Thomas Hartley - 1033944

Final Report

Video Realistic Facial Modelling and Synthesis

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

CM0343 - 40 Credits

Abstract

By using an Active Shape Model to track facial features, the aim of this project is to be able to externally represent a users facial movements by extracting points of data and mapping them to a 3D model. Two contrasting methods have been developed, one that keeps all processing functions confined to a single device whilst a second method makes use of multiple devices to share the workload.

1. Introduction

The aim of this project was to create a video realistic facial modelling system that could track and synthesise the movements from a users facial expression in realtime. This was implemented within the iOS environment with the desire that the device be able to both track the users features and then map them to a 3D model and display it upon the screen. To achieve this final goal a number of areas were required to be researched and a number of differing implementations were worked through. This is because the project grew in a fairly organic manner to cope with the problems that the research encountered.

In the interim report a number of conclusions were reached about how to proceed with the project and these were followed through to this report. Firstly, Jason Saragih's facial tracker was to be utilised, as it was found that that this would allow the most flexible implementation possible within the iOS environment. The interim report also proposed a core objective which would be to implement the tracker in such a way that users features could be tracked onto a basic representation of a 3D model. In addition to this a number of features were proposed that could be investigated if there was time, this include areas such as communication between devices and the potential use of more detailed 3D models.

This report will aim to convey the process that occurred from the end of the interim report to the end of the project. Because the project was essentially a research one, the report is broken down into general steps that took place to reach the final objective. This means that the report will begin by talking about the full implementation of Saragih's tracker before moving on to discuss the use of 3D, the additional 3D methods that were required and finally an investigation into allowing a communication system to separate the functionality of the system to work over separate devices. A section on testing and evaluation will attempt to quantify the differing final approaches that were reached, before a conclusion draws together the projects differing strands.

Testing formed an important part of this project and occurred throughout at each major implementation. This happened in the form of white box testing, with the amount of CPU time being spent processing each frame recorded and analysed so that comparisons could be made between the current and previous implementations. The testing that took place at the end of the report was therefore black box testing where a number of aspects were

tested that made no effort to query the inner workings of the system, only it's overall functionality.

Finally, it should be noted that each major implementation had its own application made for it, with a new one being created for each new stage of the project. They can be found in zip files that accompany this report. As each one is essentially a time capsule and remains unchanged from the time it was superseded by a newer implementation, the code that is the most optimal and recent will be of that found in the last two implementations (iOSFaceTracker4 and iOSFaceTracker5). The mapping of implementations to report sections is shown below:

Report Section	Implementation
Basic Tracker	iOSFaceTracker2
OpenGL and Implementation	iOSFaceTracker3
OBJ Reader	iOSFaceTracker4
Keyframe Animation	iOSFaceTracker4
OSC Study and Implementation	iOSFaceTracker5

Whilst all the code produced during the project is available in these uploaded files, selected portions of the code relevant to this report can be found in the appendix.

In addition to these files, a number of videos will also be found that directly relate to this report.

2. Basic Tracker

This section refers primarily to the trackerWrapper.h and trackerWrapper.mm found in the appendix.

At the end of the interim report a basic implementation of the tracker had been achieved within the iOS environment, which could be sent a UIImage and return a drawn on image that was then displayed. To continue on with the project a number of additions needed to be made to this starting point. The first is that information contained within Saragih's

tracker needed to be made accessible from within the application, in particular this consisted of the position of the vertex that make up the 3D mesh as well as information such as rotation and scale. This was done by creating new methods within the trackerWrapper that could be accessed from within the main body of code thereby abstracting the entire process.

Because the tracker formed the bedrock from which the remainder of the project was built upon, it has been thoroughly analysed with regards to where the CPU time is spent and how it could be optimised. Any bugs that impaired the functionality of the tracker were also corrected to allow for a stable foundation to be in place once the tracker was built upon.

2.1 3D Mesh Data

To access the data held within Saragih's tracker, an understanding must be gained of where to find each piece of information, as it is not labeled within the code itself as to where specific values can be found. For this, Kyle McDonalds FaceOSC application (discussed in the interim report) served as a good starting point as he collaborated with Saragih to better understand how he could access certain data values (McDonald, 2012).

To begin to move on from this stage of the project, it was vital to be able to get a copy of the vertex positions of the 3D mesh used by Saragih's tracker, to the main code for it to be utilised for controlling a 3D facial representation. This would be achieved using two methods within the trackerWrapper, the first was called ‘getSpecificPoint’ which had the function of actually extracting the information from the data structures within the tracker. A second method called ‘get3dMesh’ would be used to call the first method over the range of vertex required and create an array that can be returned to the main code for further use. As Saragih's tracker is written in C++ with heavy use of OpenCV, these methods are also required to convert from data structures native to these formats to ones more suited to the iOS environment.

The method required to produce the 3D positional data exists within a class in the tracker (PDM.cc) but it gives no way of returning values outside the tracker. For the ‘getSpecificPoints’ method, this method was reproduced in the trackerWrapper but in such a way that the results were accessible to the main code. The method found within the tracker (‘CalcShape3D‘) works by combining the information from three vectors to form a final vector of the correctly positioned 3D data. The vectors used are two that are found

within a point distribution model (PDM) class and a vector within a constrained local model (CLM) class. The first two vectors were the mean 3D shape vector and the basis of variation, whilst the vector found in the CLM class consisted of local parameters (McDonald, 2013). The method worked by multiplying the local parameters with the variation vector before adding them to the mean 3D shape vector. To replicate this within the trackerWrapper the 'getSpecificPoint' method works by first creating a Mat variable called mesh where the Mat type represents a 2D numerical array (OpenCV, 2010). Because there is no Vector type in OpenCV the Mat variable is initialised with only one column and the same number of rows as there are contained within the vector which contains the mean 3D shape. Finally a type is specified that the Mat variable can hold, as it will be storing precise coordinates, a 64-bit floating point number is to be stored, this is indicated by passing "CV_64F" as an argument. With the variable created, the method used to create the 3D positions in the tracker are recreated in the 'getSpecificPoint' method in the tracker wrapper but now the output is stored in the mesh variable.

The mesh variable now contains a non-interleaved vector of positions, as the positions are stored contiguously (i.e. all X positions followed by all Y positions etc) then it is necessary to navigate to the desired indexes of the array using a stride value. To calculate the stride value, the number of rows contained within the mesh is divided by 3 as it's known that the mesh will contain n numbers of X,Y,Z values so the number of rows should always be divisible by 3. This stride value can then be used to indicate where to jump to when obtaining a coordinate as the X values will all be the within 0 to stride and Y will be contained with in the indexes stride to stride+stride etc. With the ability to now locate specific X,Y,Z values for a given point, the results for each value can be placed within an NSNumber object before being placed into an NSArray and returned. By placing the values into the NSNumber object it becomes much more flexible within the iOS environment and easier to pass around using NSArrays (which can only contain objects) (Apple, 2013)

With a method now available to return a set of X,Y and Z coordinates for a single point on the 3D model, the 'get3dMesh' method can now iterate through the number of known positions, in this case 66, and add the returned array to a new array that will store X,Y and Z values for all positions in the model.

2.2 Rotation and Scale

Again using McDonalds FaceOSC it was discovered that within the tracker there is another vector that contains both the rotation and scale of the underlying mesh. This vector is found in the CLM class and contains the global parameters for the model. By extracting the value at the first position in this vector, a variable can be returned that indicates the users proximity to the camera. The rotation values are found at positions 1 to 3 in the same vector and can be extracted in the same way to give the X, Y and Z rotation amount in radians.

2.3 Optimisation of Image Buffer Conversion

When the initial implementation of the iOS tracker was done, a fairly lengthy process was used to convert from the CMSampleBufferRef that the camera outputs to a Mat that can be used by Saragih's tracker. This was partly due to a lack of time in not being able to optimise the code and also a lack of understanding about how the tracker and the iOS application were fitting together. The initial output from the camera is a CMSampleBufferRef which, when dealing with camera data, contains a CVImagebuffer. By extracting the CVImageBuffer, the information pertaining to the actual image data, including direct access to the locations in memory of the pixels via pixel buffers, becomes available to use (Apple,2013). This, however, was not fully understood to begin with so the original implementation converted the CMSampleBufferRef up to the most abstracted image type possible (a UIImage) before then converting back down to a Mat. Whilst this method worked, it did not make a lot of sense as the Mat type required exactly what the CVImageBuffer contained, i.e. the pixel data. Once a greater familiarity of the iOS environment, Objective-C and OpenCV was gained it was possible to correct this inefficiency. By extracting the CVImageBuffer from the CMSampleBufferRef, it transpired that it was possible obtain the base address as well as the width and height of the buffer. This then allowed for a Mat to be created directly using this data by specifying the size it would be and passing the address to the Mat constructor, a new Mat is constructed which contains the pixel data direct from the CVImageBuffer. This was packaged into a new method within the trackerWrapper class called ‘trackWithCVImageBufferRef’.

With these new methods now in place and the conversion process refined, it seemed prudent to test what exactly was happening in the system to this point before anything else was added.

2.4 Tracker Testing

2.4.1Time and CPU Analysis

An important aspect to consider when testing and analysing the system is how much time is spent on computing each frame, and perhaps more crucially, where the time is spent within each frames life cycle. By analysing these factors it allows a better understanding to be had of how the system is operating and which specific areas should be targeted for optimisation. This time analysis took a structured approach with timings being taken over four sections in the code within the core functionality, namely, the process from obtaining the image, to displaying the tracked information. The four areas that were timed were the initial conversion process (from initial camera buffer to Mat), the tracking process, the drawing of the 2D mesh onto the Mat and finally the reconversion from Mat to UIImage for displaying on the screen. As the app is designed to be used whilst looking at the screen of the iPad, the front camera is used. As each frame captured by the camera is required to undergo some relatively time consuming processing, it's clear that the size of the of the image will be of concern. For this reason, timings have been conducted for three of the AVCaptureSession Presets available, namely, the Low, Medium and High presets, their resolutions can be found in figure 1.

Preset	Resolution
Low	192 x 144
Medium	480 x 360
High	640x480

Figure 1. Showing the resolutions of the presets tested (Apple, 2013)

A simple timer class was constructed and a timer object was created and placed around the code that encapsulated each of the sections were timing was required. Averaged over a period of 50 frames, the results in milliseconds can be seen in figure 2.

Preset	Low	Medium	High
Conversion	3.64	23.48	42.33
Tracking	118.60	129.27	136.79
Drawing	0.73	0.90	1.02
Reconversion	0.53	2.73	4.85

Preset	Low	Medium	High
Avg. Frame Duration	123.62	156.14	186.05

Figure 2. Showing average time duration in milliseconds for a frame over four key areas

From these results it is clear to see that the majority of the time is spent in the tracking code with the initial conversion process taking up the next largest amount of time. The difference between image presets is of interest as whilst the quality of the image returned using the High preset is more pleasing, the trade off between a better quality image and the time taken to process the relevant frame potentially is not worth it in the final context of the project which is to disregard the final tracked image and instead rely on the tracking information that can be gleaned from the frame. In this context it may be wise to gravitate towards the Low or Medium presets to allow a smoother final product, however, as was shown in the interim report, the quality of the low preset once it's had the 2D mesh overlaid makes it almost unusable, especially when shrunk into the corner of a screen.

Whilst Saragih's code is regarded as a black box in this project, an analysis of where the time is spent inside it would provide a greater understanding to its overall integration. As part of Xcode, there is a suite of instruments which can collect a wealth of information such as the CPU usage in real time and how the memory is being used (Apple, 2013). Running the instrument for CPU usage observation allows us to see the percentage of time spent in the various methods that constitute the tracker.

Saragih's tracker consists of eight C++ files and from profiling it, the majority of time seems to be spent in three of these, these are related to a method in the patch class, a method in the faceTracker class and also one in the CLM class.

The patch class consumes 51.8% of the time spent in the tracker. A patch in an Active Shape Model (ASM) is the area around each point to track that can be analysed to find that feature. The ASM that is used consists of 66 tracking points, this means that there will be 66 patches being looked at every frame. The majority of the time spent in this class (41.1%) is spent on an OpenCV method called matchTemplate which seems like it's a method that takes the patch of the image surrounding the point that's being tracked and searches for matches against the trained tracked point that it's looking for. As this is being

done a number of times per frame it can get fairly intensive time wise. The remaining 10% of the time spent in this class seems to be used on iterating through the OpenCV matrices.

The faceTracker class is the entry point to the tracker where the method 'Track' exists which then passes on data to the classes that do the actual tracking. A method in this class called 'ReDetect' accounts for 18.1% of the total time spent in the tracker and, again, the majority of the time (15.7%) is spent on the matchTemplate method.

Finally the method in the CLM class accounts for 13.8% of the time spent in the tracker. CLM is a Constrained Local Model and is essentially the set of points that are being tracked (in fact the method in the patch class that takes so much time is called from this one). The method that consumes the time is 'optimise' which is likely to be pulling the points into their optimised positions so that they resemble what is expected. The majority of the time in this process is spent on matrix operations such as calculating a Jacobian Matrix or iterating through a matrix.

From these results it would seem that the areas available for optimisation within the tracker are limited as they are all complex actions with no comparable iOS API that are optimised to perform the same tasks. However, it should be noted that the above timings were averaged over periods when the frame was tracking and not when it was attempting to discover faces. If images are sent to the tracker that contain no face to perform tracking on, then the tacker will use OpenCVs cvHaarDetectObjects on every frame to try and identify a face until one appears at which point it will begin the tracking process. Video 1 shows how poorly this performs. This is an area that does have the potential for optimisation though, as contained within an iOS framework called CoreImage is a class, CIDetection, that allows for the creation of an object that "*uses image processing to look for features (i.e. faces) in a picture*" (Apple, 2013). The object subsequently returns the coordinates of the rectangle around the found feature. Video 2 shows how much more optimised this is, and therefore smoother, when compared to video 1. This is of interest as if CIDetection were used prior to the image entering the tracker then it would be possible to pass the detected faces location into the tracker along with the image allowing for the OpenCV method currently being used to be disregarded.

2.4.2 Memory Analysis

A recurring problem throughout the early stages of the implementation was that when the application was run, it would crash after a period of time. Initial investigations showed that the length of time from the app starting to it crashing were dependent on the AVCaptureSession Preset being used, with the high one causing a crash much sooner than the low one. With the crash seeming to be dependent on the resolution of the image, it suggested that it could be due to a memory issue whereby the image was being stored but not released which would explain why the larger presets were causing it to crash sooner. To investigate this, the memory analysis instrument was used so as to ascertain what exactly was happening to the memory. Figure 3 shows that as the application is run the memory usage continues to grow until it reaches about 324MB of live memory, and then crashes.

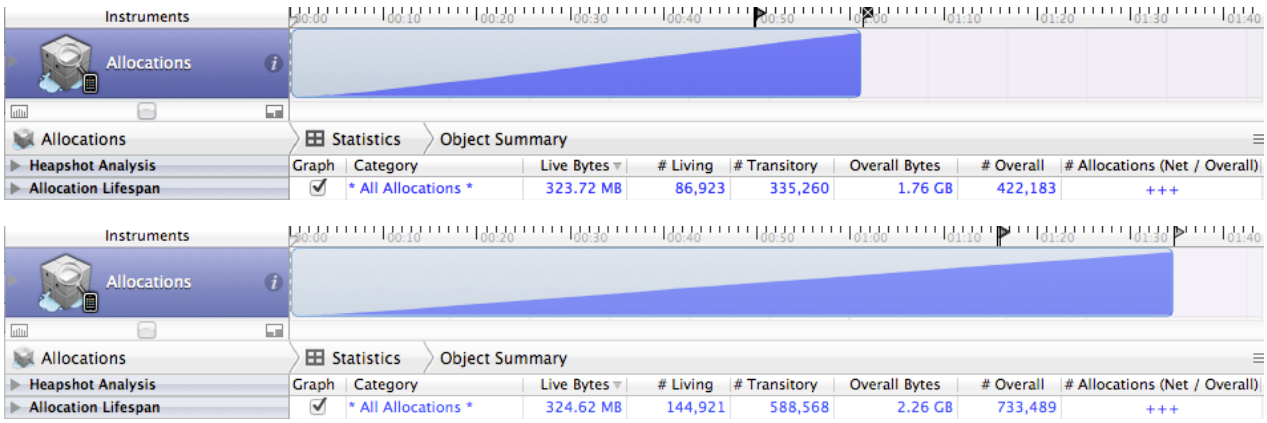


Figure 3. Showing how the memory usage grows for the High preset (top) and Medium preset (bottom) until a crash occurs.

Because it's known that the memory usage grows faster and crashes sooner when a larger image preset is used, it suggests that a memory leak is present somewhere in the method that takes the image buffer from the camera and uses it. The Apple documentation suggests that when code is run that creates a number of temporary objects, as is the case here, then they should be placed in an autorelease pool block (Apple, 2013). This results in any code being placed within this block of code having any objects created in it being released when it is exited. This is perfect for the application, as we only want to keep any information related to a frame for as long as is needed to track and then release its memory just before we get the next frames buffer. With this change made, the memory analysis tool was run again for the medium and high presets with the results shown in

figure 4. With the addition of the autorelease pool block the memory usage remains constant and will run without crashing for as long as was possible to test.

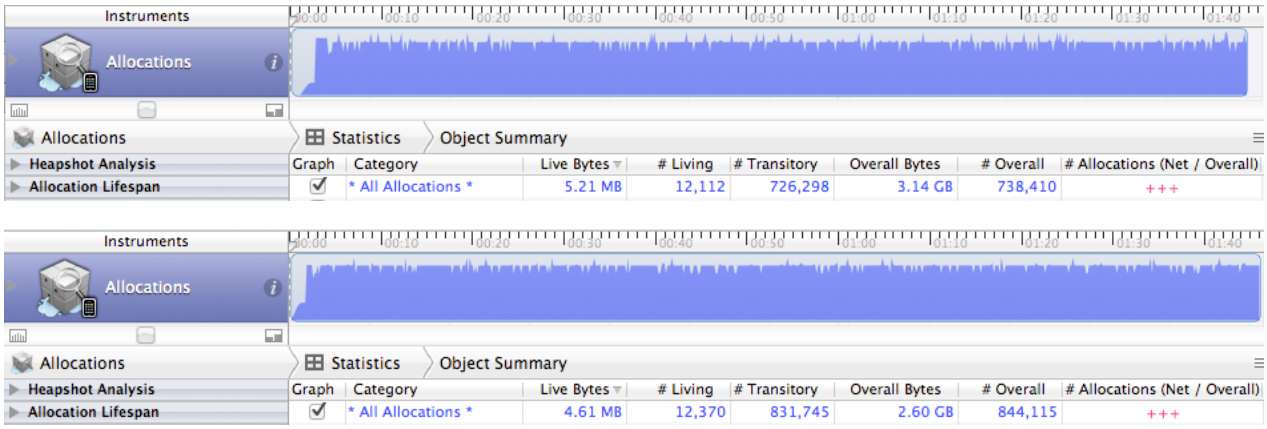


Figure 4. Showing how the memory usage is stabilised for the High preset (top) and Medium preset (bottom) once an autorelease pool is utilised.

3. OpenGL and Implementation

This section refers primarily to the iOSViewController3.h and iOSViewController3.mm found in the appendix.

With the basic trackers functionality fully implemented, a 3D graphics environment is required to be created within the iOS application. This will allow the 3D mesh data being extracted from the tracker to be rendered and subsequently updated in realtime. OpenGL ES is required as this defines the standard for embedded 3D graphics (Buck, 2013), however, there are two distinct branches of OpenGL ES 1.1 and 2.0. OpenGL ES 1.1 makes use of a fixed pipeline which means that anything that relates to the 3D environment (such as lights, colours and cameras) has to be created using built in functions. OpenGL ES 2.0 offers a contrasting way of creating a 3D environment by using a programmable pipeline. This results in the loss of built in functions that OpenGL ES 1.1 but in return offers more control over what can be achieved (Wenderlich, 2011). Apple recommends that OpenGL ES 2.0 is the best choice for newer applications due to its flexibility and increased power (Apple, 2013).

Whilst OpenGL ES is a cross platform API, Apple implements a framework on iOS devices called GLKit that helps to alleviate some of the lack of built in functions with OpenGL ES

2.0. GLKit is used to simplify common programming tasks related to using OpenGL ES and helps to hide the differences between the two OpenGL ES versions (Buck, 2013). In particular GLKit is used to provide functionality in four key areas, these are: Texture Loading, Math Libraries, Effects and View Controllers (Apple, 2013) all of which will be used extensively throughout the project.

3.1 Integration into Project

The first key difference when transitioning from the previous, non 3D environment, was the correct use of the view controller. In an iOS application, the view controller links the applications internal data to its on-screen appearance and also has the role of managing multiple views (Apple, 2013). In the basic tracker implementation, the ViewController class extends UIViewController, the most basic of the view controller classes which only concerns itself with the standard visual behaviours of an application (Buck, 2013). Any implementation that requires 3D in this project subsequently makes use of the GLKViewController class to extend their ViewControllers. By extending to this class, a rendering loop is automatically created which allows a glkViewControllerUpdate method to exist which subsequently allows frames to be rendered to the views display.

The aim at this stage of the project was to link the mesh data, now accessible via the tracker, to the 3D environment allowing it to be rendered in realtime. The desire was to keep this first 3D implementation as simple as possible, so there was no texture or lighting planned at this point and the vertex data used to plot the 3D model would be stored in the simplest way possible. Because no texture and lights are being used, a colour will be specified for each vertex that will then be projected onto the 3D model to add definition to it.

3.2 Linking of Tracker to OpenGL Environment

OpenGL ES allows for vertex data to be specified in two distinct ways, the first is to have separate arrays for the different information (such as vertex position, colour, normal etc) whilst the second way is to have a single interleaved array where all the information for a single vertex sits side by side in an array. This second option is the preferred method for use on iOS devices as it “provides better memory locality for each vertex” (Apple, 2013) thereby allowing the rendering process to become more efficient. To create the desired interleaved data, an array of structs is required. The struct used for this consists of an

array of 3 floats for the coordinates, and an array of 4 floats to represent the colour (R,G,B,A). However, the vertex position data that is obtained from the tracker is just information about where each individual vertex is located in 3D space, it holds no information about how they are connected to form the final 3D representation. For this an array of indices is required that will tell OpenGL ES how each of the vertex connects to one another and can be passed at the same time as the array of vertex structs are. This allows OpenGL ES to perform the calculations necessary to extract the correct vertex information in the required order prior to transferring it to the graphics hardware for rendering. The data for the indices can be extracted from a file that comes with Saragih's tracker, called ‘face.tri’, which is used for the defining the connections on the flattened 2D drawing found in the basic implementation.

To get this data to the graphics hardware, the information must be placed into a buffer object. Buffer objects allow the vertex data to be initialised once within the graphics hardware and then reused without having to reload the information until the vertex data is required to change (Marucchi-Foino, 2012). This is done by first generating a buffer using ‘glGenBuffers’, this buffer is then made active by binding it using ‘glBindBuffer’, finally ‘glBufferData’ is used to reserve space and assign the data we wish to use to the allocated space. The ‘glBufferData’ method is of particular interest to this implementation as it takes an argument titled “usage” which defines how the data will be read and written after it’s been allocated (Shreiner, 2009). Because the vertex positions will be updated constantly, the argument that will be passed will either have to be GL_STREAM_DRAW or GL_DYNAMIC DRAW. The Apple documentation suggests the preferred argument for a project such as this should be GL_DYNAMIC_DRAW as it is designed for scenarios where the information contained within the buffer changes during the rendering loop. This is in contrast to GL_STREAM_DRAW which is used primarily when calculations are performed within the shader and a double buffer is used (Apple, 2013). However, to avoid computational waste, these arguments only need to be supplied to the buffers that contain information that will be updated (i.e. vertex positions) whilst any buffer that can be specified once and never changed can instead be passed GL_STATIC_DRAW. This argument will be used for all other buffers as even though the vertex position will be changing, their colour (or in later cases, texture coordinates) will remain fixed.

Any implementation in this project using 3D will contain a method called ‘setupGL’ where the above functions will exist. Once the data is in the buffer, another function exists called

glBufferSubData that allows to the data to be updated quickly in place, rather than having to create a new buffer and dispose of the old one (Munshi et al, 2009). This method will be used extensively as the data will be changing rapidly as the users facial features move.

With a way of storing vertex data and a way of placing it into buffers for the graphics hardware to consume, the vertex data from the tracker must be linked into the iOS environment. For this, a method is created called ‘updateVertex’ that takes the array of vertex position information from the tracker and interleaves this information into the array containing the struct with position and colour information. As this information is being extracted from every frame that is passed to the tracker, a continuous stream of vertex data is received and sent to the method. Once the new vertex position data is inserted into the interleaved array, the ‘glBufferSubData’ function is called which updates the buffer held in the graphics hardware and renders the new information to the screen. Finally a method called ‘glkView:drawInRect’ is called which first prepares the application for drawing before ‘glDrawElements’ is called. This final method is the one that takes the vertex data from it’s interleaved array and uses the information from the indices array to correctly create the final 3D mesh (OpenGL, 2013). When the application is run, a 3D mesh appears that mimics the users movements across the range of tracking points, this can be seen in figure 5.

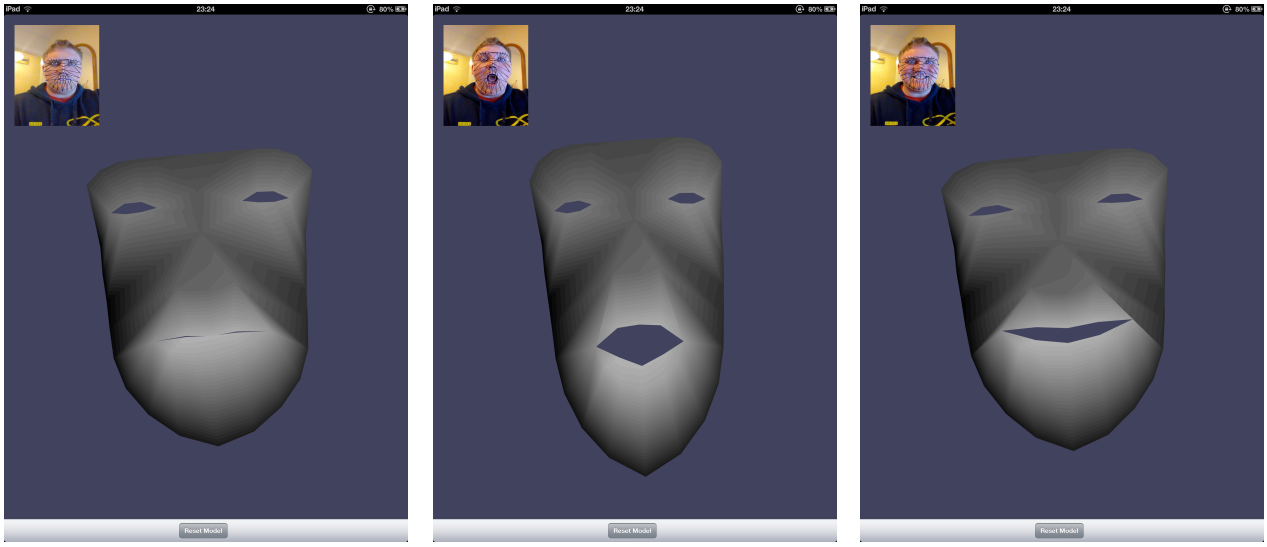


Figure 5. Showing the first 3D implementation linked to the tracked positional vertex data

At this stage only the movements of the users facial features are taken into account when rendering the model, however this misses out a large portion of what the user is doing with

their head as a whole. In reality the user will also likely be rotating and moving their head closer and further from camera which would be useful to convey to the the final 3D model. To rotate and scale a 3D model in OpenGL ES it is not the vertex positions that are altered but rather the co-ordinate system within which they reside in. This is done by defining a new co-ordinate system by rotating a new set of axes relative to the original set whilst keeping the origin the same. GLKit provides a number of functions to make this a fairly simple proposition, by allowing for each axis to have a translation applied to it independently by using a method called 'GLKMatrix4Rotate' (Buck, 2013). This method allows an angle in radians to be passed to it, and the specific axes to be stipulated. This means that the rotation array available from the trackerWrapper class can be called and its values inserted into separate methods for each axis. As these rotation values are updated with every frame passed to the tracker, the coordinates of the 3D environment are also rotated causing the 3D model to render in a newly rotated position. This can be seen in Figure 6.

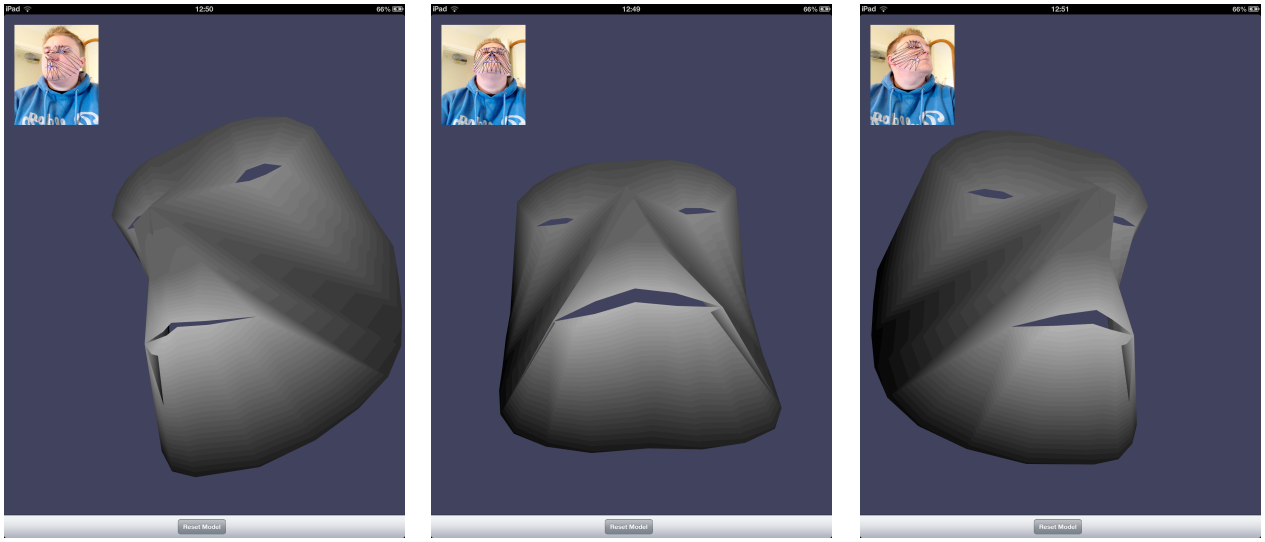


Figure 6. Showing how the 3D model moves with the users head rotation

Scaling works on a similar principle to rotation as a new set of axes is created but displaced in the direction of our choosing. As the face is only moving on the axis that goes into and out of the screen (the Z-axis) then only one method is required, this is 'GLKMatrix4MakeTranslation' in which can be specified the number of units in each axis for which they are to be displaced. In this case, the 'getScale' function can be called from the trackerWrapper (which returns a float) and passed as an argument for the Z axis.

When combined with the rotation this now gives the face the ability to rotate with the users head and move forwards and backwards in sympathy with their motions.

3.3 Incorporating Higher Quality Models

At this stage of the project, the core goals set out in the initial plan have been met. On a high level these were to first get Saragih's tracker working correctly in the iOS environment, and secondly to link up the information being returned from the tracker to a basic 3D facial representation so that it can be controlled remotely. To move on from this point, the main area of improvement needs to be the 3D representation of the users face as at this stage this is the most basic representation possible. Unfortunately there are no methods that allow for 3D models to be loaded into the OpenGL ES (LaMarche, 2009) so in order to proceed with using more detailed models, a system must be devised that allows for a models data to be loaded, converted and rendered within the iOS environment. Once this is achieved, the new model must then be linked to the information being returned from the tracker. However this will pose a number of difficulties as a one-to-one mapping (as previously used) will not be possible so a new method of converting a users facial movements into 3D information will have to be developed.

4. OBJ Reader

This section refers primarily to the object3D.h and object3D.m found in the appendix.

As there is no built in object loader within OpenGL ES, care must be taken when first developing one to ascertain what the best format is to use for passing 3D model data into the application. From research it seemed that the best format to use was the Wavefront file format which consists of two distinct files, ‘.obj’ (OBJ) and ‘.mtl’ (MTL). The OBJ file is used to hold information that relates to each vertex (position, normal, texture coordinates) of a 3D model. The MTL file is used to describe the materials that are used by the 3D model as well as data such as lighting information (Marucchi-Foino, 2012). The final goal of this section will be to have the ability to load an OBJ file into the iOS environment, parse it and extract the vertex information and then render it with an accompanying texture. For this reason, no attention will be paid to the MTL file because as long as the model can be textured correctly, a simple light can be created using GLKit rather than having to use exported lighting data to get it to look a particular way.

4.1 The OBJ Standard

The key reason that the Wavefront format is appealing is that it allows the 3D model information to be described in an ASCII format which subsequently allows for parsing to be implemented relatively easily within the iOS environment. Because the files use ASCII, the data is ordered in an intuitive way with a number of different elements of data easily found. For this project, only a sub-set of the the information available in the OBJ file will be used, namely the vertex information and the face definition information. The vertex data that is required for this project is separated into separately defined areas. For example, the vertex positional data will begin with a ‘v’ character followed by the X,Y and Z positions, then the following lines will consist of all the other vertex positions. the textures and normal information will follow the same format but with ‘vt’ and ‘vn’ line prefixes instead (Wavefront, 2013). Whilst the vertex positional coordinates have been discussed in the interim report, the information used to represent the texture is slightly different to anything encountered thus far so will be examined in further detail.

4.2 Texture information

When the obj file is loaded into the iOS environment, a file that contains the models texture will also be loaded at the same time. This will be a single image that contains every visual element required to be mapped onto the 3D model to give it the correct final look. The interesting aspect to this is how the mapping occurs between the vertex and the texture image. Whilst the vertex position is in 3D space (X, Y, Z), the texture coordinates only map to a 2D space (T,S), and regardless of size, will always be a value between 0 and 1. This texture coordinate system is shown in figure 7.

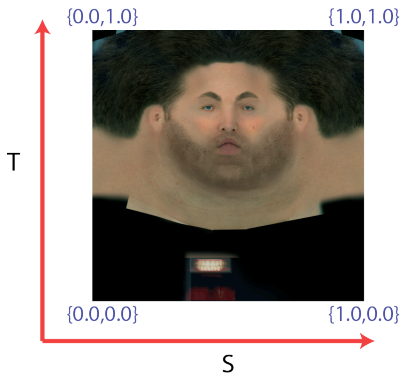


Figure 7. Showing the texture associated with the 3D head with the T, S coordinate system overlaid on top.

When the texture coordinates are specified in the OBJ file, it will be as a T and S value meaning that each vertex will relate to a distinct point on the image file. When three vertex join to form a face, the texture coordinates will carve out a section of the image that will then be rendered over that face at run time (Buck, 2013).

4.3 Face information

All the vertex data is listed in an incremental way, with the vertex at the same position in each list being unrelated to one another, i.e the first 'v' lines information could be unrelated to the first 'vt' lines information. Each vertex is only being described in these lists, they are not in an order that can be used to piece together how the vertex combine to the form the final model. The final piece of information is the face information and it describes how each of the pieces of vertex information is joined. This is a method of saving space as it allows for the number of vertex needed to be reduced as rather than drawing two sets of separate triangles to form a square, by saying which vertex are being used, it allows vertex information to be reused. In the OBJ file the face information is defined by any line that begins with 'f' and is generally followed by three sets of three numbers. The three sets represent the three vertex that make up each face, whilst the three numbers in each set (separated by a '/') represent the index of the position, normal and texture information. An example of the four different types of information encountered to this point can be seen in figure 8.

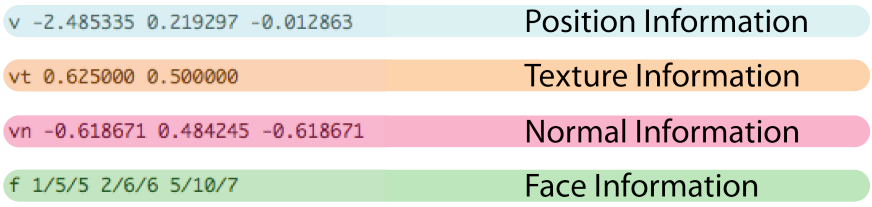


Figure 8. Showing an example of the types of information encountered in an OBJ file

4.4 OBJ Reader Design Decisions

When planning the implementation it seemed that there were two key issues that needed to be resolved before a new object loading class could be made that worked well with the application. The first decision was whether or not to have the class as an entirely separate desktop application that would be given the OBJ file and produce header files that could be imported into the iOS application or if the OBJ file should be parsed at runtime and the data stored in memory. The second decision to be made was whether or not to link the vertex information to the index information when parsing so that the result was a list of

vertex information in the correct order to be displayed, or if to simply pass the index information and the basic vertex information as had previously been done.

During research for the OBJ importing a number of methods were discovered. The first was a perl script that would take the OBJ file and parse it to create a header file that could then be imported into the code and accessed directly from there (Cepeda, 2012). The second was the creation of an object within the iOS environment that would parse the OBJ file and then store the information as internal variables that could be accessed via getter methods (LaMarche, 2009). Clearly there is a trade off between having a quick loading time by using precompiled headers, against the flexibility that comes from loading the vertex information at runtime. For example, if in the future a number of different facial models were to be made available to the user, it would be wise to only allow the vertex data of the desired model to be loaded rather than pre-emptively have all the models information loaded into the environment (Buck, 2013). Using a class to parse OBJ files whilst the application is running also allows for numerous OBJ files to be tested without having to precompile header files specifically for them which could prove to be time efficient when it comes to testing the application. For these reasons it was decided that creating a new class that would allow for files to be passed to it and their vertex information stored internally would be the best approach. This would also allow model objects to be kept if multiple models information were required to be loaded simultaneously.

The second decision whether to pre-link the vertex information to the face indexes or allow OpenGL ES to perform the operation, was again informed out of research. Apple suggests that a best practice for working with vertex data is to “reduce the pre-processing that must occur before OpenGL ES can transfer the vertex data to the graphics hardware” (Apple, 2013). By computing how the model is built by combining the face information with the vertex information at the OBJ loading stage, it is hoped that an increase in loading time will be offset by a reduction in the rendering time required for each frame.

4.5 Implementation

As mentioned previously, the OBJ files that contain the information to be parsed are all in ASCII format so can be dissected using the built in methods associated with an NSString. An NSString is a class that allows for a number of methods to be applied to it and will be the foundation of how the information is parsed by the new OBJ reading class. To begin, a

new NSString is initialised with the entire content of the OBJ file, then using a method called ‘componentsSeparatedByString’ (which is supplied the arguments of a line break character), the separated lines are each read into an NSArray. This results in an array where each line of the OBJ input file resides at a separate index of the array.

The first use of this array containing NSStrings is to fast enumerate through it a line at a time and extract information regarding the number of times each different line of information occurs. Because each line starts with a series of characters that denotes its contents, an NSString method can be used called ‘hasPrefix’ that allows for lines to be selected and counted based on their starting characters. The amount of each line type are required to be counted because, in order to increase efficiency, standard C arrays will be used and will be required to have space allocated to them. C arrays are being used as they are fast and use little memory compared to the NSArrays (Stevenson, 2010) and the lack of conveniences won’t make a difference as they will be used only to store the information and nothing else. With the required C arrays allocated the correct amount of space in memory, the array of lines is again fast enumerated through and the information is extracted into the relevant C arrays. To get the information from each line, a number of NSString methods are used to separate the line from the initial prefix, and then to divide the information either by white space or by a ‘/’ depending on the line type being used.

Because the face information is going to be used at this point to match up the vertex information to create the faces, care must be taken in deciding how to store the information gained from each line. The aim is to be able to match the required vertex information to an index that is specified by a number in the face line and place them into a new array that will incorporate all the required vertex data in the order it will be rendered. For this reason, each lines information was kept in separate arrays. For example, the positional data would exist in individual arrays for X, Y and Z coordinates whilst the face data would exist in three separate arrays for position, normal and texture indexes. By then iterating through the face indexing information the correct vertex information can be extracted and placed into an NSArray within the main array. This concept is show in figure 9.

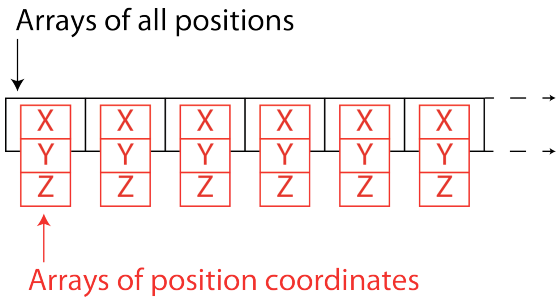
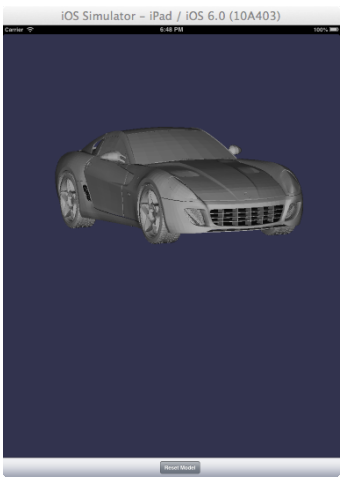


Figure 9. Showing how the vertex information is stored, in this case, the positional data.

With the OBJ loader complete, the ‘updateVertex’ method previously created is updated to allow for the newly created vertex information to be formatted in an interleaved manner ready to be uploaded to the OpenGL ES buffer. Because the vertex data was grouped into each position of an array, it is a matter of iterating through the arrays (which are all of identical length) and extracting the array of coordinates at each index and then taking the data from each of the sub arrays.

4.6 Effectiveness

With the OBJ loader able to obtain and use the vertex data to create three final arrays of vertex information in the correct order, and the ‘updateVertex’ method updated to make these ready for consumption by OpenGL ES, it should be checked whether it works with a number of different OBJ files and not just one specific test file. To test this, a number of different 3D models were found that spanned a range of different complexities. Some were simple objects with low face counts whilst others (such as the 3D head mentioned earlier) were much higher in complexity. To also test what would happen in the case of no texture file being present, models were tested that had no texture information associated with them. The above tests should show whether the three pieces of vertex information are being handled correct as the model should be built correct (positional data), be textured correctly (texture data) and in the case of no texture, the model normals should allow the lighting of the surface to be visible. The information from this testing can be seen in figure 10.



‘v’ lines	7940	350829	2351
‘vn’ lines	22338	353737	13860
‘vt’ lines	0	0	2828
Total faces	7446	681893	4620

Figure 10. Showing the results of different OBJ files being loaded and their respective information.

The results from these tests show that the OBJ reader can cope with a variety of different OBJ files being passed to it. However, it should be noted that all of these OBJ files made use of triangular faces as OpenGL ES does not support square faces (Buck, 2013). Indeed this is something that could be regarded as future work, recognising when a four sided face is being parsed and then creating two triangular faces from the single square face by splitting it in in half.

With a method of loading 3D models and their relevant texture data into the iOS environment, a method is now required to link the information being returned from the tracker to the model itself.

5. Keyframe Animation

This section refers primarily to the keyFrameMethods.h and keyFrameMethods.mm found in the appendix.

There is no inbuilt functionality within OpenGL ES to implement more advanced forms of animation without the use of either a commercial engine such as Unity 3D (which costs $400 for an iOS license) or for 3rd party libraries that are beyond the scope of this project

such as Cocos3d or Ogre3D (LaMarche, 2011). Because of this, a much simpler method of animating the facial model had to be found. The most basic form of animation is to store key positions of what the model should look like at a given time, and then work out how to get from one to the other (LaMarche, 2009). This technique is called keyframe animation and will form the basis for the mapping of movement from the user to the onscreen model. For the purposes of this project, only three keyframes have been implemented that will map to the user, these are shown in figure 11 and consist of a neutral frame, a raised brow frame and an open mouth frame.

Figure 11. Showing the three keyframes used in this project.

By loading the vertex data associated to each frame, it becomes possible to interpolate between the frames so as to give the appearance of a smooth transition. Interpolation allows for the positions in between the keyframes to be created by working out the differences for each vertex from the beginning and end key frames. By then incrementally adding the difference onto the start frame till the keyframe is reached, the illusion of animation is achieved. Figure 12 demonstrates this principle.

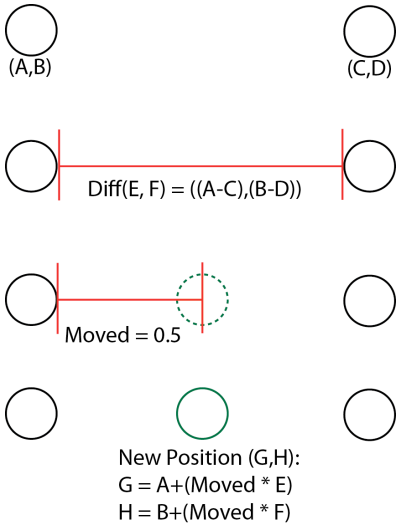


Figure 12. Showing the interpolation between two 2D vertex to a centre point

As this project is reliant on the user dynamically inputting the amount of movement, a method was required that would allow for the application to translate the position of the users various features and output it in such a way that it could be applied to the interpolation method. This would take the form, for example, of a value that would represent how open the users mouth was, as a value between 0 and 1. When used with the interpolation, it would be possible to multiply this value with the vertex difference between the two keyframes and end up with a new set of vertices that represent a frame in the middle of the animation. This is achieved by working out the distance from one vertex to another and then scaling it to within the 0 to 1 range.

Before the distance between two vertex is calculated, the choice of the vertex is of critical importance as the vertex chosen must represent the action being depicted accurately. An example of such a choice could be how far the user is opening their mouth, by picking the vertex in the underlying mesh that represent the centre of the top lip and the centre of the bottom lip, the distance between vertex will increase in line with the mouth openness. To obtain the distance between the vertex, the magnitude between the two vectors (consisting of X, Y and Z coordinates) that represent the vertex are calculated. To find the magnitude between two vectors, A and B, the equation in figure 13 is used.

$$Distance(A,B) = \sqrt{(A1 - B1)^2 + (A2 - B2)^2 + (A3 - B3)^2}$$

Figure 13. Showing the equation used to find the magnitude of two vectors.

This result gives an accurate representation of the users current facial state for that specific region, i.e. mouth openness, which when scaled and used for the interpolation allows for an accurate new set of vertex points to be created which when rendered will give a smooth transition between keyframes.

5.1 Optimising Keyframe Animation

In order to allow a number of keyframes to be used simultaneously an additional method was devised to allow for keyframes to be compared and where a difference in vertex coordinates occurred its index was stored in an array for later use. The interpolation methods were subsequently modified to allow for this difference array to be passed

through, allowing for the interpolation calculations to occur only on vertex that are known to have differences. This provides two key functions within the project, firstly it allows for the interpolation methods to become faster as they are not calculating differences for vertex that aren't moving. Secondly it allows the application to interpolate to multiple keyframes as long as there is no overlap in the difference arrays. Without this method it would only be possible to interpolate between whole keyframes, for example the neutral to raised brow movement would never feature any sort of mouth movement regardless of what the user was doing. By separating the interpolation methods to only work in different regions, transition between keyframes can occur on differing aspects of the 3D model in a simultaneous way (figure 14).

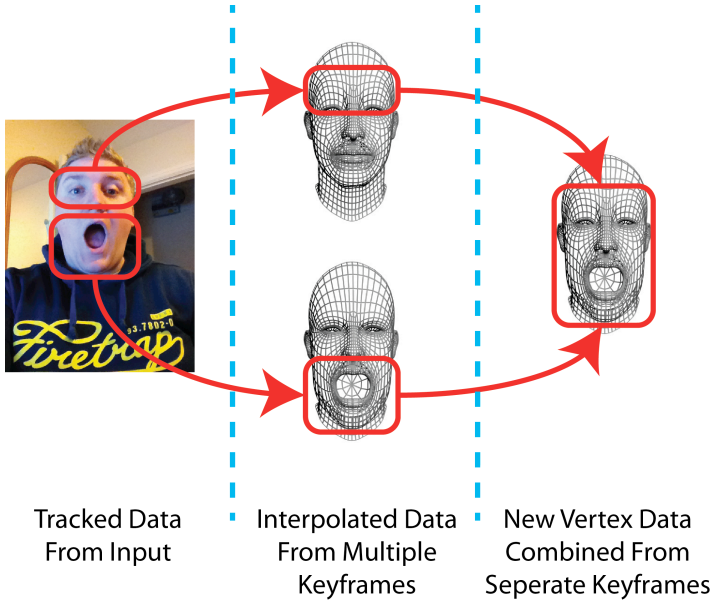


Figure 14. Showing how the difference in keyframe vertex between keyframes can be used to spilt the model into regions for simultaneous interpolation

5.2 Limitations

Whilst this technique works on a basic level, it does encounter a number of serious problems that reduce its effectiveness in scenarios such as this. The first is that it is a fairly intensive process, particularly when the limited resources that an embedded system such as an iOS device has when compared to a desktop computer are taken into account (Danihelka et al, 2011). When an analysis of the time is made in the method that takes the frame, tracks it and maps that data to the interpolated models, it is clear to see that the combined efforts of calculating the interpolation and updating the vertex data account for a

larger portion of the time than the tracker does. This is illustrated in figure 15 where the tracker takes 31.1% of the time, whereas the keyframe tracking (for the human head with three keyframes) takes a total of 50.1% of the time.

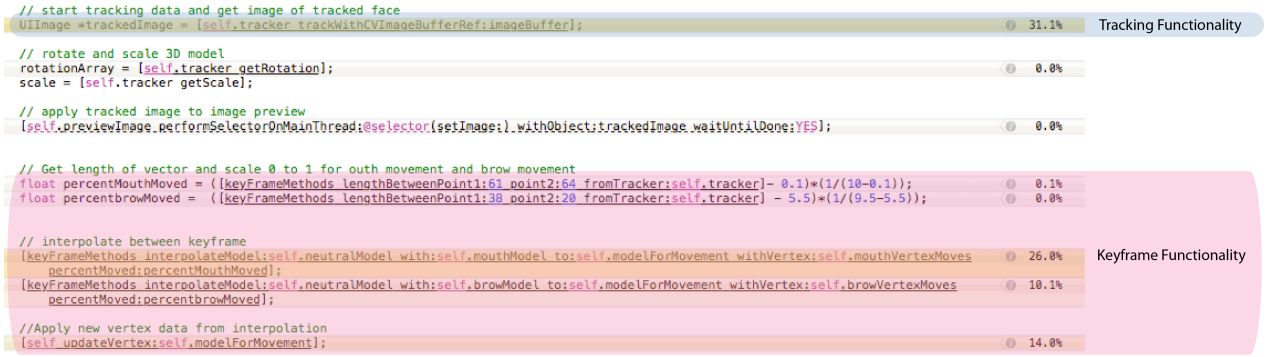


Figure 15. Showing the time comparison for tracking and interpolation functionality.

A second major limitation of this technique is that for every action that is required to be represented, a keyframe must be present. When attempting to replicate a human face this becomes a challenge as the number of keyframes required would become unwieldy and the need to be able to combine numerous keyframes as occurred above also becomes impractical. Ekman and Friesen (1978) identified 46 action units which can be used to describe a range of facial movements. For example, they suggest that happiness can be be conveyed by displaying action units 6 and 12, namely a cheek raise and a lip corner pull. Clearly 46 keyframes is too many to be able to realistically reproduce smoothly on an iOS device as the system currently slows down when only dealing with three. The alternative to keyframe animation is the use of skeletal animation, which allow a models mesh to be associated to bones, which when moved will cause the vertex to deform in a way that seems natural. As it is the vertex that are rendered, the underlying skeleton can be moved in any combination to encompass multiple actions by blending or adding skeletal poses (Marucchi-Foino, 2012). However, as mentioned previously this is not supported natively in OpenGL ES and would require a third party graphics engine to run. There is also the worry that, when combined with the computing needs of the face tracker, the skeletal animation would remain fairly slow and cumbersome.

Keyframe animation does however have a potential use within this project. If the idea of using a video realistic rendition of a human face is disregarded and an avatar put in its place then the system has the potential to work much more efficiently. An avatar in this

context could be something that has what would be recognised as facial features but its range of movements are severely limited, in this way, it could be thought of as a ventriloquist doll where only the mouth moves in sympathy with the user, disregarding any other movements. To test this theory, an avatar has also been implemented within the project that only has two keyframes, a neutral and an open mouthed one (figure 16). The interpolation that is required is much simpler as the model itself has fewer vertex points to interpolate to between and multiple keyframes do not have to be considered with each movement of the users face. This in turn results in a more responsive representation of the users movements, however, it is still to slow to be considered to be real-time.

Figure 16. Showing the only models required for a simple avatar

Because of the time it takes to compute the interpolation when working with good quality 3D representation, an additional option is to be put forward that involves splitting the processing of the 3D data to a separate device. For instance, when the iOS device was only concerned with tracking then the system was relatively fast and reliable and only became slowed down when the 3D elements were introduced. If the 3D aspect of the application could be implemented on an external device, then a balance could potentially be found. As the data to this point suggests that at present, there are too many limitations present on a mobile device to correctly track a users face whilst controlling and displaying a 3D model simultaneously, this is an option that will be further investigated.

6. OSC Study and Implementation

This section refers primarily to the oscMethods.h and oscMethods.m found in the appendix.

6.1 Initial Research

The use of Open Sound Control (OSC) as a means of communication between multimedia devices was briefly discussed in the interim report, however, as the core of the project was completed earlier than expected, and difficulties were encountered with animating models using OpenGl ES, OSC was explored further. The interim report talks about the basic means of transmission within OSC i.e. the use of packets and messages, but didn't go into depth about how an iOS device using OSC would communicate with another device on a more detailed level.

Using OSC within the application will allow for the information that can be acquired from the tracker to be passed over a network to any devices that are currently connected to it. OSC is mostly used for communication over IP networks (OSCulator, 2012) with an OSC packet being able to be represented with a datagram from a network protocol, primarily TCP and UDP (Wright, 2002). Once the OSC message has been received though, it either has to be utilised as is, or converted to another form of information so it can be consumed by another application. A problem with using OSC is that it is implementation specific, that is, as it can transmit such a wide variety of different information, it must be tailored to the application that is using it's data. A format such as MIDI though is a lot less flexible, as it uses a compact binary message format which only has a limited number of fields such as channel, velocity etc. (CNMAT, 2013) This rigidness though can also be seen as an advantage as it allows for a pre-defined format to be used which in turn allows for devices and applications to interoperate on a common level. Because of these differences and the established nature of MIDI, there are "thousands of software and hardware products that support MIDI, compared to dozens that support OSC" (MIDI, 2008). If it were possible to convert the OSC signals from the iOS device into MIDI, this would open up a range of possibilities for the use of the data from the tracker. Thankfully, there are a number of applications that can receive OSC signals and interpret them in someway with varying degrees of options and complexity. For this project two OSC receivers have been made use of, these are: OSCTestApp and Osculator. Both of these will fulfil different roles, with OSCTestApp being a simple way of seeing messages received and displayed as a way of

telling if the OSC application is working correctly, whilst OSCulator is much more complex and will allow for the OSC data to be mapped and viewed as MIDI information before it is passed to a final application for use.

6.2 Implementation

OSC is not a technology that is natively supported in the iOS environment with no official Apple frameworks being made to support it. There are however a small number of open source libraries that allow for OSC communication between devices to occur, namely, “BBOSC”, “VVOSC” and CocoaOSC. Upon further research however it would seem that a number of them were either not working with a new version of Xcode or incomplete implementations (Martin, 2013). This resulted in VVOSC being made the library of choice for this project as it has existed since 2008 so is well established and the amount of code required to be implemented in order to send a message is streamlined to reduce it to only a few lines (Trembl, 2011).

Once the library is downloaded and it must be compiled into a static library for use within the iOS environment. This is done using an automated script that comes packaged with the download and once compiled for use as a library within OSC, it can be imported as any other framework would be.

By defining the use of OSC within this application it becomes possible to narrow down the required functionality when creating an OSC method to use. As it is only desired that a message is sent over a network to some other device, the methods being created will need to be able to receive information. This results in only three VVOSC classes being required to be created, these are an OSCManager, an OSCOutPort and an OSCMessage. The OSCManager is at the core of sending information via OSC as it has the capability to create and delete inputs and outputs, and is also the main co-ordinator for data travelling in OSC format. The OSCOutPort is created with the OSCManger and is assigned an address and port number to send information to across the network. Finally the OSCMessage contains the values that are to be sent as well as an address path that they are being sent to (VVOSC, 2013).

To begin with ascertaining that VVOSC would indeed work as expected, a simple method was built that begins by initialising the three required objects with their required information (i.e. network address, port number and message address). To add a value to a message

their are a number of class methods that allow for a specific type to be added, for example, to add an integer to an OSCMessage object called msg, the following would be used: “[msg addInt:intToSend]”. With the desired value in the message, it can be sent by passing it to the OSCOutPort object with a method called “sendThisMessage”.

To test if the VVOSC methods are functioning correctly within the application, a test was conducted that would add an increasing integer value to a message and send it every time a frame was processed. By performing this test, it should indicate that VVOSC works correctly and that it works directly from the method using the tracked information. OSCTestApp will be used to visualise the messages being sent over the network and it’s hoped that a continuous stream of incrementing values will be received. When this test was run, the information streamed across as expected indicating that VVOSC was functioning correctly. An example of the information received by OSCTestApp can be seen in figure 17.

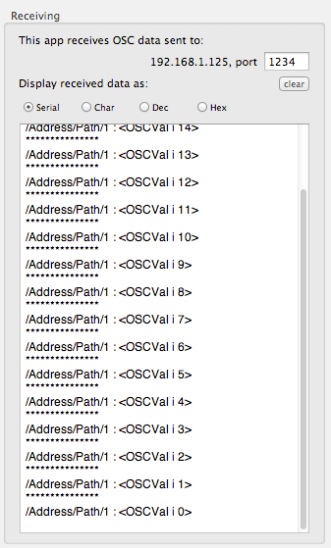


Figure 17. Showing the incremental test data received by OSCTestApp

Knowing that VVOSC is working correctly means that the task of linking the tracked information to the OSC output can be undertaken. Because the intention is for the data being sent to act as control signals, it would seem sensible to process the data coming from the tracker in the same way as was previously done with the OpenGL ES implementation. By again calculating the distance between specific vertex found on the underlying 66 vertex mesh, suitable control information can be gathered and sent to the receiving device. With a method already in place to calculate the distance between two

vertex, it becomes about how to best package the data for transmission. Because each piece of data will have the potential to control a specific control point on a separate device, each message sent from the iOS device will contain only one piece of control data and will be labeled according to what it correspond to. The labelling can be done by assigning it an address to be sent to that corresponds to role it has, for example, a message could be created with the address “/facialMovement/mouth” to indicate that the data being sent was related to the mouth. One important aspect that should be noted is that in the OpenGL ES implementation, the distance between vertex was scaled into a range of 0 to 1, however, this is not necessary as the information can be sent raw to OSCulator which can then be set to perform the scaling instead. For the purposes of this project, six pieces of information will be sent from the iOS device to OSCulator, these are: brow movement, mouth width, mouth height and the three axis of head rotation. When sent to OSCulator they are scaled to be between a value of 0 to 127 which is the size of the MIDI byte available for carrying data (Li et al, 2004). Figure 18 shows the addresses attached to messages that OSCulator is successfully receiving data from.

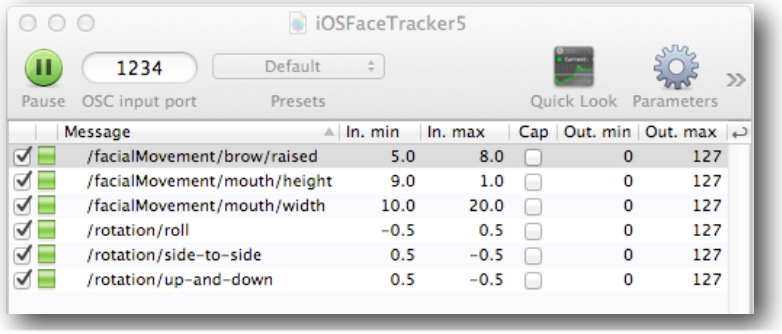


Figure 18. Showing the address attached to messages that are being successfully received by OSCulator.

6.3 Effectiveness and Usefulness

With the ability to now use the tracking data from the iOS device as MIDI control data, a method of transposing that information onto a facial model is required. Because the constraints of the iOS device and OpenGL ES are removed, it becomes possible to make use of much more powerful applications. The textured facial model used for the OpenGL ES implementation originated as a rigged 3ds Max model, this means that the model can be deformed in specific ways according to predefined target objects (Polygon, 2012). This is useful as if a method of linking up the data from OSCulator to the control points on the rigged facial model were possible then the model could be controlled by the users movements directly from the iOS device.

3ds Max has a feature that allows an object (such as the movement handle for a part of the rigged face) to be controlled by an external source (including MIDI), this system is called motion capture control (Autodesk, 2013). By taking the MIDI information being outputted from OSCulator and combining it with this motion capture system an accurate link can be established between the users movements and the 3d model residing in 3ds Max. This new method of controlling the 3D facial model provides a much smoother experience, with a much better mapping of user movement when compared to the OpenGL ES implementation. Figure 19 shows an example of the tracker and 3ds motion capture controls working together within the 3ds Max environment.

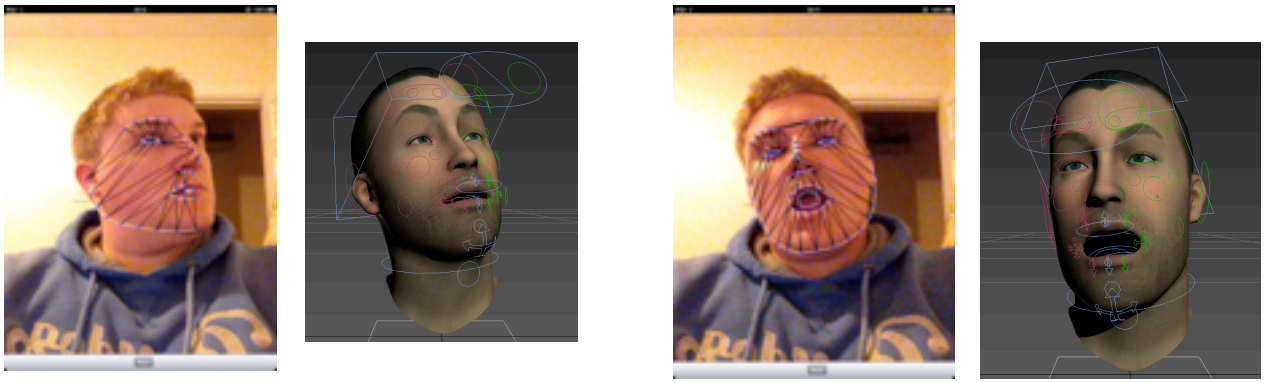


Figure 19. Showing the 3ds Max motion capture controls working with the iOS device tracker

When compared to the OpenGL ES implementation, which involved more processing power being spent on the keyframe animation than the actual tracking, the OSC implementation is much more efficient. When the CPU timings are taken as with the previous implementations, this decrease in computational power is clearly shown. In this implementation the tracker takes 98.5% of the processing time for each frame whilst the entire process of calculating vertex distance and sending the messages takes only 1.1% of the time. As it's known from our initial tests that the tracker takes 129.27 milliseconds on the medium quality preset (which this uses) then the amount of time spent on the OSC messaging will be tiny in comparison. Figure 20 shows the breakdown in computation times for each frame.

Figure 20. Showing the percentage of time spent the CPU spends on each function for each frame.

7. Testing

Up to this stage only white box testing has occurred for each major implementation by analysing how each change to the functionality of the device affects the amount of time the CPU spends within each method. No black box testing has been attempted on the system as a whole and therefore an appraisal will not be possible until such testing is completed. This section will therefore aim to test and analyse the final two system approaches as a whole before comparing them.

With two contrasting paths taken to implementing a system that can track a users facial movements and represent it as a 3D model, both approaches need to be compared to one another to allow for a critical evaluation. This will be useful as both approaches, whilst sharing a common tracking function, are inherently different in the approach they take. The OpenGL ES implementation attempts to do everything by itself from tracking to displaying the model whilst the OSC approach attempts to offer a better experience by offloading everything related to the 3D element of the project to a separate device.

For each implementation the following will be tested:

- Test case #1: How well is the users face tracked
- Test case #2: Does the 3D Model mimic the users movements
- Test case #3: Can the application run for a period of time without crashing

Test case #1 will test how well the tracker copes with the user moving their face around. To do this the application will be loaded and a users face will be placed in the centre of the screen, the face will then move up and down speeding up as it goes. This test will indicate how well the face is initially tracked as well as how it subsequently copes with faster changes in position.

Test case #2 will test how well the 3D model responds to the users movements. To do this the application will be loaded and a users face will be in the centre of the screen. The user will then look up and down, side to side, then open and shut their mouth followed by raising of the eyebrows. To complete the eyebrows will be raised with an open mouth whilst moving the face side to side. This test will show how well the movements obtained from the tracker are being mapped to the 3D model and how well it copes with multiple variations in the model.

Test case #3 will test how robust the application to see if memory leaks or any general errors appear. To perform this test the iOS device will be run from xCode to allow the console to display any warnings if necessary then the application will be left for 5 minutes with no face. When there is no face present the tracker spends much more time trying to find a face than usual thereby excerpting more computational power than usual. Once this test is complete, the application will be run for 5 minutes with a face being tracked and the 3D model displayed in the applicable way.

Videos will be recorded of test cases #1 and #2.

7.1 OpenGL ES

7.1.1 Test Case #1

The outcome from this test is fairly conclusive. As the users facial movements speed up, the ability for the tracker to follow the movements decreases. As discussed in the CPU analysis of the OpenGL ES implementation, the process involved in the keyframe animation is incredibly intensive compared to everything else that is happening and this results in a knock on effect through to the user. Because more time is being spent processing each frame it results in the flow of frames being sent from the camera to be slowed down dramatically. This is a result of the AVFoundation framework recognising that processing is still being done on a previous frame and simply not sending any more until

that processing is complete (Apple, 2013). As a result the frames that are being sent to the tracker become less cohesive when the user begins to move their face around. For example, if the frames are only being sent every half a second then in this time the user could have moved their face to the other side of the screen. The problem with this is that there is no mechanism to tell the tracker that this is happening and it will be expecting the face position to have only changed slightly. This results in a number of mistakes occurring due to the way the Active Shape Model at the core of the tracker is working. Each point that is tracked has a region that it is able to search in to detect movements. When a users feature moves, the tracker searches along this region for what it thinks it should be positioned on and when it finds it, the point is moved to the new position (Cootes et al, 1995). However, the region that it has to search along is limited in size so that it is only likely to encounter the point it's looking for rather than get a false positive. What's happening in the OpenGL ES implementation is that when a frame is sent to the tracker the points are found as normal. However, when the frame rate drops, the next frame that is received has the potential to have moved out of the range of each points search region. This results in the user moving but the tracked points not moving at the same time. This is a large problem with this implementation as it severely affects the usability of the system. By losing the ability to track rapid movements, the user is forced to either move slowly or risk having the 3D model loose the connection between their facial movements and itself. This test can be seen in video x.

7.1.2 Test Case #2

This test served to highlight how interconnected the tracker was to the 3D implementation, with the failure shown in test case #1 severely affecting the performance of the 3D model to mimic the users facial movements. Also apparent is a cracking effect when the model is animating between keyframes, this is potentially due to the interpolation being unable to keep up with the OpenGL ES rendering loop and half interpolated results being displayed. Besides these problems the mapping of the the users movements does function to some extent, it's just too slow to be considered realistic given the above problems and the limited number of keyframes available for interpolation. This test can be seen in video x.

7.1.3 Test Case #3

The application was run for 5 minutes with and without a face to track and there were no errors or warnings present at any point.

7.2 OSC Implementation

7.2.1 Test Case #1

In this test the tracker performed well and only began to fail when the users head movements became fast enough to induce motion blur. As the tracker is the only process on the device that requires a fairly large amount of processing power, and as seen previously, the OSC methods take virtually nothing in comparison, this means that the iOS device does not have to share its processing power with other large processes running at the same time. This results in the frames being processed in a timely manner allowing the camera to keep up a high frame rate output and the tracker to have the power required to perform its function. This implementation can be seen in video x.

7.2.2 Test Case #2

Because the tracked data is being sent over the network in realtime the 3ds model that moves with the users movement is very responsive with virtually no lag. By having the ability to transmit the tracking data to a separate device that is dedicated to working with the 3D information, the actual rendering and animation involved becomes markedly cleaner and flowing. Also apparent are the more advanced animation techniques available outside of the iOS environment such as how the neck remains static whilst the head moves relative to it. This implementation can be seen in video x.

7.2.3 Test Case #3

The application was run for 5 minutes with and without a face to track and there were no errors or warnings present at any point.

8. Future Work

Whilst additional features have been implemented within this project that were not deemed to be requirements in the interim report, there are a wide range of possibilities for future work based on the system so far. This section will attempt to outline five potential future areas of action.

The first area that could be worked upon is the optimisation and integration of Saragih's tracker into a fully native iOS library. At present it remains in C++ and uses OpenCV data structures extensively, however, the use of OpenCV is not as efficient as using iOS’s own

data structures. Work could be conducted to port the trackers methods over to an iOS environment. The potential problem with this is that a number of highly specialised OpenCV methods are implemented that have no direct equivalence in the iOS frameworks. This could require that OpenCV still be used, but in a much more limited manner than previously employed. Functions such as the face detection could be swapped in favour of ones available natively in iOS as discussed previously.

The second area of future investigation could be based around working with more advanced animation within the iOS environment. This could work in a number of ways, for instance it could be that the commercial engine is used or that the 3rd party libraries are researched and implemented where possible. Alternatively, if enough time and expertise were available, advance animation techniques could be built directly using OpenGL ES rather than extrapolating many of the functionality to GLKit as was done in this project. The use of advanced animation techniques could also improve the performance of the tracker as the current keyframe animation methods are slightly reliant on brute force that could be substituted for more elegant methods.

The third area that could built upon is the use of multiple iOS devices to communicate over a network to each other. It has already been shown that it is possible to both send OSC data and display 3D models on iOS devices, so it is potentially just a case of being able to receive the OSC data and apply it in the necessary way. This could open up the ability for an iOS device to track a users face, but display somebody else's facial movements so that a conversation via avatar could be had. The only new area of research would be the potential to also transmit audio over the same network that OSC is using so that a full conversation could be had with another user.

A fourth aspect ready for extension from the origins in this project is the use of more detailed tracking information and it's subsequent 3D mapping. Currently the maximum number of mapped data points is six, however, by calculating more detailed movements a wider range of facial representations could be explored. For example at present the OSC version only allows a mouth to be opened or closed, or slightly widened or tightened. By including more information it is possible that smiles or frowns could be recreated, however, this also lies partly with the 3D model and it's ability to be animated in such ways.

Finally, an incredibly open ended area available to be worked on is the extended use of OSC and MIDI as methods of controlling other devices or applications. This offers an almost unlimited scope for interesting and creative ideas, from simple uses such as controlling sounds to more complex ideas such as controlling robots who hit drums depending on the MIDI signals being sent (Laursen, 2011). This could work with the iOS application in a number of ways, for example if it were possible to detect whether a user were smiling or frowning, an OSC signal could be sent telling a sequencer to play major chords or minor chords depending on the expression. Or it could be as simple as using the height of the users open mouth to determining how loud something will play.

9. Conclusions

This project set out with the aim of utilising a method of tracking a users facial features within an iOS environment and then mapping the information to a 3D model that would move in sympathy with the users facial movements. In the initial plan a number of different options were proposed for tracking the users face but by the interim report this had been narrowed down to a face tracker created by Jason Saragih. This tracker formed the foundation that the remainder of this project was built on and in the majority of cases performed well. The only times that the tracker failed to perform were primarily down to the features that had been added as part of this project that either slowed it down or didn't give it the correct information to begin with. To try and ensure that the tracker provided the best foundation possible, constant analysis was conducted and optimisations made where possible.

With the tracker known to be reliable and the process of sending and receiving information from it created, the need to display a 3D model was fulfilled using OpenGL ES within the iOS environment. With this in place the information coming from the tracker was successfully linked to a basic 3D representation of the face by using the underlying 3D mesh found in the tracker as a guide to where the 3D points should be rendered. However, this was a fairly rudimentary method of representing a human face in 3D and, although, it fulfilled the core aims of the project it was not as refined as it could be. To proceed from the basic representation to a more advance facial model required the ability to load pre-made models into the iOS environment. Unfortunately there is no built in function that

allows this so a class was created that could read in 3D data contained in OBJ files and store the values for future use within OpenGL ES.

Following the ability to load 3D models into the iOS environment, a method was required of animating them in sympathy with the users motions. This was eventually done by using keyframes to interpolate between different pre-rendered expression. However this gave a very limited range of movements and in testing it was revealed that the processing power required to perform the interpolation methods was enough to make the tracker unresponsive. This essentially signalled the end of the desire to perform both the tracking and 3D model representation on a single iOS device, instead two devices would now be used in order to separate the areas of functionality (tracking and 3D rendering) to allow for each function to perform at its maximum ability.

Research into OSC took place as a method of allowing the iOS device to communicate over a network to another device. This allowed the data from the tracker to be transported and captured by OSCulator that would then convert it to MIDI for future use. By the linking the MIDI from OSCulator to motion capture controllers in 3ds Max the ability to control a 3D model that had been rigged specifically for that task became possible. By removing the OpenGL ES aspect to the system and instead replacing it with OSC functionality, the performance of the tracker was greatly improved. By also allowing a device to be purely dedicated to performing the 3D tasks required, the fluidity of the motions that were able to be mapped from the user to the 3D model greatly increased.

When this project began there was some amount of uncertainty about how much would be achievable. However, the project has progressed well from the beginning and at a number of stages has been ahead of the proposed schedule. This meant that once the core objective of creating a basic facial representation that was linked to the tracker was achieved, a more advanced approach could be taken to displaying and controlling the 3D models. At this point however the project diverged into two separate implementations each with their own capabilities. In the end however, when both were tested and evaluated it became clear to see that the option of using the iOS device solely as a tracker and an external device as the main 3D model handler was the best option. not only did it allow for the tracker to function correctly and quickly, but the quality of the animation on the 3D model was vastly improved. The OpenGL ES implementation on the other had failed

primarily down to lack of processing power available to it, and it would be interesting to see how this would function with improved hardware in the iOS device.

10. Reflection on Learning

This section of the report will aim to understand and reflect upon the impact of my actions on the project. This will be done in the hope that faults identified here can be addressed so that similar situations and problems encountered in the the future can be dealt with in a more prepared manner. To analyse each reflective point, first the fault will be identified, then its consequence on the project will be noted before finally finding a solution to prevent it from happening in future projects. Unlike the rest of the report, this section will be written in the first person.

This project was the largest project I have undertaken in terms of both the scope of what was to be achieved as well as the number of different tools that were required to be learnt about before it could even be begun. Because the project was larger in scope than anything in my previous experience it was difficult to conceptualise in my mind how it would all play out and fit in with the other pieces of university work that had to be completed alongside this project. When it came to completing a long term plan for the project, in hindsight I may have been over optimistic in estimating how quickly external pieces of work could be achieved. This resulted in periods of time where it was not possible to work on the project, as time was being diverted to finishing up coursework that had taken longer than first anticipated. In the future a more robust schedule that incorporates the worst case rather than the best case scenarios would allow for a much more feasible plan. By building in extra time, it allows for movement within the plan rather than immediately falling behind when something is forced to run over.

A problem that also developed over the course of the project was an all or nothing approach. This meant that I would either dedicate all my resources to the project or none at all resulting in progress on the project being made either slowly or incredibly fast. Because a constant progress was not being maintained it often became difficult to know quite what was going on as there could be a long lull (up to a week and a half) between work on the project resulting in my thoughts being removed from the implementations. What would be a much more desirable scenario would be one where a constant stream of

work could be achieved, for example dedicating 2-3 hours to the project every other day to allow work to be accomplished in smaller chunks but in a way that keeps it fresh in my memory. This would allow for a much more cohesive implementation experience throughout the course of the project.

The final area that could see room for improvement would be that of making sure I actually complete a section before moving on. This problem became apparent with the multiple implementations that built upon one another as I would occasionally have a piece of code that functioned correctly but had some fault that at the time did not impact on the functionality. However, once an implementation further down the line had built upon it the fault became apparent and made the process more difficult than it should have been. To remedy this would have been a matter of simply keeping a log of all the bugs and flaws with the project as I go and making sure that they are all looked into before moving on to another step. This would allow for problems to be ironed out before they have the opportunity to become to entrenched within the code.

Having recognised these faults though I am incredibly happy with the result. At the beginning of the project I had never programmed in Objective-C or C and had never used OpenCV or OpenGL. The final implementations that were created far outstrip anything I thought I would be capable of achieving at the start of the project and hopefully with the points addressed above I will much more prepared and capable for whatever may come next.

11. References

Apple Inc. 2011. iOS Developer Library [Online]
Available at: http://developer.apple.com/library/ios/navigation/
[Accessed: 24th April]

Autodesk. 2013. 3ds Max *Documentation* [Online]
Available at: http://docs.autodesk.com/3DSMAX/15/ENU/3ds-Max-Help/
[Accessed: 30th April]

Buck, E M. 2013. *Learning OpenGL ES for iOS*. Addison-Wesley.

Cepeda, R. 2012. *mtl2opengl* [Online]
Available at: https://github.com/ricardo-rendoncepeda/mtl2opengl
[Accessed: 2nd May]

CNMAT: Center For New Music and Audio Technology. 2013. *What is the difference between OSC and MIDI?* [Online]
Available at: http://opensoundcontrol.org/what-difference-between-osc-and-midi
[Accessed: 18th April]

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

Danihelka,J. Hak, R. Kencl, L. Zara, J. 2011. *3D Talking-Head Interface to Voice-Interactive Services on Mobile Phones.* International Journal of Mobile Human Computer Interaction. 3(2) pp.50-64

Ekman, P. Friesen, W. 1979. *FACS - Facial Action Coding System* [Online]
Available at: http://www.cs.cmu.edu/~face/facs.htm
[Accessed: 23rd April]

LaMarche, J. 2009. *Fundamentals of Animation and Keyframe Animation* [Online]
Available at: http://iphonedevelopment.blogspot.co.uk/2009/12/opengl-es-from-ground-up-part-9a.html
[Accessed: 22nd April]

LaMarche, J. 2011. *3D Game and Graphics Engines* [Online]
Available at: http://iphonedevelopment.blogspot.co.uk/2011/05/3d-game-and-graphics-engines.html
[Accessed: 26th April]

Laursen, T. 2011. *MIDI Controlled Drum Machine Robots* [Online]
Available at: http://laughingsquid.com/midi-controlled-drum-machine-robots-by-tim-laursen/
[Accessed: 1st May]

Li, Z. Drew, M. 2004. *Fundamentals of Multimedia*. Pearson Education.

Martin, C. 2013. *Finding a Good OSC Library For iOS* [Online]
Available at: http://charlesmartin.com.au/blog/2013/3/26/finding-a-good-osc-library-for-ios
[Accessed: 30th April]

Marucchi-Foino, R. 2012. *Game and Graphics Programming for iOS and Android with OpenGL ES 2.0.* Wiley.

McDonald, K. 2012. *Kyle McDonald Explains FaceTracker* [Online].
Available at: http://makematics.com/research/facetracker
[Accessed: 30th April]

McDonald, K. 2013. *FaceOSC Source Code* [Online]
Available at: https://github.com/kylemcdonald/ofxFaceTracker/tree/master/FaceOSC
[Accessed: 29th April]

MIDI. 2008. *Comparison of MIDI and OSC* [Online]
Available at: http://www.midi.org/aboutmidi/midi-osc.php
[Accessed: 18th April]

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

OpenCV. 2010. *OpenCV Documentation* [Online]
Available at: http://opencv.willowgarage.com/documentation/
[Accessed: 30th April]

OpenGL. 2013. *OpenGL Reference Pages* [Online]
Available at: http://www.opengl.org/sdk/docs/man/xhtml/
[Accessed: 24th April]

OSCulator. 2012. *OSCulator 2.12 Manual* [Online]
Available at: http://dl.osculator.net/doc/OSCulator+2.11+Manual.pdf
[Accessed: 18th April]

Ploygon. 2012. *Character Rigging* [Online]
Available at: http://www.polygonblog.com/character-rigging/
[Accessed: 30th April]

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

Stevenson, S. 2010. *Cocoa and Objective-C: Up and Running*. O'Reilly

Trembl. 2011. *OSC to and from the iPhone with VVOSC* [Online]
Available at: http://www.trembl.org/codec/532/
[Accessed: 30th April]

VVOSC. 2013. *VVOSC Documentation* [Online]
Available at: http://www.vidvox.net/rays_oddsnends/vvosc_doc/index.html
[Accessed 30th April]

Wavefront. 2013. *WaveFront Object File Format* [Online]
Available at: http://people.cs.clemson.edu/~dhouse/courses/405/docs/brief-obj-file-format.html
[Accessed: 27th April]

Wenderlich, R. 2011. *OpenGL ES 2.0 for iPhone Tutorial* [Online]
Available at: http://www.raywenderlich.com/3664/opengl-es-2-0-for-iphone-tutorial
[Accessed: 24th April]

Wright, M. 2002. *The Open Sound Control 1.0 Specification* [Online]
Available at: http://opensoundcontrol.org/spec-1_0
[Accessed: 18th April]

12. Appendix

trackerWrapper.h
trackerWrapper.mm

```
//
//  trackerWrapper.h
//  iOSFaceTracker 2
//
//  Created by Tom Hartley on 01/12/2012.
//  Copyright (c) 2012 Tom Hartley. All rights reserved.
//

#import <Foundation/Foundation.h>
#import <opencv2/opencv.hpp>
#import "Tracker.h"
#import "imageConversion.h"
#import <AVFoundation/AVFoundation.h>


@interface trackerWrapper : NSObject


-(void)initialiseModel;
-(void)initialiseValues;
-(void)resetModel;
-(UIImage *)trackWithImage:(UIImage *)im;
-(UIImage *)trackWithCvMat:(cv::Mat)im;
-(UIImage *)trackWithCVImageBufferRef:(CVImageBufferRef)imageBuffer;
-(NSMutableArray *)getRotation;
-(double)getScale;
-(NSArray *) get3dMesh;
-(NSArray *)getSpecificPoint:(int)point;

@end
```

```
//
//  trackerWrapper.m
//  iOSFaceTracker 2
//
//  Created by Tom Hartley on 01/12/2012.
//  Copyright (c) 2012 Tom Hartley. All rights reserved.
//

#import "trackerWrapper.h"




@implementation trackerWrapper {

    int switchVal;

    FACETRACKER::Tracker model;
    cv::Mat tri;
    cv::Mat con;

    std::vector<int> wSize1;
    std::vector<int> wSize2;
    std::vector<int> wSize;

    bool fcheck;
    double scale;
    int fpd;
    bool show;

    int nIter;
    double clamp,fTol;

    cv::Mat gray,im;

    bool failed;

    imageConversion *imageConverter;
}




-(void)initialiseModel
{

    NSString *modelPath = [[NSBundle mainBundle] pathForResource:@"face2"
        ofType:@"tracker"];
    NSString *triPath = [[NSBundle mainBundle] pathForResource:@"face"
        ofType:@"tri"];
    NSString *conPath = [[NSBundle mainBundle] pathForResource:@"face"
        ofType:@"con"];

    const char *modelPathString = [modelPath cStringUsingEncoding:
        NSASCIIStringEncoding];
    const char *triPathString = [triPath cStringUsingEncoding:
        NSASCIIStringEncoding];
```

```
    const char *conPathString = [conPath cStringUsingEncoding:
        NSASCIIStringEncoding];

    model.Load(modelPathString);
    tri=FACETRACKER::IO::LoadTri(triPathString);
    con=FACETRACKER::IO::LoadCon(conPathString);

    imageConverter = [[imageConversion alloc] init];

}


-(void)initialiseValues
{
    wSize1.resize(1);
    wSize2.resize(3);
    wSize1[0] = 7;
    wSize2[0] = 11;
    wSize2[1] = 9;
    wSize2[2] = 7;

    fcheck = false;
    scale = 1;
    fpd = -1;
    show = true;
    nIter = 5;
    clamp=3;
    fTol=0.01;
    failed = true;
}




-(void) draw
{
    cv::Mat shape = model._shape;
    cv::Mat visi = model._clm._visi[model._clm.GetViewIdx()];

    int i,n = shape.rows/2; cv::Point p1,p2; cv::Scalar c;

    //draw triangulation
    c = CV_RGB(0,0,0);
    for(i = 0; i < tri.rows; i++){
        if(visi.at<int>(tri.at<int>(i,0),0) == 0 ||
           visi.at<int>(tri.at<int>(i,1),0) == 0 ||
           visi.at<int>(tri.at<int>(i,2),0) == 0)continue;
        p1 = cv::Point(shape.at<double>(tri.at<int>(i,0),0),
                       shape.at<double>(tri.at<int>(i,0)+n,0));
        p2 = cv::Point(shape.at<double>(tri.at<int>(i,1),0),
                       shape.at<double>(tri.at<int>(i,1)+n,0));
        cv::line(im,p1,p2,c);
        p1 = cv::Point(shape.at<double>(tri.at<int>(i,0),0),
                       shape.at<double>(tri.at<int>(i,0)+n,0));
        p2 = cv::Point(shape.at<double>(tri.at<int>(i,2),0),
                       shape.at<double>(tri.at<int>(i,2)+n,0));
        cv::line(im,p1,p2,c);
        p1 = cv::Point(shape.at<double>(tri.at<int>(i,2),0),
```

```
                        shape.at<double>(tri.at<int>(i,2)+n,0));
            p2 = cv::Point(shape.at<double>(tri.at<int>(i,1),0),
                        shape.at<double>(tri.at<int>(i,1)+n,0));
            cv::line(im,p1,p2,c);
        }

        //draw connections
        c = CV_RGB(255,255,255);
        for(i = 0; i < con.cols; i++){
            if(visi.at<int>(con.at<int>(0,i),0) == 0 ||
               visi.at<int>(con.at<int>(1,i),0) == 0)continue;
            p1 = cv::Point(shape.at<double>(con.at<int>(0,i),0),
                        shape.at<double>(con.at<int>(0,i)+n,0));
            p2 = cv::Point(shape.at<double>(con.at<int>(1,i),0),
                        shape.at<double>(con.at<int>(1,i)+n,0));
            cv::line(im,p1,p2,c,1);
        }

        //draw points
        for(i = 0; i < n; i++){
            if(visi.at<int>(i,0) == 0)continue;
            p1 = cv::Point(shape.at<double>(i,0),shape.at<double>(i+n,0));
            c = CV_RGB(255,0,0); cv::circle(im,p1,2,c);
        }


    }

    -(void)track
    {

        if(failed) {
            wSize = wSize2;
        } else {
            wSize = wSize1;
        }

        if(model.Track(gray,wSize,fpd,nIter,clamp,fTol,fcheck) == 0) {

            [self draw];
            failed = false;

        }else{

            [self resetModel];
            failed = true;
        }

    }

    -(void)resetModel
    {
        model.FrameReset();
    }

    -(UIImage *)trackWithImage:(UIImage *)image
    {
        static cv::Mat frame;
        frame = [imageConverter cvMatWithImage:image];
```

```
    if(scale == 1)im = frame;
    else cv::resize(frame,im,cv::Size(scale*frame.cols,scale*frame.rows));
    cv::flip(im,im,1);
    cv::cvtColor(im,gray,CV_BGR2GRAY);

    [self track];



    return [imageConverter UIImageFromMat:im];

}

-(UIImage *)trackWithCvMat:(cv::Mat)frame
{
    //frame =  image;

    if(scale == 1)im = frame;
    else cv::resize(frame,im,cv::Size(scale*frame.cols,scale*frame.rows));
    cv::flip(im,im,1);
    cv::cvtColor(im,gray,CV_BGR2GRAY);

    [self track];

    return [imageConverter UIImageFromMat:im];

}

-(UIImage *)trackWithCVImageBufferRef:(CVImageBufferRef)imageBuffer
{
    CVPixelBufferLockBaseAddress(imageBuffer,0);

    /*Get information about the image*/
    uint8_t *baseAddress = (uint8_t *)CVPixelBufferGetBaseAddress(imageBuffer)
        ;
    size_t width = CVPixelBufferGetWidth(imageBuffer);
    size_t height = CVPixelBufferGetHeight(imageBuffer);


    cv::Mat frame(height, width, CV_8UC4, (void*)baseAddress);

    // Make image the correct orientation for upwards iPad
    cv::Mat dst;
    cv::transpose(frame, dst);
    cv::flip(dst, dst, 1);

    // Convert from native BGRA to RGBA
    cvtColor(dst,frame,CV_BGRA2RGBA);


    if(scale == 1)im = frame;
    else cv::resize(frame,im,cv::Size(scale*frame.cols,scale*frame.rows));
    cv::flip(im,im,1);
    cv::cvtColor(im,gray,CV_BGR2GRAY);


    [self track];
```

```
    CVPixelBufferUnlockBaseAddress(imageBuffer,0);

    return [imageConverter UIImageFromMat:im];
}



- (NSMutableArray *)getRotation
{
    cv::Mat pose = model._clm._pglobl;

    NSMutableArray *rotationArray = [[NSMutableArray alloc] initWithCapacity:3
        ];

    for (int i = 1; i<4; i++) {

        [rotationArray addObject:[NSNumber numberWithDouble:pose.at<double>(i,
            0)]];
    }

    return rotationArray;

}

-(double)getScale
{
    CvMat pose = model._clm._pglobl;

    return cvGetReal2D(&pose,0,0) ;
}


-(NSArray *) get3dMesh{
    static cv::Mat mesh;


    mesh.create(model._clm._pdm._M.rows,1,CV_64F);

    mesh = model._clm._pdm._M + model._clm._pdm._V*model._clm._plocal; // mean
        + variation * weights;

    int n = mesh.rows/3;

    NSMutableArray *meshArray = [[NSMutableArray alloc] initWithCapacity:n];


    for (int i = 0; i<n; i++) {
        [meshArray addObject:[self getSpecificPoint:i]];
    }

    return meshArray;
}

-(NSArray *)getSpecificPoint:(int)point
{

    static cv::Mat mesh;

    mesh.create(model._clm._pdm._M.rows,1,CV_64F);
```

```
    mesh = model._clm._pdm._M + model._clm._pdm._V*model._clm._plocal; // mean
        + variation * weights;

    int stride = mesh.rows/3;

    NSNumber *x = [NSNumber numberWithDouble:mesh.at<double>(point, 0)];
    NSNumber *y = [NSNumber numberWithDouble:-(mesh.at<double>(point+stride, 0
        ))]; // Made negative to account for reverse y-axis in OpenGl
    NSNumber *z = [NSNumber numberWithDouble:(-mesh.at<double>(point+stride+
        stride, 0))];

    return @[x,y,z];
}


@end
```

iOSViewController3.h
iOSViewController3.mm

```
//
//  HelloGLKitViewController.h
//  iosFaceTracker3
//
//  Created by Tom Hartley on 30/01/2013.
//  Copyright (c) 2013 Tom Hartley. All rights reserved.
//

#import <GLKit/GLKit.h>
#import "trackerWrapper.h"
#import <AVFoundation/AVFoundation.h>
#import "Timer.h"

@interface HelloGLKitViewController : GLKViewController <
    AVCaptureVideoDataOutputSampleBufferDelegate>

- (IBAction)resetModelToolBarButton:(UIBarButtonItem *)sender;
@end
```

```
//
//  HelloGLKitViewController.m
//  iosFaceTracker3
//
//  Created by Tom Hartley on 30/01/2013.
//  Copyright (c) 2013 Tom Hartley. All rights reserved.
//

#import "HelloGLKitViewController.h"


typedef struct {
    float Position[3];
    float Colour[4];
} Vertex;

Vertex Vertices[] = {
    // Front
    {{-21.456487,5.801546,-71.543480}, {0.5,0.5,0.5,1}},
    {{-20.917742,0.443765,-71.123396}, {0.5,0.5,0.5,1}},
    {{-20.075825,-4.887262,-70.786568}, {0.5,0.5,0.5,1}},
    {{-18.833412,-10.013067,-68.500374}, {0.5,0.5,0.5,1}},
    {{-16.776484,-14.642291,-63.490461}, {0.5,0.5,0.5,1}},
    {{-13.319503,-18.299496,-57.938732}, {0.5,0.5,0.5,1}},
    {{-8.822831,-20.823048,-53.396860}, {0.5,0.5,0.5,1}},
    {{-3.782325,-22.275233,-49.158219}, {0.5,0.5,0.5,1}},
    {{1.521818,-22.322379,-45.394461}, {0.5,0.5,0.5,1}},
    {{6.774960,-21.472490,-49.291942}, {0.5,0.5,0.5,1}},
    {{11.479597,-19.195781,-53.596112}, {0.5,0.5,0.5,1}},
    {{15.484089,-15.961767,-58.171004}, {0.5,0.5,0.5,1}},
    {{18.329587,-11.812603,-63.733008}, {0.5,0.5,0.5,1}},
    {{19.743014,-6.978280,-68.685473}, {0.5,0.5,0.5,1}},
    {{20.295502,-1.813105,-70.993326}, {0.5,0.5,0.5,1}},
    {{20.397992,3.498609,-71.294482}, {0.5,0.5,0.5,1}},
    {{20.165688,8.794027,-71.662469}, {0.5,0.5,0.5,1}},
    {{-17.496869,13.272905,-50.065322}, {0.5,0.5,0.5,1}},
    {{-15.208282,15.449287,-48.356494}, {0.5,0.5,0.5,1}},
    {{-12.204102,16.462513,-46.968780}, {0.5,0.5,0.5,1}},
    {{-9.034825,16.562366,-45.629764}, {0.5,0.5,0.5,1}},
    {{-5.922171,16.046801,-44.417982}, {0.5,0.5,0.5,1}},
    {{3.515730,16.555049,-44.428084}, {0.5,0.5,0.5,1}},
    {{6.504847,17.511077,-45.697029}, {0.5,0.5,0.5,1}},
    {{9.646969,17.868475,-47.090450}, {0.5,0.5,0.5,1}},
    {{12.762780,17.314087,-48.529012}, {0.5,0.5,0.5,1}},
    {{15.338637,15.504058,-50.238640}, {0.5,0.5,0.5,1}},
    {{-0.755812,11.614081,-45.009992}, {0.5,0.5,0.5,1}},
    {{-0.565972,8.621884,-42.230376}, {0.5,0.5,0.5,1}},
    {{-0.377136,5.655457,-39.291466}, {0.5,0.5,0.5,1}},
    {{-0.184157,2.697766,-36.477819}, {0.5,0.5,0.5,1}},
    {{-3.430718,-0.804921,-43.624217}, {0.5,0.5,0.5,1}},
    {{-1.678744,-1.169700,-42.708734}, {0.5,0.5,0.5,1}},
    {{0.139543,-1.230575,-41.840747}, {0.5,0.5,0.5,1}},
    {{1.919648,-0.925141,-42.772606}, {0.5,0.5,0.5,1}},
    {{3.585371,-0.322129,-43.695402}, {0.5,0.5,0.5,1}},
    {{-13.211100,9.268335,-49.005244}, {0.5,0.5,0.5,1}},
    {{-11.196948,10.612815,-48.810075}, {0.5,0.5,0.5,1}},
    {{-8.733164,10.858565,-48.776158}, {0.5,0.5,0.5,1}},
    {{-6.530737,9.923226,-49.041845}, {0.5,0.5,0.5,1}},
    {{-8.696171,9.201894,-48.890388}, {0.5,0.5,0.5,1}},
```

```
    {{-11.004340,8.920330,-48.874784}, {0.5,0.5,0.5,1}},
    {{5.203371,10.714781,-49.046221}, {0.5,0.5,0.5,1}},
    {{7.246429,11.905326,-48.858096}, {0.5,0.5,0.5,1}},
    {{9.700226,11.988706,-48.955189}, {0.5,0.5,0.5,1}},
    {{11.853192,10.925634,-49.196762}, {0.5,0.5,0.5,1}},
    {{9.726532,10.296146,-49.007258}, {0.5,0.5,0.5,1}},
    {{7.418772,10.276976,-48.965976}, {0.5,0.5,0.5,1}},
    {{-6.700641,-8.786599,-47.128056}, {0.5,0.5,0.5,1}},
    {{-4.671005,-6.871975,-45.078298}, {0.5,0.5,0.5,1}},
    {{-2.233790,-5.529063,-43.400550}, {0.5,0.5,0.5,1}},
    {{0.471691,-5.565271,-41.820422}, {0.5,0.5,0.5,1}},
    {{3.141672,-5.159842,-43.531126}, {0.5,0.5,0.5,1}},
    {{5.719892,-6.145710,-45.230864}, {0.5,0.5,0.5,1}},
    {{7.964404,-7.765118,-47.265249}, {0.5,0.5,0.5,1}},
    {{5.888823,-9.113807,-46.149808}, {0.5,0.5,0.5,1}},
    {{3.396275,-9.792699,-44.750762}, {0.5,0.5,0.5,1}},
    {{0.803657,-10.010833,-43.313214}, {0.5,0.5,0.5,1}},
    {{-1.809886,-10.157775,-44.632195}, {0.5,0.5,0.5,1}},
    {{-4.414506,-9.844757,-46.030579}, {0.5,0.5,0.5,1}},
    {{-2.262063,-7.610143,-44.645851}, {0.5,0.5,0.5,1}},
    {{0.612076,-7.415914,-42.684745}, {0.5,0.5,0.5,1}},
    {{3.480886,-7.211939,-44.752675}, {0.5,0.5,0.5,1}},
    {{3.469733,-7.368464,-45.467992}, {0.5,0.5,0.5,1}},
    {{0.629268,-7.499055,-43.502819}, {0.5,0.5,0.5,1}},
    {{-2.220072,-7.768379,-45.353626}, {0.5,0.5,0.5,1}},
};

const GLubyte Indices[] = {
    //    // Front
    //    0, 1, 2,
    //    2, 3, 0,
    //    // Back
    //    4, 6, 5,
    //    4, 5, 7,
    //    // Left
    //    8, 9, 10,
    //    10, 11, 8,
    //    // Right
    //    12, 13, 14,
    //    14, 15, 12,
    //    // Top
    //    16, 17, 18,
    //    18, 19, 16,
    //    // Bottom
    //    20, 21, 22,
    //    22, 23, 20




    20, 21, 23,
    21, 22, 23,
    0, 1, 36,
    15, 16, 45,
    0, 17, 36,
    16, 26, 45,
    17, 18, 37,
    25, 26, 44,
```

```
        17, 36, 37,
        26, 44, 45,
        18, 19, 38,
        24, 25, 43,
        18, 37, 38,
        25, 43, 44,
        19, 20, 38,
        23, 24, 43,
        20, 21, 39,
        22, 23, 42,
        20, 38, 39,
        23, 42, 43,
        21, 22, 27,
        21, 27, 39,
        22, 27, 42,
        27, 28, 42,
        27, 28, 39,
        28, 42, 47,
        28, 39, 40,
        1, 36, 41,
        15, 45, 46,
        1, 2, 41,
        14, 15, 46,
        28, 29, 40,
        28, 29, 47,
        2, 40, 41,
        14, 46, 47,
        2, 29, 40,
        14, 29, 47,
        2, 3, 29,
        13, 14, 29,
        29, 30, 31,
        29, 30, 35,
        3, 29, 31,
        13, 29, 35,
        30, 32, 33,
        30, 33, 34,
        30, 31, 32,
        30, 34, 35,
        3, 4, 31,
        12, 13, 35,
        4, 5, 48,
        11, 12, 54,
        5, 6, 48,
        10, 11, 54,
        6, 48, 59,
        10, 54, 55,
        6, 7, 59,
        9, 10, 55,
        7, 58, 59,
        9, 55, 56,
        8, 57, 58,
        8, 56, 57,
        7, 8, 58,
        8, 9, 56,
        4, 31, 48,
        12, 35, 54,
        31, 48, 49,
        35, 53, 54,
```

```
    31, 49, 50,
    35, 52, 53,
    31, 32, 50,
    34, 35, 52,
    32, 33, 50,
    33, 34, 52,
    33, 50, 51,
    33, 51, 52,
    48, 49, 60,
    49, 60, 50,
    50, 60, 61,
    50, 51, 61,
    51, 52, 61,
    61, 62, 52,
    52, 53, 62,
    53, 54, 62,
    54, 55, 63,
    55, 56, 63,
    56, 63, 64,
    56, 57, 64,
    64, 65, 57,
    57, 58, 65,
    58, 59, 65,
    48, 59, 65,
};


@interface HelloGLKitViewController () {
    float _curRed;
    BOOL _increasing;

    GLuint _vertexBuffer;
    GLuint _indexBuffer;
    //GLuint _vertexArray;
    float _rotation;

    GLKMatrix4 _rotMatrix;

    GLKMatrix4 rotationMatrix;
    double xRotation, yRotation, zRotation;
    BOOL resetModelButtonHasBeenPushed;
    NSArray *rotationArray;
    double scale;

    Timer *newTimer;
}



@property (strong, nonatomic) EAGLContext *context;
@property (strong, nonatomic) GLKBaseEffect *effect;


@property (weak, nonatomic) IBOutlet UIImageView *previewImage;

@property (strong, nonatomic) trackerWrapper *tracker;
@property (nonatomic, strong) AVCaptureSession *session;
@property (nonatomic, strong) AVCaptureDevice *device;
@property (nonatomic, strong) AVCaptureDeviceInput *input;
```

```
@property (nonatomic, strong) AVCaptureVideoDataOutput *output;
@property (nonatomic, strong) CIContext *ciContext;
@end

@implementation HelloGLKitViewController
@synthesize context = _context;

- (void)updateVertex :(NSArray * )inputArray {

    int i = 0;


    for (NSArray* xyz in inputArray) {
        Vertices[i] = {{[[xyz objectAtIndex:0] floatValue],[[xyz
            objectAtIndex:1] floatValue],[[xyz objectAtIndex:2] floatValue]},
            {static_cast<float>((i*0.01)) ,static_cast<float>((i*0.01)),
            static_cast<float>((i*0.01)),1}};
        i++;

    }

}

- (void)setupGL {

    [EAGLContext setCurrentContext:self.context];

    self.effect = [[GLKBaseEffect alloc] init];


    glEnable(GL_DEPTH_TEST);



    glGenBuffers(1, &_vertexBuffer);
    glBindBuffer(GL_ARRAY_BUFFER, _vertexBuffer);
    glBufferData(GL_ARRAY_BUFFER, sizeof(Vertices), Vertices, GL_DYNAMIC_DRAW)
        ;

    glGenBuffers(1, &_indexBuffer);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, _indexBuffer);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(Indices), Indices,
        GL_STATIC_DRAW);


    glEnableVertexAttribArray(GLKVertexAttribPosition);
    glVertexAttribPointer(GLKVertexAttribPosition, 3, GL_FLOAT, GL_FALSE,
        sizeof(Vertex), (const GLvoid *) offsetof(Vertex, Position));
    glEnableVertexAttribArray(GLKVertexAttribColor);
    glVertexAttribPointer(GLKVertexAttribColor, 4, GL_FLOAT, GL_FALSE, sizeof
        (Vertex), (const GLvoid *) offsetof(Vertex, Colour));

    _rotMatrix = GLKMatrix4Identity;


}

- (void)tearDownGL {
```

```
    [EAGLContext setCurrentContext:self.context];

    glDeleteBuffers(1, &_vertexBuffer);
    glDeleteBuffers(1, &_indexBuffer);
    self.effect = nil;

}

- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view.

    self.context = [[EAGLContext alloc] initWithAPI:kEAGLRenderingAPIOpenGLES2
        ];

    if (!self.context) {
        NSLog(@"Failed to create ES context");
    }

    GLKView *view = (GLKView *)self.view;
    view.context = self.context;

    view.drawableMultisample = GLKViewDrawableMultisample4X;

    [self setupGL];


    self.tracker = [[trackerWrapper alloc] init];
    [self.tracker initialiseModel];
    [self.tracker initialiseValues];

    resetModelButtonHasBeenPushed = NO;

    if (!(TARGET_IPHONE_SIMULATOR)) {
        [self createAndRunNewSession];
    }
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}


#pragma mark - GLKViewDelegate

- (void)glkView:(GLKView *)view drawInRect:(CGRect)rect {

    glClearColor(0.2, 0.2, 0.3, 1.0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    [self.effect prepareToDraw];


    glDrawElements(GL_TRIANGLES, sizeof(Indices)/sizeof(Indices[0]),
        GL_UNSIGNED_BYTE, 0);
```

```
}

#pragma mark - GLKViewControllerDelegate

- (void)update {


    float aspect = fabsf(self.view.bounds.size.width / self.view.bounds.size.
        height);
    GLKMatrix4 projectionMatrix = GLKMatrix4MakePerspective
        (GLKMathDegreesToRadians(65.0f), aspect, 0.0f, 300.0f);

    self.effect.transform.projectionMatrix = projectionMatrix;

    GLKMatrix4 modelViewMatrix = GLKMatrix4MakeTranslation(0.0f, 0.0f, -(60-
        (scale*2)));



    modelViewMatrix = GLKMatrix4Rotate(modelViewMatrix, [[rotationArray
        objectAtIndex:1] floatValue], 1.0f, 0.0f, 0.0f); // X Rotation
    modelViewMatrix = GLKMatrix4Rotate(modelViewMatrix, -[[rotationArray
        objectAtIndex:2] floatValue], 0.0f, 1.0f, 0.0f); // Y Rotation

    self.effect.transform.modelviewMatrix = modelViewMatrix;


    glBindBuffer(GL_ARRAY_BUFFER, _vertexBuffer);
    glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(Vertices), Vertices);


}

-(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {

}

-(void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {

    UITouch *touch = [touches anyObject];
    CGPoint location = [touch locationInView:self.view];
    CGPoint lastLoc = [touch previousLocationInView:self.view];
    CGPoint diff = CGPointMake(lastLoc.x - location.x, lastLoc.y - location.y)
        ;

    float rotX = -1 * GLKMathDegreesToRadians(diff.y/2.0);
    float rotY = -1 * GLKMathDegreesToRadians(diff.x/2.0);


    bool isInvertible;
    GLKVector3 xAxis = GLKMatrix4MultiplyVector3(GLKMatrix4Invert(_rotMatrix,
        &isInvertible),GLKVector3Make(1, 0, 0));
    _rotMatrix = GLKMatrix4Rotate(_rotMatrix, rotX, xAxis.x, xAxis.y, xAxis.z)
        ;
    GLKVector3 yAxis = GLKMatrix4MultiplyVector3(GLKMatrix4Invert(_rotMatrix,
        &isInvertible),GLKVector3Make(0, 1, 0));
    _rotMatrix = GLKMatrix4Rotate(_rotMatrix, rotY, yAxis.x, yAxis.y, yAxis.z)
        ;
```

```
}


-(CIContext *) ciContext
{
    if(!_ciContext) {
        _ciContext = [CIContext contextWithOptions:nil];
    }
    return _ciContext;
}

#pragma mark - AVFoundationCode
- (AVCaptureDevice *) findFrontCamera
{
    AVCaptureDevice *frontCamera = nil;
    NSArray *devices = [AVCaptureDevice devices];
    for (AVCaptureDevice *currentDevice in devices) {
        NSLog(@"%@", currentDevice);
        if ([currentDevice hasMediaType:AVMediaTypeVideo]) {
            if ([currentDevice position] == AVCaptureDevicePositionFront) {

                frontCamera =  currentDevice;
            }

        }

    }
    return frontCamera;
}




- (void) createAndRunNewSession
{
    self.session = [[AVCaptureSession alloc] init];
    self.session.sessionPreset = AVCaptureSessionPresetMedium;


    self.device = [AVCaptureDevice defaultDeviceWithMediaType:AVMediaTypeVideo
        ];

    self.device = [self findFrontCamera];
    self.input = [AVCaptureDeviceInput deviceInputWithDevice:self.device
        error:nil];

    self.output = [[AVCaptureVideoDataOutput alloc] init];
    self.output.videoSettings = [NSDictionary dictionaryWithObject:[NSNumber
        numberWithInt: kCVPixelFormatType_32BGRA] forKey:(id)
        kCVPixelBufferPixelFormatTypeKey];


    dispatch_queue_t queue;
    queue = dispatch_queue_create("new_queue", NULL);

    [self.output setSampleBufferDelegate:self queue:queue];
```

```
    [self.session addInput:self.input];
    [self.session addOutput:self.output];
    [self.session startRunning];


}



- (void) captureOutput:(AVCaptureOutput *)captureOutput didOutputSampleBuffer:
    (CMSampleBufferRef)sampleBuffer fromConnection:(AVCaptureConnection *)
    connection
{

    @autoreleasepool {

        if(resetModelButtonHasBeenPushed) {
            resetModelButtonHasBeenPushed = NO;
            [self.tracker resetModel];
        }

        CVImageBufferRef imageBuffer = CMSampleBufferGetImageBuffer
            (sampleBuffer);

        UIImage *trackedImage = [self.tracker trackWithCVImageBufferRef:
            imageBuffer];

        // Update vertex with tracked positions
        [self updateVertex:[self.tracker get3dMesh]];



        rotationArray = [self.tracker getRotation];
        scale = [self.tracker getScale];


        [self.previewImage performSelectorOnMainThread:@selector(setImage:)
            withObject:trackedImage waitUntilDone:YES];

    }
}



- (IBAction)resetModelToolBarButton:(UIBarButtonItem *)sender {
    resetModelButtonHasBeenPushed = YES;
    NSLog(@"Model Reset");
}

@end
```

object3D.h
object3D.m

```
//
//  objReader.h
//  fileInputTest
//
//  Created by Tom Hartley on 09/03/2013.
//  Copyright (c) 2013 Tom Hartley. All rights reserved.
//

#import <Foundation/Foundation.h>



typedef struct {
    float Position[3];
    float Normal[3];
    float TextureCoords[2];
} Vertex;


@interface object3D : NSObject

- (void)readInObjfromPath:(NSString *)path;
- (int) getNumberOfFaces;
- (NSArray *) getArrayOfVertex;
- (NSArray *) getArrayOfTextureCoords;
- (NSArray *) getArrayOfNormals;
- (BOOL) hasTexture;
- (void) setVert:(NSArray *)inVertices;


@end
```

```
//
//  objReader.m
//  fileInputTest
//
//  Created by Tom Hartley on 09/03/2013.
//  Copyright (c) 2013 Tom Hartley. All rights reserved.
//

#import "object3D.h"


@interface object3D()

//@property (nonatomic)Vertex *vertices;
//@property (nonatomic) GLubyte *indices;

@end


@implementation object3D {
    uint numberOfVertex;
    uint numberOfFaces;
    uint numberOfNormals;
    uint numberOfTextureCoords;

    //NSMutableArray *meshArray;
    NSMutableArray *arrayOfTextureCoords;
    NSMutableArray *arrayOfVertCoords;
    NSMutableArray *arrayOfNormals;



    int *vertIndexA;
    int *vertIndexB;
    int *vertIndexC;
    float *vertX;
    float *vertY;
    float *vertZ;

    int *normIndexA;
    int *normIndexB;
    int *normIndexC;
    float *normX;
    float *normY;
    float *normZ;


    int *texIndexA;
    int *texIndexB;
    int *texIndexC;
    float *texX; // Stores texture x coords
    float *texY; // Stores texture y coords

}




- (void)readInObjfromPath:(NSString *)path
```

```
{


    NSString *objData = [NSString stringWithContentsOfFile:path encoding:1
        error:NULL];

    NSArray *lines = [objData componentsSeparatedByString:@"\n"];

    for(NSString *line in lines) {


        if([line hasPrefix:@"v "]) {
            numberOfVertex ++;
        }
        if ([line hasPrefix:@"f "]) {
            numberOfFaces ++;
        }
        if ([line hasPrefix:@"vn"]) {
            numberOfNormals ++;
        }
        if ([line hasPrefix:@"vt"]) {
            numberOfTextureCoords ++;
        }

    }

    NSLog(@"Number of VERTEX lines: %d", numberOfVertex);
    NSLog(@"Number of FACE lines: %d", numberOfFaces);
    NSLog(@"Number of NORMAL lines: %d", numberOfNormals);
    NSLog(@"Number of TEXTURE lines: %d", numberOfTextureCoords);




    vertX = malloc(sizeof(GLuint)*numberOfVertex);
    vertY = malloc(sizeof(GLuint)*numberOfVertex);
    vertZ = malloc(sizeof(GLuint)*numberOfVertex);
    vertIndexA = malloc(sizeof(int)*(numberOfFaces));
    vertIndexB = malloc(sizeof(int)*(numberOfFaces));
    vertIndexC = malloc(sizeof(int)*(numberOfFaces));

    texX = malloc(sizeof(float) *numberOfTextureCoords);
    texY = malloc(sizeof(float) *numberOfTextureCoords);
    texIndexA = malloc(sizeof(int)*(numberOfFaces));
    texIndexB = malloc(sizeof(int)*(numberOfFaces));
    texIndexC = malloc(sizeof(int)*(numberOfFaces));

    normX = malloc(sizeof(GLuint)*numberOfNormals);
    normY = malloc(sizeof(GLuint)*numberOfNormals);
    normZ = malloc(sizeof(GLuint)*numberOfNormals);
    normIndexA = malloc(sizeof(int)*(numberOfFaces));
    normIndexB = malloc(sizeof(int)*(numberOfFaces));
    normIndexC = malloc(sizeof(int)*(numberOfFaces));


    numberOfFaces = 0;
    arrayOfTextureCoords = [[NSMutableArray alloc] init];
```

```
arrayOfVertCoords = [[NSMutableArray alloc] init];
arrayOfNormals = [[NSMutableArray alloc] init];



int faceIndex = 0;
int numVert = 0;


// Go through lines and pick out vertices to go into struct
for (NSString *line in lines) {
    if ([line hasPrefix:@"v "]) {

        NSString *lineTrunc = [line substringFromIndex:2];
        NSArray *lineVertices = [lineTrunc
            componentsSeparatedByCharactersInSet:[NSCharacterSet
            whitespaceCharacterSet]];




        vertX[numVert] = [[lineVertices objectAtIndex:0] floatValue];
        vertY[numVert] = [[lineVertices objectAtIndex:1] floatValue];
        vertZ[numVert] = [[lineVertices objectAtIndex:2] floatValue];
        numVert ++;




    } else if ([line hasPrefix:@"f "]) {

        NSString *lineTrunc = [line substringFromIndex:2];
        NSArray *faceGroups = [lineTrunc
            componentsSeparatedByCharactersInSet:[NSCharacterSet
            whitespaceCharacterSet]];

        NSArray *firstFaceGroup = [[faceGroups objectAtIndex:0]
            componentsSeparatedByString:@"/"];
        NSArray *secondFaceGroup = [[faceGroups objectAtIndex:1]
            componentsSeparatedByString:@"/"];
        NSArray *thirdFaceGroup = [[faceGroups objectAtIndex:2]
            componentsSeparatedByString:@"/"];


        vertIndexA[faceIndex] = [[firstFaceGroup objectAtIndex:0] intValue
            ]-1;
        vertIndexB[faceIndex] = [[secondFaceGroup objectAtIndex:0]
            intValue]-1;
        vertIndexC[faceIndex] = [[thirdFaceGroup objectAtIndex:0] intValue
            ]-1;

        numberOfFaces += 3;


        texIndexA[faceIndex] = [[firstFaceGroup objectAtIndex:1] intValue]
            -1;
```

```
            texIndexB[faceIndex] = [[secondFaceGroup objectAtIndex:1] intValue
                ]-1;
            texIndexC[faceIndex] = [[thirdFaceGroup objectAtIndex:1] intValue]
                -1;


            normIndexA[faceIndex] = [[firstFaceGroup objectAtIndex:2] intValue
                ]-1;
            normIndexB[faceIndex] = [[secondFaceGroup objectAtIndex:2]
                intValue]-1;
            normIndexC[faceIndex] = [[thirdFaceGroup objectAtIndex:2] intValue
                ]-1;


            faceIndex++;

        }

    }


    NSLog(@"Number of Faces: %d", numberOfFaces/3);



    int numNorm= 0;
    int numTexture = 0;

    // Go through lines and pick out normals to go into struct
    for (NSString *line in lines) {
        if ([line hasPrefix:@"vn"]) {

            NSString *lineTrunc = [line substringFromIndex:3];
            NSArray *lineVertices = [lineTrunc
                componentsSeparatedByCharactersInSet:[NSCharacterSet
                whitespaceCharacterSet]];




            normX[numNorm] = [[lineVertices objectAtIndex:0] floatValue];
            normY[numNorm] = [[lineVertices objectAtIndex:1] floatValue];
            normZ[numNorm] = [[lineVertices objectAtIndex:2] floatValue];
            numNorm ++;



        }


        else if ([line hasPrefix:@"vt"]) {

            NSString *lineTrunc = [line substringFromIndex:3];
            NSArray *lineVertices = [lineTrunc
                componentsSeparatedByCharactersInSet:[NSCharacterSet
                whitespaceCharacterSet]];


            texX[numTexture] = [[lineVertices objectAtIndex:0] floatValue];
```

```
            texY[numTexture] = [[lineVertices objectAtIndex:1] floatValue];
            numTexture ++;
        }
    }
```

```
    for (int i=0; i<numberOfFaces/3; i++) {

        // Sort Texture Vertex
        int indexA = texIndexA[i];
        int indexB = texIndexB[i];
        int indexC = texIndexC[i];

        NSNumber *xTextCoord = [NSNumber numberWithDouble:texX[indexA]];
        NSNumber *yTextCoord = [NSNumber numberWithDouble:texY[indexA]];
        [arrayOfTextureCoords addObject:@[xTextCoord,yTextCoord]];


        xTextCoord = [NSNumber numberWithDouble:texX[indexB]];
        yTextCoord = [NSNumber numberWithDouble:texY[indexB]];
        [arrayOfTextureCoords addObject:@[xTextCoord,yTextCoord]];


        xTextCoord = [NSNumber numberWithDouble:texX[indexC]];
        yTextCoord = [NSNumber numberWithDouble:texY[indexC]];
        [arrayOfTextureCoords addObject:@[xTextCoord,yTextCoord]];


        // Sort Position Vertex
        indexA = vertIndexA[i];
        indexB = vertIndexB[i];
        indexC = vertIndexC[i];

        NSNumber *xVertCoord = [NSNumber numberWithDouble:vertX[indexA]];
        NSNumber *yVertCoord = [NSNumber numberWithDouble:vertY[indexA]];
        NSNumber *zVertCoord = [NSNumber numberWithDouble:vertZ[indexA]];
        [arrayOfVertCoords addObject:@[xVertCoord,yVertCoord,zVertCoord]];

        xVertCoord = [NSNumber numberWithDouble:vertX[indexB]];
        yVertCoord = [NSNumber numberWithDouble:vertY[indexB]];
        zVertCoord = [NSNumber numberWithDouble:vertZ[indexB]];
        [arrayOfVertCoords addObject:@[xVertCoord,yVertCoord,zVertCoord]];

        xVertCoord = [NSNumber numberWithDouble:vertX[indexC]];
        yVertCoord = [NSNumber numberWithDouble:vertY[indexC]];
        zVertCoord = [NSNumber numberWithDouble:vertZ[indexC]];
        [arrayOfVertCoords addObject:@[xVertCoord,yVertCoord,zVertCoord]];



        // Sort Normal Vertex
        indexA = normIndexA[i];
        indexB = normIndexB[i];
        indexC = normIndexC[i];

        NSNumber *xNormCoord = [NSNumber numberWithDouble:normX[indexA]];
```

```
        NSNumber *yNormCoord = [NSNumber numberWithDouble:normY[indexA]];
        NSNumber *zNormCoord = [NSNumber numberWithDouble:normZ[indexA]];
        [arrayOfNormals addObject:@[xNormCoord,yNormCoord,zNormCoord]];

        xNormCoord = [NSNumber numberWithDouble:normX[indexB]];
        yNormCoord = [NSNumber numberWithDouble:normY[indexB]];
        zNormCoord = [NSNumber numberWithDouble:normZ[indexB]];
        [arrayOfNormals addObject:@[xNormCoord,yNormCoord,zNormCoord]];

        xNormCoord = [NSNumber numberWithDouble:normX[indexC]];
        yNormCoord = [NSNumber numberWithDouble:normY[indexC]];
        zNormCoord = [NSNumber numberWithDouble:normZ[indexC]];
        [arrayOfNormals addObject:@[xNormCoord,yNormCoord,zNormCoord]];
    }

    NSLog(@"Number of vertCords: %d,   Number of texCords: %d,   Number of
        Normals: %d", [arrayOfVertCoords count], [arrayOfTextureCoords count],
        [arrayOfNormals count]);


}



- (void) setVert:(NSMutableArray *)inVertices {

    arrayOfVertCoords = [inVertices copy];
}


- (int) getNumberOfFaces {
    NSLog(@"Number of Faces being returned: %d",numberOfFaces);
    return numberOfFaces;
}


- (NSArray *) getArrayOfVertex {

    return arrayOfVertCoords;
}

- (NSArray *) getArrayOfTextureCoords {

    return arrayOfTextureCoords;
}

- (NSArray *) getArrayOfNormals {
    return arrayOfNormals;
}

- (BOOL) hasTexture {

    BOOL retValue = NO;

    if (numberOfTextureCoords > 0) {
        retValue = YES;
    }

    return retValue;
```

keyFrameMethods.h
keyFrameMethods.mm

```
//
//  keyFrameMethods.h
//  iOSFaceTracker4
//
//  Created by Tom Hartley on 22/04/2013.
//  Copyright (c) 2013 Tom Hartley. All rights reserved.
//

#import <Foundation/Foundation.h>
#import "object3D.h"
#import "trackerWrapper.h"



@interface keyFrameMethods

- (void)interpolateModel:(object3D *)model1
                    with:(object3D *)model2
                      to:(object3D *)movementModel
              withVertex:(NSArray *)vertexMoves
            percentMoved:(float)percentMoved ;


-(NSArray *)getMovementVertexFrom:(object3D *)model1
                         andAlso:(object3D *)model2 ;

-(float) lengthBetweenPoint1:(int)p1
                      point2:(int)p2
                fromTracker:(trackerWrapper *)tracker;


@end
```

```
//
//  keyFrameMethods.m
//  iOSFaceTracker4
//
//  Created by Tom Hartley on 22/04/2013.
//  Copyright (c) 2013 Tom Hartley. All rights reserved.
//

#import "keyFrameMethods.h"


@implementation keyFrameMethods

- (void)interpolateModel:(object3D *)model1 with:(object3D *)model2 to:
    (object3D *)movementModel withVertex:(NSArray *)vertexMoves percentMoved:
    (float)percentMoved {

    NSArray* neutModel = model1.getArrayOfVertex;
    NSArray* finalModel = model2.getArrayOfVertex;
    NSMutableArray* newModel = [movementModel.getArrayOfVertex mutableCopy];


    GLfloat neutModX;
    GLfloat neutModY;
    GLfloat neutModZ;

    GLfloat finalModX;
    GLfloat finalModY;
    GLfloat finalModZ;

    GLfloat diffX;
    GLfloat diffY;
    GLfloat diffZ;


    NSNumber *x;
    NSNumber *y;
    NSNumber *z;


    int i;

    for (int j=0; j<[vertexMoves count]; j++) {

        i = [[vertexMoves objectAtIndex:j] intValue];

        neutModX = [[neutModel[i] objectAtIndex:0] floatValue];
        neutModY = [[neutModel[i] objectAtIndex:1] floatValue];
        neutModZ = [[neutModel[i] objectAtIndex:2] floatValue];

        finalModX = [[finalModel[i] objectAtIndex:0] floatValue];
        finalModY = [[finalModel[i] objectAtIndex:1] floatValue];
        finalModZ = [[finalModel[i] objectAtIndex:2] floatValue];

        diffX = finalModX - neutModX;
        diffY = finalModY - neutModY;
        diffZ = neutModZ - finalModZ;
```

```
        x = [NSNumber numberWithDouble:(neutModX + (percentMoved * diffX))];
        y = [NSNumber numberWithDouble:(neutModY + (percentMoved * diffY))];
        z = [NSNumber numberWithDouble:(neutModZ + (percentMoved * diffZ))];


        [newModel replaceObjectAtIndex:i withObject:@[x,y,z]];
    }

    [movementModel setVert:newModel];


}

-(NSArray *)getMovementVertexFrom:(object3D *)model1 andAlso:(object3D *)
    model2 {

    NSArray* neutModel = model1.getArrayOfVertex;
    NSArray* finalModel = model2.getArrayOfVertex;


    GLfloat neutModX;
    GLfloat neutModY;
    GLfloat neutModZ;

    GLfloat finalModX;
    GLfloat finalModY;
    GLfloat finalModZ;

    NSLog(@"Starting movement from vertex");

    NSMutableArray *movementVertex = [[NSMutableArray alloc] init];

    int i;

    for (i=0; i<[neutModel count]; i++) {

        neutModX = [[neutModel[i] objectAtIndex:0] floatValue];
        neutModY = [[neutModel[i] objectAtIndex:1] floatValue];
        neutModZ = [[neutModel[i] objectAtIndex:2] floatValue];

        finalModX = [[finalModel[i] objectAtIndex:0] floatValue];
        finalModY = [[finalModel[i] objectAtIndex:1] floatValue];
        finalModZ = [[finalModel[i] objectAtIndex:2] floatValue];

        if (neutModX!=finalModX || neutModY!=finalModY || neutModZ!=finalModZ)
            {
            [movementVertex addObject:[NSNumber numberWithInt:i]];
        }
    }

    NSLog(@"Vertex points that move: %d", [movementVertex count]);
    return [movementVertex copy];

}

-(float) lengthBetweenPoint1:(int)p1 point2:(int)p2 fromTracker:
    (trackerWrapper *)tracker{
```

```
    float length = 0;



    NSArray *point1 = [tracker getSpecificPoint:p1];
    NSArray *point2 = [tracker getSpecificPoint:p2];

    float p1X = [[point1 objectAtIndex:0] floatValue];
    float p1Y = [[point1 objectAtIndex:1] floatValue];
    float p1Z = [[point1 objectAtIndex:2] floatValue];

    float p2X = [[point2 objectAtIndex:0] floatValue];
    float p2Y = [[point2 objectAtIndex:1] floatValue];
    float p2Z = [[point2 objectAtIndex:2] floatValue];


    float newX = powf((p1X-p2X),2);
    float newY = powf((p1Y-p2Y),2);
    float newZ = powf((p1Z-p2Z),2);

    length = sqrtf((newX+newY+newZ));
    return length;

}


@end
```

oscMethods.h
oscMethods.m

```
//
//  oscMethods.h
//  iOSFaceTracker5
//
//  Created by Tom Hartley on 01/05/2013.
//  Copyright (c) 2013 Tom Hartley. All rights reserved.
//

#import <Foundation/Foundation.h>
#import <VVOSC/VVOSC.h>

@interface oscMethods:NSObject


- (void) setupWithAddress:(NSString *)networkAddress andPort:(int)port;
- (void)sendOSCMessage:(NSArray *)msgArray;


- (void)sendIntMessage:(int)number toAddress:(NSString *)address;
- (void)sendFloatMessage:(float)numberIn toAddress:(NSString *)address;

@end
```

```
//
//  oscMethods.m
//  iOSFaceTracker5
//
//  Created by Tom Hartley on 01/05/2013.
//  Copyright (c) 2013 Tom Hartley. All rights reserved.
//

#import "oscMethods.h"

@implementation oscMethods {
    OSCManager *manager;
    OSCOutPort *outport;

}

- (void) setupWithAddress:(NSString *)networkAddress andPort:(int)port {

    manager = [[OSCManager alloc] init];
    [manager setDelegate:self];
    outport = [manager createNewOutputToAddress:networkAddress atPort:port];

}

- (void)sendOSCMessage:(NSArray *)msgArray {


    OSCMessage *msg = nil;


    // make an OSC message
    msg = [OSCMessage createWithAddress:@"/Address/Path/1"];

    int i=0;
    for (NSArray* xyz in msgArray) {

        [msg addFloat:[[xyz objectAtIndex:0] floatValue]];
        [msg addFloat:[[xyz objectAtIndex:1] floatValue]];
        [msg addFloat:[[xyz objectAtIndex:2] floatValue]];

        i++;
    }


    // send the OSC message
    [outport sendThisMessage:msg];
}


- (void)sendIntMessage:(int)numberIn toAddress:(NSString *)address {

    // make an OSC message
    OSCMessage *msg = nil;
    msg = [OSCMessage createWithAddress:address];
    [msg addInt:numberIn];

    // send the OSC message
    [outport sendThisMessage:msg];
```

```
}

- (void)sendFloatMessage:(float)numberIn toAddress:(NSString *)address {

    // make an OSC message
    OSCMessage *msg = nil;
    msg = [OSCMessage createWithAddress:address];
    [msg addFloat:numberIn];

    // send the OSC message
    [outport sendThisMessage:msg];
}

@end
```