Interim Report

Video Realistic Facial Modelling and Synthesis

Author: Thomas Hartley
Supervisor: Prof. Marshall
Moderator: Dr. Sidorov

CM0343 - 40 Credits

Abstract

By using a pre-existing Active Shape Model to track facial features, the aim of this project is to be able to take the points of data generated from the tracker and use them to drive a 3D facial model presented using the OpenGL ES API. There is also potential to communicate these points of data using Open Sound Control to allow for the information to be re-synthesised on a different device.

Table of Contents

Table of Figures

Introduction

The human face has the power to weave a rich tapestry of emotion and convey information via its changing expressions, without the need for verbal communication. The ability to be able to track, recognise and reproduce these gestures in a potentially altered form is therefore an interesting area to investigate, with many applications ranging from video games (Bernhaupt et al, 2007) to music creation (Kirn, 2011).

The aim of this project is to create a video realistic facial modelling system that can track and synthesise the movements from a user's facial expressions in realtime. The synthesis will take the form of a 3D model that will mimic the user's facial movements. This is to be implemented on an iOS device (specifically an iPad) and will require a number of smaller components to work together in order to function correctly. At its core, a method of accurately detecting and tracking points on a user's face will be required, this method will also have to allow access to the location of the points on the face so that a facial mesh can be built upon it. The system will require a 3D environment to also be present to enable rendering of the facial model. Finally, these features will all have to work within the limitations of the iOS environment, be it processing power or the nature of its closed system.

If time permits then it may be possible to implement a method of communication between devices to allow for real-time transfer of the tracked data so that the user's facial gestures can be synthesised on a new device. This will require the ability to package any data received from the tracker into an easily understood form that can be re-opened and used with minimal effort. Indeed, if such a communication system were to be implemented then the ability to receive and make use of the data captured from facial movements could be used in a wide range of applications, not just in a visual form, but also in a non-visual form such as audio manipulation.

Background

This section will aim to outline some of the key technologies involved in this project. It will begin with discussing the core of the project, the facial tracker, and how the choice of implementation will affect the project. In the initial plan it was thought that the project would make use of an Active Appearance Model (AAM), but after investigation it transpired that the use of an Active Shape Model (ASM) would allow for faster and more accurate tracking in real time as it won't have to concern itself with the texture of the inputted frame (Cootes et al, 1999).

Facial Trackers

A key aspect of this project will be the use of systems that allow for a user's features to be tracked in real time so that the representation of them can be created and displayed in synch with the movements they are making. This poses an interesting problem as any tracker that is used must conform to a list of requirements, notably; lightweight, accessible and reliable.

The tracker has to be lightweight in terms of computing processing involved as it will be required to analyse incoming frames and output them in such a way that a good frame rate is maintained. If a tracker takes too long to process a frame then a knock on effect will occur where each frame takes too long, resulting in slow response times from the 3D model the tracker is driving. The tracker will also need to be accessible with regards to access to its internally stored information, as it will be necessary to get data such as the location of each tracked point and head rotation. Finally, reliability will be a factor as it is desirable for the tracker to be able to continue to function despite movement and occlusion. Movement may be a particular issue on an iPad as there's no guarantee that the camera will be static so the tracker could potentially have to deal with movement of both the face and camera.

From initial research undertaken during the initial plan, three potential face trackers were identified as being readily available and capable of working in an iOS environment. Two were commercial Software Development Kits (SDKs) from Image Metrics and Visage Technologies and one was open source code developed by Jason Saragih.

However, after the initial research it seemed that Saragih's code would be the best option for the project. It would not only allow for much more control over the system (due to direct access to the source code), but there was also existing open source work that had already built upon the code that could be studied and analysed to further the progress of the project. Rather than choosing a black box that would just work with little interaction, it was also decided that a much more openly designed tracker would allow for both finer detailed testing and more streamlined implementation into the iOS environment.

Active Shape Models

Saragih's face-tracker is based on an Active Shape Model (ASM) which has been optimised to enhance its performance with regards to both speed and accuracy. ASMs use statistical models to iterate towards a best fit by examining an initial approximate fit, then going through a series of improved positions until the changes in position become negligible (Sonka et al, 2008). The statistical model that it relies on is built upon a Point Distribution Model (PDM). A PDM is used for describing features that have a general shape but are not rigid and are subject to variation. The PDM uses a training set of normalised images of the object that is to be located and each member of the image set is manually labeled with points so that the object can be represented by a set of points (Cootes et al, 1994). From the set of landmarked images, a statistical model can be built using Principal Component Analysis (PCA) on the vectors that have been used to describe the shapes present in the training set (Cootes and Taylor, 2001). By having a range of landmarks for each image it is possible to fit shapes to new unseen data as the location of the boundaries at a specific point are most likely

to lie within the range of the landmarks (Sonka et al, 2008). From an initial rough (yet educated) guess of where the shape lies, an iterative process is used to examine the region around each landmark and update them to best fit a new found boundary (Fig 1). The search for a boundary is confined to normal of the landmark. This is repeated until convergence is achieved.

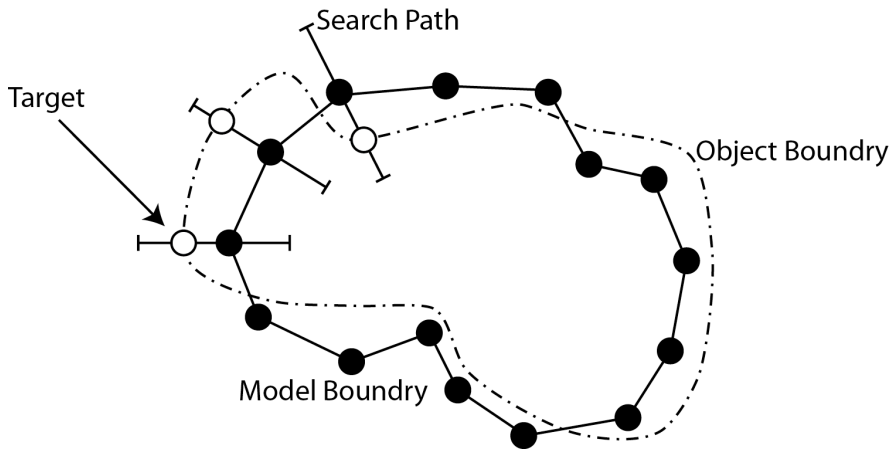


Figure 1: Showing the initial boundary of the model being fitted to the object and an example of the search paths used (adapted from Sonka et al, 2008).

If no new position is found within the search range then the landmark can be left as is and will be ‘pulled’ into position as other landmarks move towards their final position.

The downside to using an ASM is that it is heavily reliant on its initial training data in order to allow new data to be presented and shapes correctly identified. This reliance means that any new shape present that is different enough from the training data won’t be recognised. For example, if a training set consisted of images of a mouth that never opened, then the model would only be able to recognise the shape of a closed mouth rather than one that had been trained in both opened and closed positions.

Saragih's Face Tracker

Saragih's face tracker is a good choice for this project, as not only is it open source, but it's robust in a number of ways. Robustness for this project takes the form of two aspects, how well the tracker performs over time and how well it copes with occlusion. The trackers performance over time is particularly important as it would not be a useful experience if the tracker were to fail and not be able to recover. What's needed is a tracker that's less likely to fail, but also be able to recover quickly if failure does occur. Saragih's face tracker, when compared against a number of other optimisation strategies, appears to have the best performance with regards to both failure recovery and overall tracker success (Fig 2)

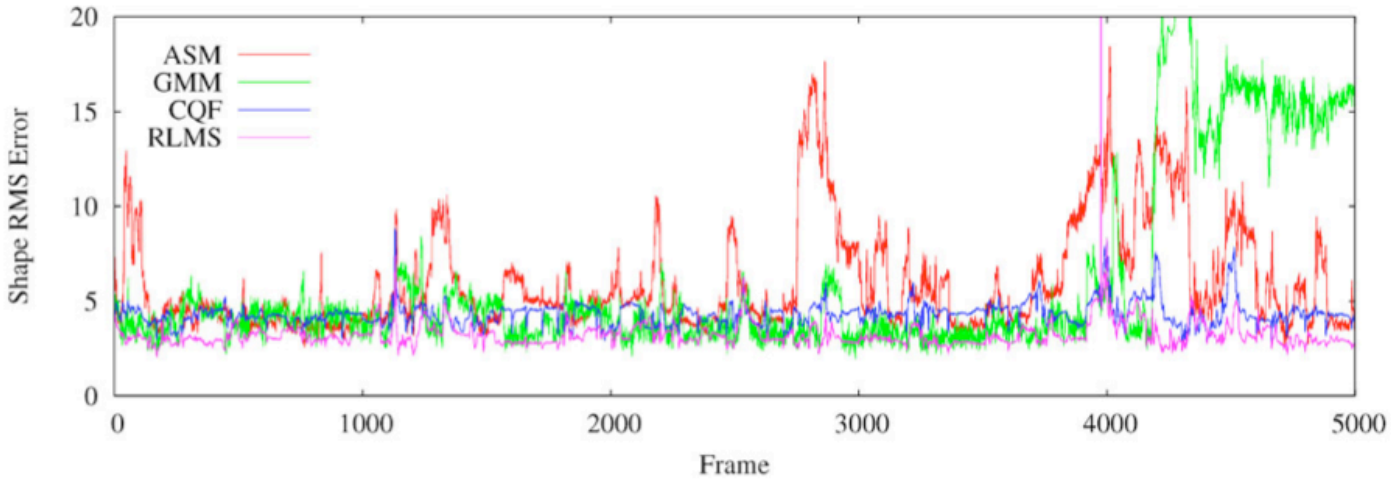


Figure 2. Showing Saragih's face tracker compared to three other optimisation strategies over the 5000 frames of FGNet talking face sequence. Saragih's tracker is shown in purple (Saragih et al, 2011).

Occlusion is handled in Saragih's face tracker using Maximum a-posterior estimate with a German-McClure kernel, and whilst the explanation of that is outside the scope of this report, it should be noted that it offers a significant tracking improvement when compared to other optimisation strategies (Saragih et al, 2011).

Whilst Saragih's code does offer reliable face tracking, it does this by removing some of the complexity involved in the tracking itself. This is done by constraining certain landmarks together by creating relationships between them thereby reducing the number of dimensions that the face tracker has to try work with. McDonald (2012a) explains it by saying "*we can make some guesses about how the features of the face move together. Instead of trying to optimise them independently we can kind of use the information from one to reduce the number of dimensions*". This means that the speed of the face tracker is increased but potentially at the loss of accuracy as it will automatically link certain aspects that, whilst regularly move together, may not necessarily all the time, resulting in a limited set of facial movements that can be captured (Fig 3).

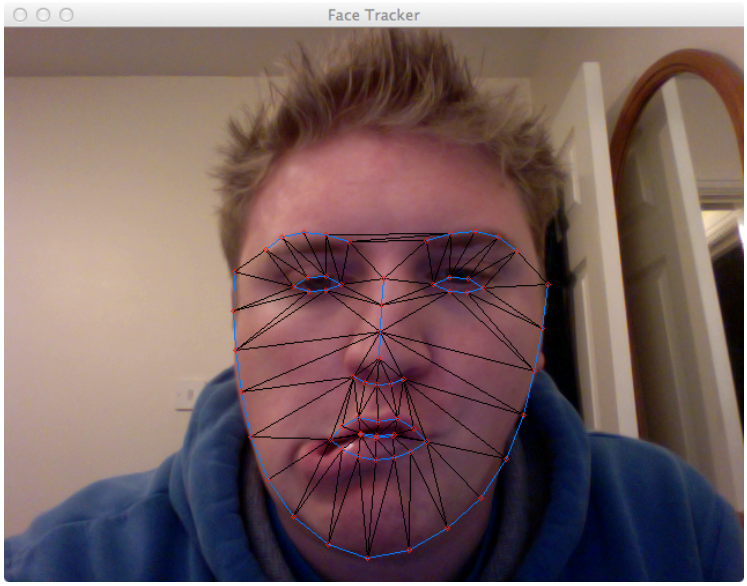


Figure 3. Showing the limitations of Saragih's face tracker when applied to non-standard facial positions.

A second potential downside to using Saragih's face tracker is that it comes with a pre-trained model that can't be adjusted. The model Saragih provides is a generic model, meaning that it's been trained on a large number of images of different people. This gives the tracker the ability to recognise the majority of faces and understand the movements being made. The other side to this is a user-specific model, this is a model that's been trained on one persons facial features only. Whilst this model provides roughly ten times the accuracy of a generic model (McDonald, 2012b), it can only be correctly used by the person whose face it was trained upon without errors occurring. For this project

it makes sense to maintain Saragih's generic model as this will allow the majority of people who come in contact with the application to get good results from the tracker. However, if needed, it may be interesting to propose a method of allowing each individual user to train their face within the application so that a user-specific model can be generated by the application. A similar method can be seen in FaceShift which requires new users to train their faces for use in the application (Faceshift, 2012). This, however, would likely fall out of the scope of this project and would be better suited to future work extending from the project.

FaceOSC and Open Sound Control

Because Saragih's face tracker is open source it's allowed other people to work with it and expand upon its initial scope. One example of this is FaceOSC, which has taken Saragih's code and wrapped it so that values that would otherwise remain hidden can be accessed for use outside the tracker. This allows for a webcam to capture and track a users facial movements, and output them to a device that can understand and make sense of this information (Fig 4). This is useful for this project as it means a blueprint of how to access the values that will be needed already exists, meaning that the effort required to work out where the values come from and how to use them will be drastically reduced. However FaceOSC is written in C++ and intended to be used with openFrameworks, meaning that a lot of the data types in the wrapper are specific to openFrameworks itself such as the representation of vectors. This will require thought when comparing and working out how to translate what's been done in FaceOSC into the iOS environment as the same data types won't be available, or alternatively, an openFrameworks framework will be required in iOS to mimic the way that FaceOSC works.

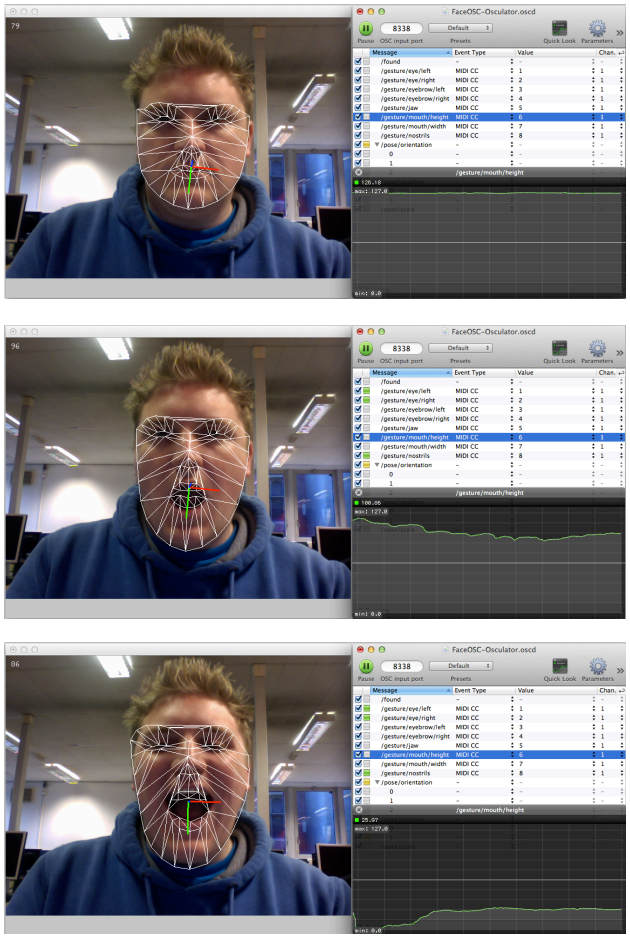


Figure 4. Showing FaceOSC tracking and sending data to another application that can make sense of it. This example is showing mouth height variation being displayed by the OSCulator application.

FaceOSC has another interesting element that could be of use to this project. The OSC section of the title stands for Open Sound Control which is an open source protocol for communicating between computers, multimedia devices and synthesisers. By defining devices as either an OSC client or an OSC server, it becomes possible to transfer the basic unit of transmission, the OSC packet, between the devices over an existing network. The packet has the ability to contain a number of different pieces of information, ranging from 4 byte MIDI messages to double integers (Center For New Music and Audio Technology, 2012). OSC could have the ability to make the communication between devices a far simpler proposition than originally envisaged by providing a simple way of transferring messages. In addition to this, OSC is already being implemented on iOS devices, and whilst there doesn't seem to be a standard framework, it could potentially take relatively little time to implement once the theory of the system is understood. Video 1 shows an iOS application called 'Control'

transmitting information using the OSC protocol which is then received by an OSC server running on a laptop. This would allow any application to be able to ‘hook in’ to these data transmissions, allowing for interesting and varied use of the information.

The use of facial control is a useful tool in a number of environments, ranging from artistic uses to computer interaction for handicapped people. Indeed, several similar systems have used the idea of using facial gestures to control parameters of a separate system. The Mouthesizer is one such system, but instead of using face tracking, it uses a camera mounted in front of the users mouth and then records the shape the mouth makes and maps the information to a controller. This is then used to control the parameters of musical devices. The benefit of this is that the system allows an artistic use of the face to represent non-verbal communication, conveying information about attention, intention, and emotion through facial expressions and gestures (Lyons et al, 2001).

OpenCV Utilisation

As mentioned previously, Saragih’s code makes use of OpenCV, a cross-platform computer vision library, for a number of its data types and at least one of its inbuilt functions for the initial face detection. As OpenCV is intrinsic to Saragih’s face tracker, it will be imperative to be able to use it in the iOS environment. Thankfully this is made easier as there is an OpenCV version that can be compiled into a framework that can then be used within iOS. Whilst Saragih makes extensive use of OpenCV, there is one item that stands out, this is the use of the CvMat data type.

The CvMat is potentially the most commonly used data structure by Saragih, appearing 276 times throughout. CvMat is how OpenCV represents a matrix, however, this matrix allows for more complex contents than single numbers as it allows a type to be defined at it’s point of initialisation. By allowing a type to be defined, OpenCV is allowing the matrix to be used for a large range of applications, for example, the type could be defined as a triplet, this would allow each element in the matrix to store

separate RGB value's that can be used to represent a simple image. This is subsequently how images are represented within Saragih's code. This data type is important in OpenCV as not only is it used to represent matrices, but because there is no vector construct in OpenCV, a matrix with one column or row is used to represent a vector (Bradski and Kaehler, 2008). This could be important as it will allow for a method of representing the data taken from the face tracker within the iOS environment without having to import another framework or define our own data type.

iOS Environment

The iOS environment is a closed one, that is everything that interacts with the core of the device has to be constructed using the Apple made tools and frameworks. It isn't possible to access a part of the environment that isn't itself accessible from within a framework, for example this project would have been impossible previous to iOS 4 when a framework called AV Foundation was introduced that allowed much lower level access to the audio visual elements present in the iOS environment (Apple, 2011).

The iOS environment uses Objective-C but crucially for this project also allows C++ to be used as long as it is declared in the file name. By default Objective-C files use a .m file type, but by changing this to .mm it enables the use of both Objective C and C++ in the same file and is known as Objective-C++. This will be of use as Saragih's face tracker is written in C++ so rather than having to port the entirety of the code into Objective-C, it will be possible to import it into a .mm file that will allow the rest of the Objective-C code to functionally correctly with it.

The iOS environment offers a numbers of ways to access the visual media present, with each offering a higher level of abstraction than the other (Fig 5).

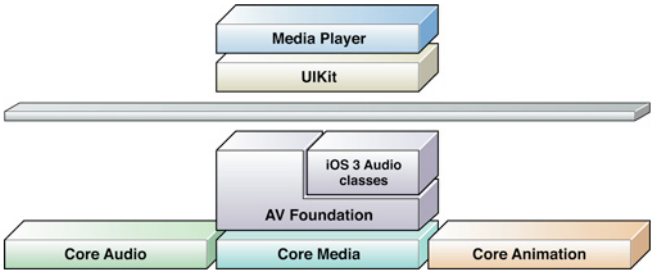


Figure 5. Showing the hierarchy of methods available to used media. High level of abstraction at the top to low level at the base (Apple, 2011).

Apple suggests to "*use the highest-level abstraction available that allows you to perform the tasks you want*" (Apple, 2011). In this project it's required that a connection to the camera is established and that the frames are passed to the face tracker in real time. The Media Player framework provides facilities for movie and audio playback but doesn't allow access to the individual elements that make up the movie that is required. Likewise the UIKit framework allows access to the camera via the UIImagePickerController class but doesn't allow access to the frames that make up the incoming video stream. As these two frameworks offer too high a level of abstraction, we are required to move down a level to the AV Foundation Framework. This framework has the level of control over the media that is required, as via the framework it's possible to access both the initialisation for the camera required as well as the frames that are being read into the system from the camera.

AV Foundation utilises sessions to manage the sort of task it's undertaking, in the case of this project we require the ability to access and capture media from the camera, so it's required that an AVCaptureSession be created. The capture session can be thought of as an object used to co-ordinate and manage the aspects required in capturing from the camera, therefore it's required that the session is given extra information about how we plan to capture and control the data (Sadun, 2012). For the purposes of this project the only information that the session needs to be told is the preset to use, a camera input and an output.

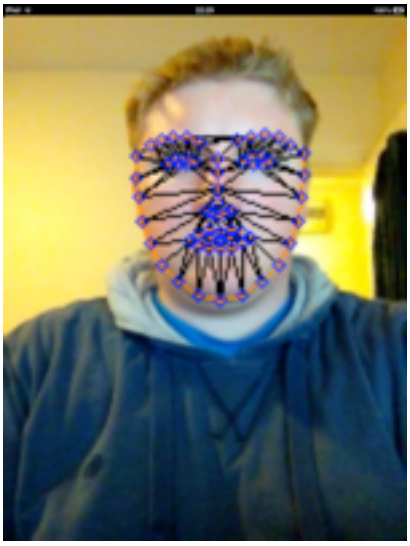
The preset tells the session at what resolution we wish to capture the frames at and are either one of three device specific presets, or one of two fixed resolution presets (Fig 6).

Symbol	Resolution	Comments
AVCaptureSessionPresetHigh	High	Highest recording quality. This varies per device.
AVCaptureSessionPresetMedium	Medium	Suitable for WiFi sharing. The actual values may change.
AVCaptureSessionPresetLow	Low	Suitable for 3G sharing. The actual values may change.
AVCaptureSessionPreset640x480	640x480	VGA.
AVCaptureSessionPreset1280x720	1280x720	720p HD.

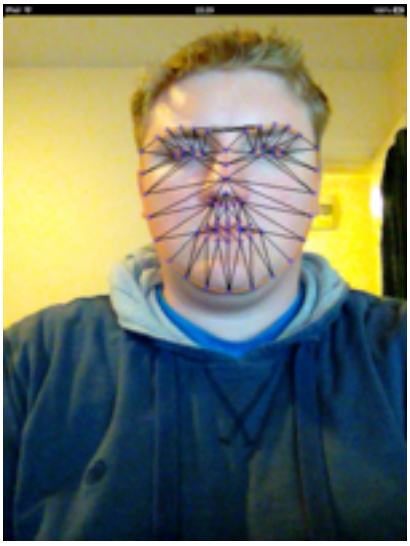
Figure 6. Showing the preset available to the capture session (Apple, 2011)

As each frame will require Saragih's face tracker to be applied to it, the resolution will obviously make a difference as with higher resolutions comes a greater number of available areas to match landmarks to. However, there is a possibility that having a reduced image quality could lead to an increased failure rate as the image could be at such low quality that the landmarks are lost. As an experiment, once the basic tracker was implemented, the preset was tried at Low, Medium and High settings to see how the tracker coped (Fig 7). It was clear to see that although the low setting image was degraded, it still tracked the face well, although further tests will need to be conducted in order to tell how accurate the data is that comes from the face tracker.

AVCaptureSessionPresetLow



AVCaptureSessionPresetMedium



AVCaptureSessionPresetHigh

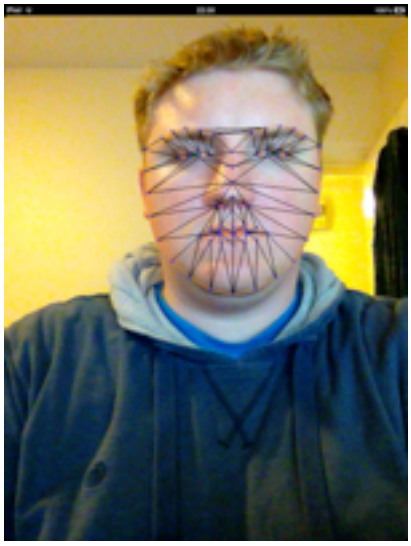


Figure 7. A comparison of the different presets showing how well they're tracked.

The input device is represented using an AVCaptureDevice object which "*abstracts a physical capture device that provides input data to an AVCaptureSession object*" (Apple, 2011). Each input device on the iPad has an AVCaptureDevice representing it, meaning there will be two video inputs, one for each camera. This project will most likely require the use of the front camera as this will allow a user to hold the iOS device and be tracked whilst looking at the screen.

Finally an output is required to allow the data captured by the camera to be accessible from the code. As we require the individual frames rather than the movie file as a whole, it's required that the output used is 'AVCaptureVideoDataOutput' as this will allow each frame to be read into a buffer where it becomes accessible for editing. To access these frames, the use of a delegate method is needed, this method is called 'didOutputSampleBuffer' and allows images to be taken from the buffer as they are captured (Roche, 2011). From here, the buffered images will need to be passed to Saragih's tracker where they can be analysed and the data returned.

3D Environment

As the aim of this project is to be able to match a user's facial movements onto a 3D representation, it will be necessary to be able to perform 3D rendering and computation in an efficient way. The open source standard for such a task is OpenGL for Embedded Systems (OpenGL ES) the latest version of which is supported by Apples devices running iOS 5 and up (Buck, 2012).

The scope of the 3D element of the project will limited in size, it's not the aim to produce a highly detailed 3D model, but rather a much more confined and simpler representation. To undertake this, it's necessary to have an understanding of how 3D models are represented in terms of both their basic geometric data and how a model is built and represented by such information.

Any geometric object in OpenGL ES can be described in terms of points, lines and triangles, the most basic of which, a point, is used to represent a location in 3D space. OpenGL ES makes use of three axes in a Cartesian coordinate system to represent a location in 3D space, with each location being represented with an X, Y and Z value (Buck, 2012). Each position in 3D space represented by these three values is known as a vertex which in OpenGL ES are represented as points (Fig 8).

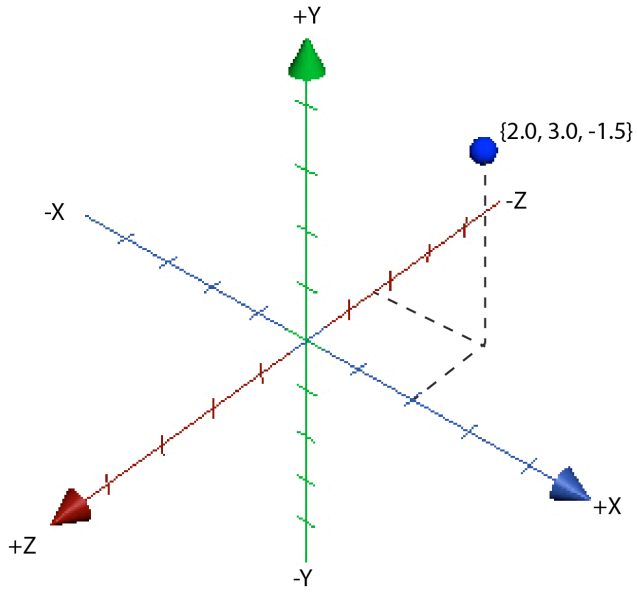


Figure 8. Showing the co-ordinate system used in OpenGL ES where X = 2.0, Y = 3.0 and Z = -1.5 .

Building from a point is a line, these are not mathematical lines, which extend to infinity, rather in the context of OpenGL ES they are line segments. As one point marks a location in 3D space, having two points specified allows for a connection to be made between them. A line is therefore specified in terms of the vertex that mark the lines endpoints (Shreiner, 2009).

A triangle is the most commonly used technique to describe a geometric object in OpenGL ES and is constructed from three lines and therefore three vertex, as each line of the triangle will share a point with its connecting line. To use the triangles to construct a geometric object it is possible to describe each individual triangle and fit them together, however this becomes inefficient as the object grows larger and more complex. As any triangles that are joined together will share a common line, it

becomes possible to reduce the amount of data required to describe the object using shared lines (Munshi et al, 2009). This reduction in information helps alleviate the amount of information required to be placed in memory and also the amount of information that needs to be processed prior to rendering the model. OpenGL ES provides two methods of allowing shared lines in the representation of a series of connected triangles, these are triangle strips and fans. A triangle strip is a series of two or more connected triangles where the first triangle is described using three vertices and every subsequent triangle is described using only two. A triangle fan is similar to a strip in that it's based on using shared sides, but in a fan all triangles share a common centre point (Buck, 2012) (Fig 9).

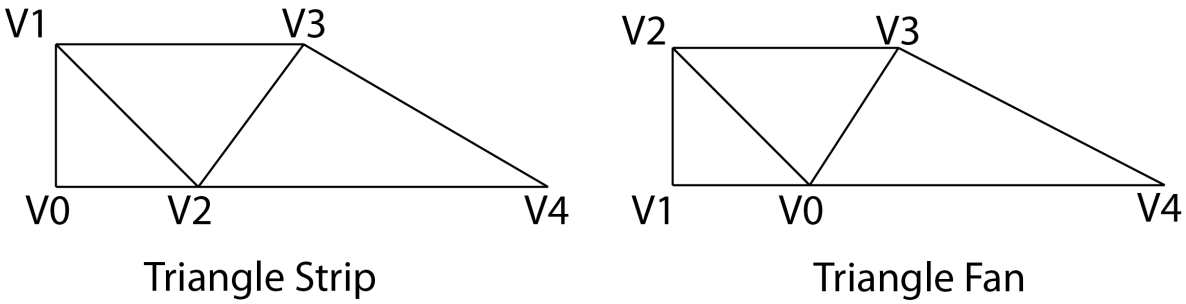


Figure 9. Showing a triangle fan and triangle strip.

By finding the location of each landmark tracked by Saragih's face tracker it should be possible to build up a simple mesh as the landmarked points are connected in triangular shapes and stored in a 3D space (McDonald, 2012). By mapping these landmarks to a 3D view, a model could be generated and updated on the fly when the landmarks in the face tracker change. This model would, theoretically, be the simplest representation possible and may make a good starting point to launch the 3D element of the project. However, if the project moves ahead of schedule then it may be possible to introduce a technique that allows for more complex models to be controlled by the data outputted from the face tracker. This technique is known as skeletal animation and allows for a mesh to be loaded at runtime and then deformed based on the location of an internal 'bone', with each vertex associated to a bone (Munshi et al, 2009). By linking a point of data to a bone, the connected vertices will be transformed at runtime based on the location of the bone. This technique is far more

complex then simply re-rendering the model every time as with the original plan and may prove to be too much to achieve given the time scale and scope of the project.

Approach

Initial Design

This section will talk about the initial design, basic implementation of the application and will attempt to describe the overall design for the project as well. At its core, the initial idea to get Saragih's tracker working is a simple one, some code will capture and send video data to the tracker where it will be returned with any required information to be outputted in the main code again. However, as mentioned previously, Saragih's tracker is written in C++, and whilst it is possible to combine it with Objective-C, it was felt that it would be bad practice to mix up the two. To rectify this, a wrapper is proposed that will take Saragih's method calls and wrap them in Objective-C methods resulting in only a small part of the code using two languages whilst the rest can be written in Objective-C (Fig 10).

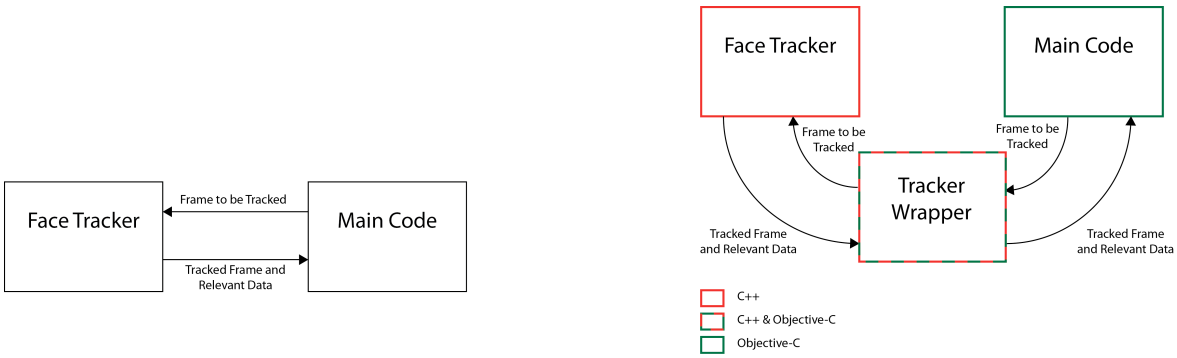


Figure 10. Showing the initial simple design and the subsequent design with wrapper included.

The ‘Main Code’ block represented in these diagrams is an umbrella term that will include such features as the AV Foundation session and an OpenGL ES environment whereby models can be rendered in line with the incoming data from the face tracker. Essentially it will be the code required to live in the iOS environment. A final problem with this is that as Saragih uses OpenCV to represent the image data, and iOS uses it’s own data types, there needs to be some way of converting between the two formats to allow compatibility. This format converter will be placed into a separate class and will fit into the structure as in Fig 11.

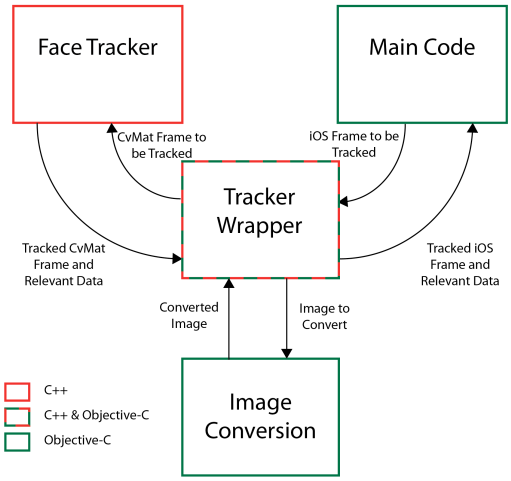


Figure 11. Layout and flow of code.

These layouts are not designed to be precise at this stage, rather likely templates that will try to be adhered to. It may be that this design will be required to change as new obstacles and challenges appear as the project progresses.

An initial design for the layout of the application has also been created (Fig 12). The vertical view will be the default and will show only the 3D model view, however, if the device is rotated to a horizontal position then two separate views will appear. These separate views will show the 3D model as well as the original frame capture with Saragih’s tracking mask laid over the top. This will not only allow for

an interesting feature, but will also be useful for debugging by allowing the 3D model and tracking information to be displayed side by side.

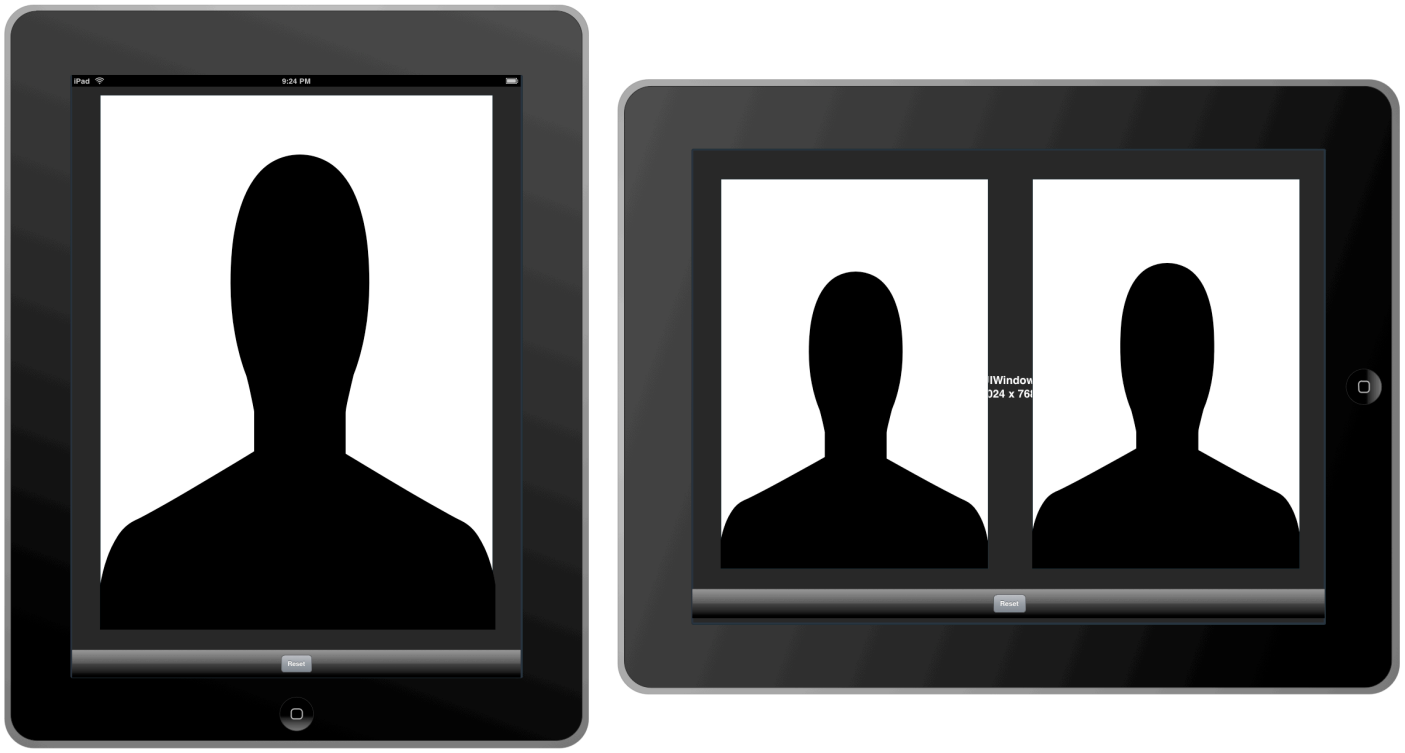


Figure 12. Basic application designs for two device positions.

Basic Tracker Implementation

In order to get the project started and progressing, a basic implementation of the application would be created to ascertain whether the initial high level design would, in fact, work. This basic version would consist of the AV Foundation session, the wrapping of Saragih's code and the implementation of the image convertor. The initial desire was to be able to output the original frame with Saragih's tracking data overlaid.

After an AVCaptureSession was created that allowed for the video data to be captured and read into a buffer, it was possible to begin creation of the tracker wrapper. As stated previously, instead of

directly linking to Saragih's code, it was deemed more desirable to use a wrapper in order to access it. The first challenge that was discovered was that the way Saragih had his code configured was to set up a while loop and then pull each frame from the camera and track it, whereas the AV Foundation set up in iOS required each frame to be put into a queue and then be sent to the tracker from there. This meant that in Saragih's code he could initialise all his models and required variables just before the 'while loop' begins. The first attempt to get Saragih's tracker to work was therefore exactly this, in each frame the model was reloaded as well as all the required variables being re-initialised. This had two major downsides, not only was it inefficient and made the frame rate incredibly slow, it also meant that as the model was being reloaded every time the tracker never had a chance to properly track the face, it was effectively starting from scratch every time, an example of this can be seen in video 2. By stripping out all initialisation calls from the tracking method into two separate methods for initialising values and models, the tracking speed and accuracy was now greatly increased. This more efficient version can be seen in video 3.

By placing Saragih's code into a wrapper it streamlined the code required to track from the app itself considerably, it also allowed for new methods specific to this project to be created that could then be used in the app itself without worrying about the context of the face tracker. There are a number of values that the tracker is passed along with the image to be tracked, this can be the number of iterations for example, that it could be desirable to have control over. By placing Saragih's code inside a wrapper it's now much easier to be able to make a method for specifically changing just the iteration value rather than simply initialising the values as he originally does. This will become an advantage later on when it comes to testing and having the ability to easily adjust parameters to see how performance is affected will be incredibly useful.

As Saragih's code was built using OpenCV, it made use of the IplImage (a data type built on CvMat) type to get the frame from the camera before being converted to a CvMat data type and then passed

to the tracker. This presented a problem as there's no data type in the iOS environment that can be used in place of the CvMat to pass the frames from the camera to the tracker. When the frame is first captured in the AVCaptureSession it's placed into a 'Core Media' sample buffer (core media is a low-level interface for managing audio-visual media), from there it can be converted into a 'Core Image' image before finally being converted to a UIImage, the highest-level way to describe image data (Apple, 2011). With this UIImage it's now possible to insert it into a method (taken from the OpenCV iOS documentation) that converts UIImages to CvMats and vice versa.

With the above elements in place, it was simply a matter of capturing the frames from the AVCaptureSession, sending them to the wrapper where they were converted, tracked and then re-converted and sent back to the main code to be displayed. The result was a relatively smooth output that tracked well and would serve as a solid foundation from which to progress from.

Conclusion

This report has shown the initial research conducted and the steps taken to implement the first phase of the project. From the initial research a number of key areas were highlighted as being important to the project. Saragih's face tracker was chosen as it provided the most open source implementation and its subsequent advantages and disadvantages have been discussed. Following on from this, the areas of the iOS environment that would require the most attention were highlighted, notably the use of the AV Foundation framework to provide the ability to capture and process video data. The OpenGL ES environment was examined and the basic representation of geometric data were discussed and how they could be used to represent information generated by the face tracker. Finally, the potential use of the OSC protocol has been highlighted as a potential means of allowing communication between two devices.

At this stage the basic tracker is now fully implemented into the iOS environment and is working well, however, this is only the foundation for the final application. From here it will be necessary to first begin to extract the data from the internal workings of Saragih's code using FaceOSC as a guide. Once the required values are located it will be possible to build them into the tracker wrapper so that rather than having to query Saragih's code directly, it will be possible to abstract it to a method such as ‘getRoatation’ in order to keep the main code as simplified as possible.

Once this is achieved, the implementation of the 3D environment will need to take precedence as this will be crucial to implementing the 3D model correctly. As discussed earlier this will be based in OpenGL ES with a simple facial model. To begin with, the aim will be to simply attempt to link one parameter from the model to a very basic 3D shape, i.e. a cube, and elicit some response. This could be the simple shape rotating in synch with the users head as proof that the two are linked and interacting with each other. From this starting point the linking between the model and the tracking data will grow more complex until all movements can be synthesised.

At the moment the project is proceeding well, and whilst the initial stages were pre-occupied with learning Objective-C, the iOS environment, research into the project and the basic tracker implementation, the overall plan is slightly ahead of schedule. This time gained will allow the task of implementing methods to access the data in the tracker to be bought forward by two weeks. However, rather than use this extra time to try and add extra features outside the core aims of this project, the time will be used to serve as a buffer incase the 3D implementation turns out to be tricker than first envisaged. Specifically, time has been added to implementing the basic 3D model into the iOS environment. An updated project plan can be found in appendix 1.

References

Apple. 2011. *iOS Developer Library* [Online].
Available at: https://developer.apple.com/library/ios/navigation/
[Accessed: 29th November 2012]

Bernhaupt, R. Boldt, A. Mirlacher, T. Wilfinger, D. Tscheligi, M. 2007. *Using Emotion in Games: Emotional Flowers. ACE '07. pp.41-48.*

Bradski, G. Kaehler A. 2008. *Learning OpenCV*. Sebastopol: O'Reilly Media.

Buck, E M. 2012. *Learning OpenGL ES for iOS*. Addison-Wesley.

Cootes, T F. Edwards, G J. Taylor, C J. (1999). *Comparing Active Shape Models with Active Appearance Models*. British Machine Vision Conference, vol. 1, pp. 173–182

Cootes, T F. Taylor, C J. 2001. *Statistical Models of Appearance for Medical Image Analysis and Computer Vision.* Proc. SPIE 4322, Medical Imaging.236 (July 3, 2001)

Cootes, T F. Taylor, C J. Cooper, D H. Graham, J. 1995. *Active Shape Models - Their Training and Application.* Computer Vision and Image Understanding 61(1) pp.38-59

Faceshift, 2012. *Faceshift: Product* [Online].
Available at: http://www.faceshift.com/product/
[Accessed: 10th December]

Kirn, P. 2011. *Music with Your Face: Artist Kyle McDonald Talks Face-Tracking Music-making with FaceOSC* [Online].
Available at: http://createdigitalmusic.com/2011/07/music-with-your-face-artist-kyle-mcdonald-talks-face-tracking-music-making-with-kinect/
[Accessed 4th December]

Lyons, M. Haehnel, M. Tetsutani, N. 2001. *The Mouthesizer: A Facial Gesture Musical Interface [Online].*
Available at: http://www.kasrl.org/Lyons_chi2001.pdf

[Accessed: 10th December]

McDonald, K. 2012a. *Kyle McDonald Explains FaceTracker* [Online].
Available at: http://makematics.com/research/facetracker/
[Accessed: 27th November 2012]

McDonald, K. 2012b. *FaceShiftOSC Tutorial* [Online].
Available at: http://www.faceshift.com/?portfolio=use-faceshift-to-drive-your-synthesizer
[Accessed: 10th December]

Munshi, A. Ginsburg, D. Shreiner, D. 2009. *OpenGL ES 2.0 Programming Guide.* Addison-Wesley.

Center For New Music and Audio Technology. 2012. Introduction to OSC [Online].
Available at: http://opensoundcontrol.org/introduction-osclivepage.apple.com
[Accessed: 9th December]

Roche, K. 2011. *Pro iOS 5 Augmented Reality*. New York: Apress.

Sadun, E. 2012. *The iOS 5 Developer's Cookbook: Core Concepts and Essential Recipes for iOS Programmers 3rd Edition.* New Jersey: Addison-Wesley

Saragih, J M. Lucey, S. Cohn, J F. 2011. *Deformable Model Fitting by Regularized Landmark Mean-Shift*. International Journal of Computer Vision. 91(2) pp.200-215

Shreiner, D. 2009. OpenGL Programming Guide. Addison-Wesley.

Sonka, M. Hlavac, V. Boyle, R. 2008. *Image Processing, Analysis, and Machine Vision 3rd ed*. Toronto: Thomson Learning.

Appendix

1. Updated Project Plan

Darker shades of colours represent completed work