centre. Available from: http://www.turkupetcentre.net/software/libdoc/libtpcmodel/nnls_8c.html [Accessed 2 July 2010].

TWIST, S., 2010. *CGI Techniques - Real-Time Tracking of Facial Markers.* Bournemouth: Bournemouth University.

WIKIPEDIA, 2010a. *Circular buffer* [online]. Wikimedia Foundation, Inc. Available from: http://en.wikipedia.org/wiki/Circular_buffer [Accessed 5 July 2010].

WIKIPEDIA, 2010b. *Frame rate* [online]. Wikimedia Foundation, Inc. Available from: http://en.wikipedia.org/wiki/Frame_rate [Accessed 5 July 2010].

WIKIPEDIA, 2010c. *Low-pass filter* [online]. Wikimedia Foundation, Inc. Available from: http://en.wikipedia.org/wiki/Low-pass_filter [Accessed 1 July 2010].

WIKIPEDIA, 2010d. *Moving average* [online]. Wikimedia Foundation, Inc. Available from: http://en.wikipedia.org/wiki/Moving_average [Accessed 1 July 2010].

WIKIPEDIA, 2010e. *Principle component analysis* [online]. Wikimedia Foundation, Inc. Available from: http://en.wikipedia.org/wiki/Principal_component_analysis [Accessed 7 July 2010].

WILSON, D., 2007. *YUV pixel formats* [online]. FOURCC.org. Available from: http://www.fourcc.org/yuv.php#YUY2 [Accessed 5 July 2010].

Chapter 8

Appendix A

The following is a reprint of our previous report, *CGI Techniques - Real-Time Tracking of Facial Markers* (Twist 2010).

CGI Techniques - Real-Time Tracking of Facial Markers

Steven Twist
MScCAVE
stevetwist@gmail.com
f9097438@bournemouth.ac.uk

May 17, 2010

Abstract

This report presents a novel approach to a real-time facial marker tracker, using Barker (2005) as a basis. Such a tracker is required for our development of a real-time facial motion capture ("mocap") system. Significant improvements to Barker's approach are made to facilitate many tracking markers in close proximity. By implementing our approach on the GPU, we achieve real-time performance at 30Hz with 40 facial markers.

1 Initial Approaches

It was initially proposed to base this project on a GPU based real-time feature tracking algorithm described by Sinha et al. (2006). The algorithm appeared to be an ideal solution to tracking facial markers on the GPU.

However, in an informal meeting with Dr. Hongchuan Yu on 23rd March 2010, the nature of facial motion was discussed, and Dr. Yu advised that such a tracker was probably over engineered for facial tracking. He advised researching algorithms specifically designed to deal with the small inter-frame translations found with facial markers, as this would probably result in better overall performance.

In my earlier tracking research, Lucas et al. (1981) became apparent as a seminal paper on feature tracking, used as the basis of countless tracking algorithms. Whilst Lucas et al. present an efficient algorithm for tracking features across frames of a video, given small inter-frame translation, it was the less efficient "obvious technique for registering two images" (ibid. p. 675), mentioned as an existing technique, that caught my attention.

This simple algorithm finds the new location of a feature by testing all possible configurations of that feature within a window. The test compares the pixels of the feature from the previous frame with the pixels of the current configuration from the new frame, to calculate some measure of the image difference. The configuration with the smallest difference (i.e. that 'best' matches the original feature) is chosen as the new location.

Whilst this search is "very time consuming" (ibid.), what concerns us with a real-time tracker is predominantly performance over efficiency. The GPU is able to evaluate many instructions in parallel (MacResearch 2009), and thus can easily test each configuration in parallel for a given feature. Because of this, it can achieve real-time performance despite the evident inefficiencies of the approach.

Our initial tracker implementation was based on this approach. Each feature to be tracked has a 'search window' which defines the area of pixels in which we test for the 'best' configuration of the feature. The image distance calculation was based on a least-squares type measure given by Lucas et al. (1981 p. 674):

$$difference = \Sigma_{x \in R}(F(x+h) - G(x))^2$$

where $F(x)$ and $G(x)$ are functions "which give the respective pixel values at each location x in two images, where x is a vector" and "We wish to find the disparity vector h that minimizes some measure of the difference between $F(x+h)$ and $G(x)$, for x in some region of interest R." (ibid.)

Our implementation achieved real-time performance at 30Hz with 40 markers being tracked. In theory, this approach should have stably tracked the features, as it finds the 'best' fit of the feature on the new frame. In practice, however, it is fundamentally flawed.

Information captured by a camera is sampled into pixels. It is highly likely that the true displacement of a feature is some fractional multiple of pixels. However, the above approach does not allow for such fractional translations. It finds the whole pixel translation at which the difference equation evaluates to the smallest value. This results in a slight drift between frames, where the true translation and calculated translation of the feature are slightly different.

Over one frame this drift is negligible, but because we use the new position of the feature as our starting point for the next frame, it is permanent. Thus, over a number of frames the drift propagates itself into a serious problem. The feature will no longer represent the original feature we were tracking, and as such we consider the tracker to have 'dropped' its feature. In almost all test cases, this drift resulted in stable tracks lasting for less than a few seconds before the tracker dropped its feature. For the practical application of this facial marker tracker, where stable tracking of all features would be required for minutes, if not hours, of continual tracking, this was unacceptable.

One solution attempted was to not update the feature pixels each frame, and instead to try and find the original feature on every subsequent frame. This resolved the issue of drift due to minor translational inaccuracies, denying that drift from propagating into the subsequent frame's calculations. However, another equally severe issue was introduced. Due to the deformation of the feature and the change in lighting conditions caused by facial motion, configurations other than the true displacement of the feature would often result in a smaller difference value. This introduced another form of drift that would again cause the tracker to drop its feature.

With both approaches unable to provide satisfactory tracking, it was decided to go completely back to the drawing board, and approach the problem from an entirely different direction.

2 The Final Approach

Rather than tracking generic features over a sequence of frames, our approach takes a fundamentally different perspective on the problem, identifying and isolating only the tracking markers from the image.

The initial inspiration for this approach came from a field within Computer Vision called "Augmented Reality" (Diggins 2005), which often uses a number of simple image processing techniques to separate and identify information in an image.

In our implementation, the inspiration for facial marker detection came from a promotional image of the facial motion capture process used in *Avatar* (2009). In Figure 1, the facial markers quite clearly consist of blue and green dots. This led to questioning the choice of blue and green, which could possibly be answered by considering the video data being tracked.

Video data consists of red, green and blue information. Blue and green tend to strongly contrast human skin tones (Bergh et al. 1999), which tend to be predominantly red. This is the reason that blue- and green-screens are commonly used to extract an actor from the background plate using so called "chroma key" algorithms (ibid.). We use similar algorithms to identify predominantly blue and predominantly green pixels in our image, and thus to extract our markers.

Figure 1: Facial motion capture in *Avatar* (2009). (Image from Petit 2010).

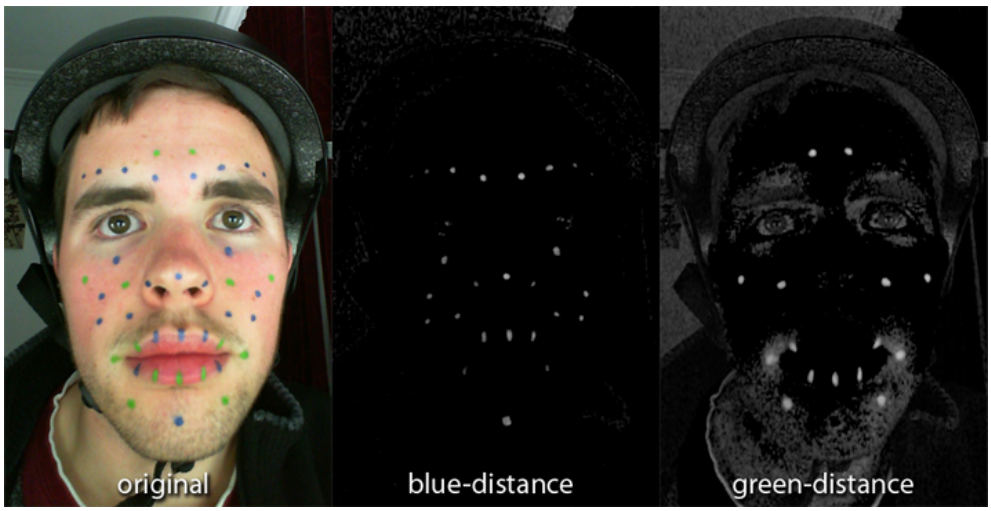


Figure 2: The results of applying the blue and green distance calculations to an original video frame.

In Bergh et al. (1999 p. 158) an efficient distance calculation determines how far from pure blue a pixel is:

$$d = 2 * B - R - G$$

Where R, G, and B range from 0.0 to 1.0, and represent the red, green and blue intensities respectively.

This equation can easily be adapted to calculate the distance from pure green:

$$d = 2 * G - R - B$$

Both equations return a distance value where $d = 2.0$ indicates a pure-blue/-green pixel, and $d <= 0.0$ indicates a pixel that is entirely non-blue/-green (be that a grey pixel, or a pixel of some other colour). Figure 2 illustrates the results of these calculations.

These distance values can be remapped and clamped between a lower and upper threshold, to remove unwanted interference from pixels that are slightly blue/green, but that don't truly belong to the tracking markers (see Figure 3).

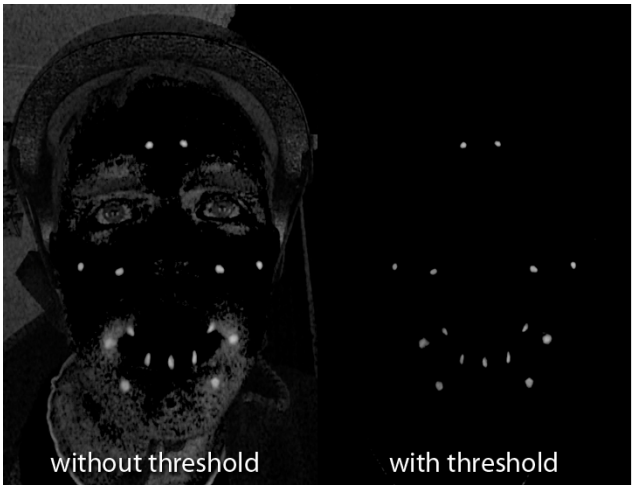


Figure 3: Applying a lower threshold to the green-distance image removes unwanted interference from areas of the face that get detected as slightly green.

Figure 4: Our diffuse lighting setup. The subject sits directly in front of the lights, as if they were sat at the desk.

To identify the markers clearly, a good general illumination of the face is required. To achieve this on a tight budget, household table lamps with white lamp shades (acting as diffusers) and day-light bulbs (to give a full spectrum of illumination) were setup around the subject (see Figure 4).

Given that we're now able to identify our markers, we can take an approach as outlined by Barker (2005) to actually track them. Barker tracks white markers by comparing the luminance of each pixel to a threshold in order to identify the markers. The position of the marker on the current frame is then calculated as the centre of mass of this luminance-threshold image within a given search window around the previous position. The calculation used to determine the centre of mass is not given.

In contrast, we use the above chroma key algorithm to detect our markers. In practice, we found that blue and green markers contrast skin tones much more strongly than white markers using a luminance calculation, and thus are detected with fewer errors.

Using Barker's approach as a basis, our approach is as follows. Given the motion characteristics of a marker (i.e. it's position and velocity over a number of previous frames) we predict its new location using a simple velocity/acceleration calculation. We take the change in velocity over the past few frames as the current acceleration. We sum this acceleration and the current velocity to calculate the new velocity. This velocity is added to the current position to predict the new position of the marker.

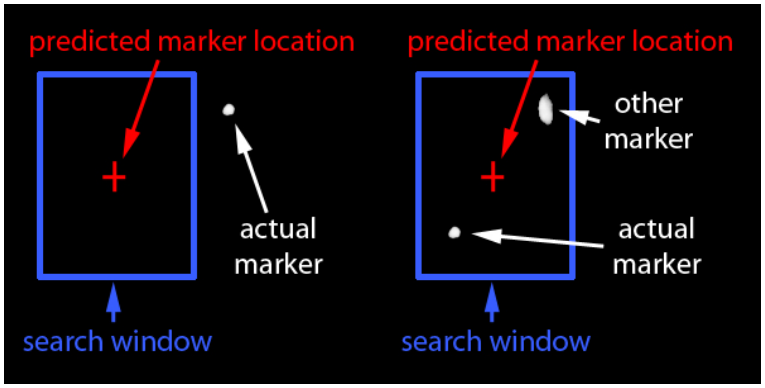


Figure 5: Left: The search window is too small, causing the marker to fall outside the search window (centred around the predicted location). Calculating the centre of mass of the pixels inside the search window will therefore return 0, as there are only black pixels inside the search window. In these scenarios, we take the existing location of the marker to be its new location. Right: The marker we're tracking falls inside the search window. However, the search window is too large, encompassing another marker within it. The centre of mass calculation will not resolve the centre of our marker, but rather some point between the two markers, giving a false track for our marker.

This predicted location is used as the centre of a search window, the dimensions of which are customisable on a marker by marker basis. Within this search window, we calculate the centre of mass of one of the two 'distance' images (depending on if the marker is tracking a blue or green dot) as follows:

$$centreOfMass = \frac{\Sigma_{x \in R}(x * i_x)}{\Sigma_{x \in R}(i_x)}$$

Where x is a vector position within the search window R, and i_x is the value from the distance image at position x. The result of the centre of mass calculation is a vector position taken to be the new location of the tracking marker for the current frame.

For each marker the search window must be large enough to encompass its new location. If the physical marker falls outside the search window we are unable to detect the centre of mass, and we take the existing location of the marker to be its new location (see left of Figure 5). However, the search window must be small enough to not encompass any other markers within it, as this will incorrectly affect the centre of mass calculations, giving a false track for our marker (see right of Figure 5).

We make two significant improvements to Barker's (2005) approach that allow for more tracking markers in close proximity to be tracked without interference from their neighbours.

Firstly, unlike Barker who searches within a window centred around the previous location, we search around a predicted new location, as mentioned above. The closer the predicted location to the true location, the smaller our search window can be and still contain the marker. The smaller the search window the better, not only to ensure no interference from other markers, but also for computational cost as less pixels need to be evaluated within the above centre of mass calculation.

Secondly, by utilising two colours of marker, we can effectively halve the perceived density of markers in the image. Blue markers only interfere with other blue markers. Likewise for green markers. Thus, in areas where markers require larger search windows (for example, the lower lip requires a large vertical search window due to its fast vertical motion) we can

make neighbouring markers the opposite colour, avoiding interference from those markers (see Figure 6).

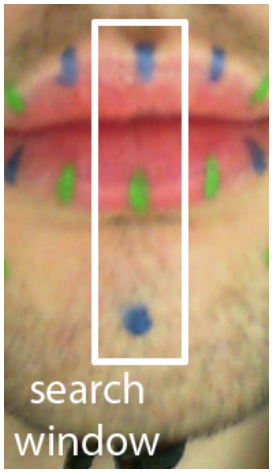


Figure 6: By interleaving markers we allow for much larger search windows without encompassing neighbouring markers. In this example, the green central marker on the lower lip requires a large vertical search window, due to its capability for rapid changes in vertical motion (the actual size of the search window is heavily exaggerated here for illustrative purposes). If the markers on the upper lip and the chin were also green, they would be detected within our marker's search window. However, by making them blue, we remove them from the green-distance image entirely. As far as our marker is concerned, those neighbouring markers do not exist.

3 Implementation

The above approach to facial marker tracking is a highly parallel problem. Thus, by utilising parallel computation on the GPU, our implementation easily achieves real-time performance (Section 4).

The green/blue distance calculations, and lower/upper thresholding, are calculated in parallel on the GPU for each pixel in the video image. The resultant 'distance' images are kept on the GPU for use in the core tracking algorithm. Each tracking marker can be processed independently. The centre of mass calculation is performed for each search window in parallel.

Optimising code for the GPU requires special attention to loading in certain sized chunks of memory, and accessing memory in particular patterns (NVIDIA 2009a and 2009b). For our particular GPU (a NVIDIA® GeForce® GTX 260), it is optimal to load in 16 pixels (64-bytes) of consecutive image data at a time, as doing so takes 1 memory access operation, rather than 16, achieving much greater performance (ibid.). Restrictions are also placed on the total number of threads permitted per 'work-group' (ibid.) (a work-group is basically a group of threads that share resources, and thus are able to share results between themselves (Khronos 2009)).

In our case, with each thread processing a single pixel, it was deemed that 256 threads (16x16 pixels) was optimal, such that we could load in 16 rows of 16 consecutive pixels, whilst maintaining a total number of threads for that group less than the maximum permitted by the hardware. It also ensured that we had enough work-groups active to enable the GPU to context switch work-groups to hide memory latencies, also increasing performance (NVIDIA 2009a and 2009b).

Thus, each search window was broken down into 16x16 pixel sub-windows. If a search window's dimensions were less than a multiple of 16, any redundant pixels of a sub-window

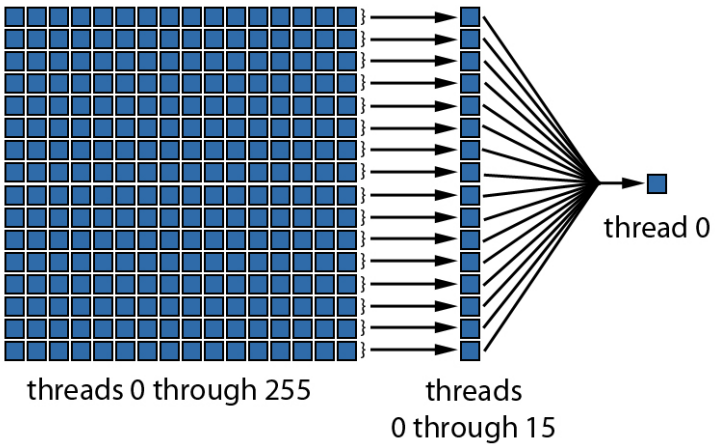


Figure 7: Illustrating how 256 threads work together to calculate and sum 256 values into 1 value, in parallel.

were simply ignored. Each sub-window was processed, on the GPU, by a single work-group. Each thread in the work group takes its pixel coordinates, and multiplies them by the distance image value for the corresponding pixel. The resultant weighted coordinate is stored into shared memory.

Once all threads have calculated their weighted coordinate, the first 16 threads continue execution (whilst the remaining threads of the work group 'go to sleep'). They each sum the weighted coordinates of a single row. They also sum the distance image values of the same row.

Once those threads have calculated these 16 weighted coordinate and distance image sums, storing the results into shared memory, a single thread continues execution (whilst the other threads 'go to sleep'), summing those 16 values (1 value per row) into a final value. This value is stored into global memory, allowing the CPU to read it back from the GPU (see Figure 7).

Once all GPU calculations have completed, the CPU reads all the values back for each work-group from the GPU. For all work-groups (16x16 pixel sub-windows) belonging to a single search window, it sums the weighted pixel coordinates, and divides that value by the sum of the distance image values. This gives the final centre of mass for the search window, which is used as the new position for that tracking marker.

Easy setup and customisation of our implementation is of particular importance. Thus, a configuration file is used to describe each tracking marker. Any configuration of markers is allowed, with each marker specifying individual search window dimensions, and whether it tracks blue or green dots. This allows for great flexibility in its intended application as part of a facial mocap pipeline, as the exact nature of the facial markers being tracked (how many, their motion characteristics, and the chosen colour of the marker) could vary during the development of the system, and even from production to production based on the preferences of the mocap operator. In addition, the individual specification of search window dimensions allows each marker's search window to be tailored to the motion characteristics of the point on the face it's tracking.

4 Performance

Using the high resolution CPU timer (Bolton 2010), which based on my CPU clock speed is accurate to 1/2666 of a second, I performed a number of performance tests of the tracking implementation, tracking an actual facial grid of 39 points in a 1280x720 (720p) resolution

Figure 8: Stills from a tracked video sequence, demonstrating markers following their corresponding dots in a number of extreme facial poses.

webcam stream. The grid contained both blue and green markers, and used search regions of varying dimensions, customised to the specific nature of each tracking marker.

The tracking algorithm alone took on average 1.5ms to evaluate per frame. Thus in theory we could run those calculations at over 650Hz.

Taking into account everything required to track points in the image, from acquiring and converting the webcam stream, through to tracking the points, the calculations take on average 5.08ms per frame. Again, in theory, we could run these calculations at over 190Hz.

This is very promising performance. To run at 30Hz, all calculations for a single frame must take less than 33.33ms. With calculations for tracking taking on average 5.08ms per frame, that leaves 28.25ms per frame for all other calculations required for the motion capture system, hopefully enabling us to run the entire motion capture system at 30Hz.

5 Conclusion

Overall, we were successful in our implementation of a real-time facial marker tracker, able to easily track 40 facial markers at 30Hz. On the whole the tracking is very stable (see Figure 8). However, choosing appropriate search window dimensions in the configuration file, along with choosing appropriate locations on the face for placing the tracking markers, is an evolving and challenging process, requiring substantial testing. As such, more time is required to test and refine these values to resolve occasional issues of the tracker temporarily dropping facial markers.

In addition, the tracking algorithm appears to result in slightly 'shaky' tracks, where even if a marker is stationary, the tracked position floats very slightly within a few pixels around that location. This is most likely caused by noise in the video stream offsetting the centre of mass very slightly. Rather than reworking the tracking algorithm, I believe a simple curve smoothing algorithm can be applied to the tracked data, as it's generated in real-time, to resolve these issues.

In addition, eyes are not currently tracked. As it is impractical to attempt to make the centre of the eye either blue or green, some other approach to eye tracking is required. Such an algorithm is outside of the scope of this project's focus on tracking facial markers.

Bibliography

BARKER, J., 2005. Tracking Facial Markers with an Adaptive Marker Collocation Model. *In: Proceedings of the 30th International Conference on Acoustics, Speech, and Signal Processing, March 18-23 2005 Philadelphia PA*. 665-669.

BERGH, F.vd. AND LALIOTI, V., 1999. Software Chroma Keying in an Immersive Virtual Environment. *South African Computer Journal*, 24, 155-162

BOLTON, D., no date. *How do I do High Resolution Timing in C++ on Windows?* [online]. About.com, NY. Available from: http://cplus.about.com/od/howtodothingsi2/a/timing.htm [Accessed 13th May 2010].

DIGGINS, D., 2005. *ARLib: A C++ Augmented Reality Software Development Kit*. NCCA Bournemouth University, Bournemouth UK.

LUCAS, B.D. AND KANADE, T., 1981. An iterative image registration technique with an application to stereo vision. *In: Proceedings of the 7th International Joint Conference on Artificial Intelligence, August 1981 Vancouver B.C. Canada*. USA: William Kaufman.

NVIDIA, 2009a. *OpenCL Programming Guide for the CUDA Architecture, Version 2.3*. Santa Clara CA, NVIDIA.

NVIDIA, 2009b. *NVIDIA OpenCL Best Practices Guide, Version 2.3*. Santa Clara CA, NVIDIA.

SINHA, S.N., FRAHM J.M., POLLEFEYS M., AND GENC Y. 2006. *GPU-based Video Feature Tracking And Matching*. Technical Report TR 06-012. Department of Computer Science, University of North Carolina, Chapel Hill NC.

TOMASI, C. AND KANADE, T., 1991. *Detection and Tracking of Point Features*. Technical Report CMU-CS-91-132. Carnegie Mellon University, Pittsburgh PA.

Additional References

AHN, S.H., 2007. *OpenGL Pixel Buffer Object (PBO)* [online]. Available from: http://www.songho.ca/opengl/gl_pbo.html [Accessed 13th May 2010].

Avatar, 2009. Film. Directed by James CAMERON. USA: Twentieth Century Fox Film Corporation.

C++ Volumes 4 and 5: OpenGL In-Depth, 2005. DVD. TN, USA: 3DBuzz Inc.

KHRONOS, 2009. *OpenCL 1.0 Specification* [online].
Available from: http://www.khronos.org/registry/cl [Accessed 5th May 2010].

MACRESEARCH, 2009. *OpenCL Tutorials* [online]. Available from:
http://www.macresearch.org/opencl [Accessed 5th May 2010]

NVIDIA, 2009c. *Introduction to GPU Computing and OpenCL* [online]. Santa Clara CA, NVIDIA. Available from: http://developer.download.nvidia.com/CUDA/training/
Intro_to_GPU_Computing_with_OpenCL.wmv [Accessed 5th May 2010].

PENDULUM, 2010. *Pendulum's AlterEgo: Clients & Projects* [online]. Available from:
http://www.studiopendulum.com/alterego/clients_and_projects.html [Accessed 13th May 2010].

PETIT, M., 2010. *Avatar and the Future of Digital Entertainment Creation* [online]. Los Angeles, CA, Animation World Network. Available from: http://www.awn.com/articles/3d/avatar-and-future-digital-entertainment-creation/page/2,1 [Accessed 13th May 2010].