Munro, Nicole

Real Time Face Tracking

Graduation work 2018-19

Digital Arts and Entertainment

Howest.be

Contents

[2 Abstract 2](#_Toc529002476)

[3 Introduction 3](#_Toc529002477)

[4 Research 4](#_Toc529002478)

[4.1 Facial tracking in real time with c++ and other 3rd party libraries 4](#_Toc529002479)

[4.1.1 The different 3rd party libraries available 4](#_Toc529002480)

[4.1.2 The difference between the libraries 4](#_Toc529002481)

[4.1.3 Which one was chosen and why 6](#_Toc529002482)

[4.2 Sending the information from the facial tracker to our engine 6](#_Toc529002483)

[4.2.1 The different methods 6](#_Toc529002484)

[4.2.2 Which one was chosen and why 7](#_Toc529002485)

[4.3 Real time mesh manipulation 7](#_Toc529002486)

[4.3.1 Blend shapes 7](#_Toc529002487)

[5 Case study 8](#_Toc529002488)

[5.1 introduction 8](#_Toc529002489)

[5.2 Tracking a face using OpenCV 8](#_Toc529002490)

[5.2.1 Installation 8](#_Toc529002491)

[5.2.2 Tracking a face 8](#_Toc529002492)

[5.2.3 Facial landmarks 8](#_Toc529002493)

[5.3 Sending the information into Unreal 9](#_Toc529002494)

[5.3.1 By reading and writing to a file 9](#_Toc529002495)

[5.3.2 Creating a DLL 10](#_Toc529002496)

[5.4 Building a system to manipulate the face rig 10](#_Toc529002497)

[5.4.1 With Blueprints 10](#_Toc529002498)

[5.4.2 Calibration 11](#_Toc529002499)

[5.4.3 Accounting for head movement 11](#_Toc529002500)

[5.4.4 The results 12](#_Toc529002501)

[5.4.5 Refactoring the code 12](#_Toc529002502)

[6 Conclusion 12](#_Toc529002503)

[7 References 14](#_Toc529002504)

# Abstract

This paper will explain the topic of real-time facial feature detection and application to a rigged facial model. A single camera will be used to track the face and third-party libraries will be implemented to find these facial features. A facial tracker app will be built that smoothly tracks the face and it’s features, this tracker will write the values of the tracked points to a file that will be read by an engine, real-time, and then those tracked points will be applied to bones and blend-weights in the rig.

# Introduction

Real-time facial feature and detection has become more broadly used for video game animations. This is the case because video games and movies want to be more realistic and that can be achieved by capturing real human expressions. Capturing these facial animations real-time is also important because we no longer want to wait for an animation to render out to see if it’s good enough for the game.  
This paper will go over the steps taken to track a face and it’s features real time and applying those tracked features to a facial rig in real-time.  
This paper will also explain the details of the libraries we will be using for facial feature tracking, what library we ended up going with and why. We will also discuss the different ways to communicate the tracked points from the facial tracker to Unreal engine. Then how we chose to manipulate the facial rig.

Our first research question would be how we could go about tracking a face and it’s features. Once we have determined what 3rd party libraries exist for facial tracking, we can do more research to determine which one to use and why. Research will also be done on what happens behind the scenes with these different libraries.

Once we have the knowledge of what libraries we need to use then we can go about researching how we can take the information from the facial tracker application to the engine we are using.

Then we are going to take the information from the facial tracker app and apply it to the bones in the face of the rig. Here we must research what the best way is to manipulate a rig in real time.

# Research

## Facial tracking in real time with c++ and other 3rd party libraries

### The different 3rd party libraries available

When it comes to real time face tracking and face recognition many libraries and existing software bases their programs on two c++ toolkits. These two are OpenCV and Dlib, and they are either used in conjunction or separately.

There are also face tracking applications available that use OpenCV and Dlib that work out of the box. Examples of the ones found were OpenFace, FaceTracker and OpenFaceTracker. These were all free to use accurate face tracker applications.

In this part of the paper, the different libraries will be discussed, and, in the end, we will conclude which library was chosen and why.

### The difference between the libraries

#### OpenCV

OpenCV (Open Source Computer Vision Library) has a focus in real time applications and is used frequently for facial tracking and facial feature tracking. Most facial tracking applications that are found online use OpenCV.

OpenCV uses the Haarcascade Classifier which is a machine learning algorithm, and a cascade function is trained from images. The algorithm first takes a lot of images with a face then a lot of images without a face to train itself. Once it can recognize a face we can move on to recognizing features. The way it works is that each feature is a value obtained by subtracting the sum of pixels under the white rectangle from the sum of pixels under the black rectangle. These are called kernels.

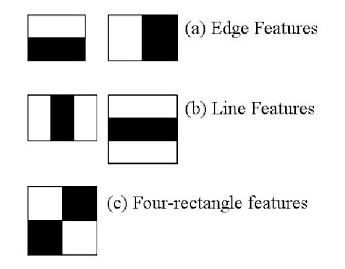
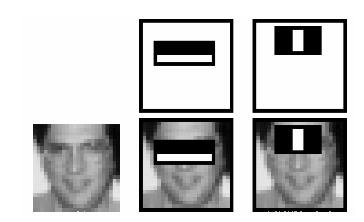


Figure : Line and edge feature detection

Figure The different kinds of detection

Now there is the issue that we need all possible sizes and locations of the kernels to detect the features of the face, which is a lot of computation even for a 24x24 image we have 160 000 features. Thus, the solution was to come up with an Integral image, which reduces the operations of each pixel to just 4 pixels.

There is another issue with this system and that is that for most of an image, the data does not contain facial data. So, we discard any data that is not a face and we only apply the feature checking on parts of the image that contain a face.

Enter the concept of Cascade of Classifiers: the features of the face are grouped into different stages and are applied. So, there are 1, 10, 15 and 50 features in the first 5 stages. If on the first stage no features are detected the image has failed and it will not move on to the next stage.

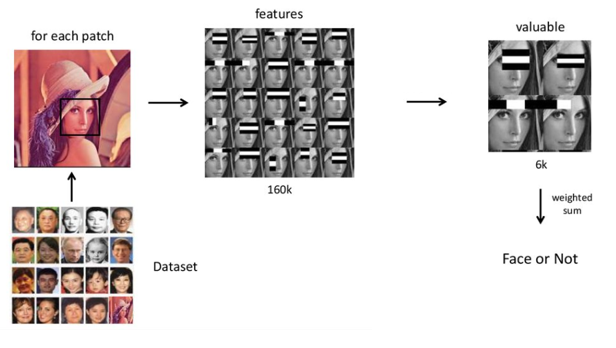
****

Figure The workflow of the Haarcascade algorithm

This runs at ~100ms on the CPU and can be inaccurate and creates false positives.

#### Dlib

Dlib is a cross platform library written in c++ and is heavily influenced by component-based software patterns. [12] Dlib is being used for real time car tracking applications for self-driving cars. But it also contains software components for dealing with networking, threads, graphical interfaces, complex data structures, linear algebra, statistical machine learning, image processing, data mining, XML and text parsing, numerical optimization, Bayesian networks, and numerous other tasks. In recent years, much of the development has been focused on creating a broad set of statistical machine learning tools.

Dlib uses machine learning algorithms and it’s face detector is based on histogram of oriented gradients (HOG) and linear SVM. With HOG the intensity of gradients or edge directions can describe the local object appearance. A histogram of gradient directions can be complied by dividing the image into small connected regions called cells and the pixels in the cells would be used to calculate this histogram.

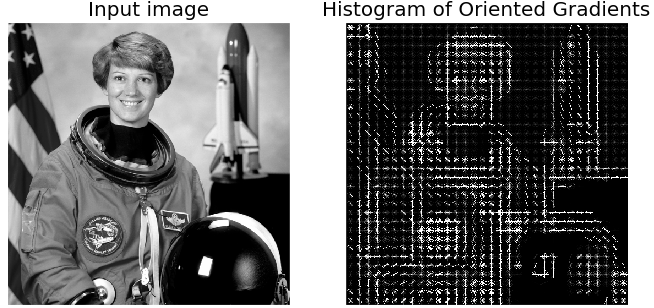


Figure HOG algorithm for edge detection

The concept of linear SVM (support vector machine) is that it is given a set of training examples, and the training algorithm assigns new examples to one category or the other, thus making it a non-probabilistic binary linear classifier.

### Which one was chosen and why

Ultimately, the facial tracking application would be built using OpenCV.

OpenCV has extensive documentation online, with an active community of people using the library to create software applications.

The reason why it was chosen was because there is a very active OpenCV community with people posting on forums and using chat services to talk to each other about issues. OpenCV also has functionality to easily use the GPU instead of the CPU for landmark detections, which is inherently faster.

## Sending the information from the facial tracker to our engine

To start manipulating a mesh real time, the data from the Face Tracking application should be sent to the engine being used.

### The different methods

#### Creating the Face Tracker in the chosen engine

Instead of creating the Face Tracker application in c++ in a separate Visual Studio solution, the engine that being used could be linked to OpenCV and the landmark detection could be built into it.

#### Sending the tracked data to the engine via reading and writing to a file every frame

The Face Tracker application could write the tracked data in a logical manner to a .csv file every frame, then the engine would read from that file and parse the data.

This is an issue for several reasons, one being how slow it is to read and write to a file. Every tick the Face Tracker application would have to open the file and re-write the data, and the engine of choice would read that data and parse it – and computationally this can be incredibly slow.

Another issue with reading and writing to the same file every frame is I/O contention, this causes degradation of performance. I/O contention is created when there are high amounts of read and write operations happening. Contention describes two or more devices competing for the same resources. [1]

#### Creating a DLL that links the Face Tracker application to the chosen engine

With this method the Face Tracker application would be a DLL that the engine loads up at the start of the game running, and the engine would be able to read the data in real time from the Face Tracker without any latency.

A dynamic link library (DLL) is used when a larger program needs to run smaller tasks at the same time. DLLs cannot be run on their own, but rather are called upon by other code that is already running. The advantage of using DLLs is that space is saved in RAM, because the files don’t need to be loaded together with the main program.

Creating a DLL also promotes modular architecture, with the DLL being separate it can easily be changed or rectified with no changes being made to the main program.

When creating functions (in the Face Tracker) that can be called by our chosen engine, these functions must be written in c. Thus, the Face Tracking system would be separated from the DLL functions that our engine calls because the Face Tracking system is written in c++.

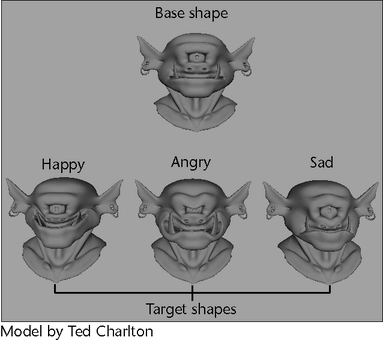
### Which one was chosen and why

In the case study both the options of reading and wring to a file, and creating a DLL are explored. In the end creating a DLL was chosen as it ran faster than the file reading/writing and it made the most sense for this project.

Building a DLL meant that we could easily change and manipulate the Face Tracking software without changing the code of the actual project.

## Real time mesh manipulation

### Blend shapes

In facial rigging and animation, animators can use controllers to create blend weights for a rig. Blend weights allow animators to easily create facial expressions – and gives them the ability to blend between them.

Blend weights of a facial rig can be exported to Unreal engine (called morph targets in Unreal) and can be manipulated in real time, for the purposes of this project the morph targets will be manipulated by the data that is given by the Face Tracker.

# Case study

## introduction

This part of the paper will discuss the steps taken and the entire pipeline of Real Time Facial Tracking and animations.

## Tracking a face using OpenCV

### Installation

To track a face the first step is to fork the OpenCV repository and its contributions, because it is within the contributions that the facial landmark detection algorithms exist. Instructions for building the binaries for OpenCV and its contributions are on the Github page of the contributions. [[4]](#_References)

The readme suggests downloading CMake to build the libraries. CMake is a tool designed to build software and control the software compilation process. [5] Once the libraries had been built using Cmake, the Visual Studio project needed to be connected to the built libraries. To do this on a project level instead of local, the following steps should be followed.

The Visual Studio project needs to be directed to the correctly built libraries of OpenCV to use its functionality, to do this in the Visual Studio solution, go to the properties page. From the properties page go to the linker settings, then direct the additional include directories to the built binaries of OpenCV. Then link it to the certain libraries that you need for the application that you are building, once that is done your Visual Studio should be connected to OpenCV and it is possible to use all its functionalities. [13]

### Tracking a face

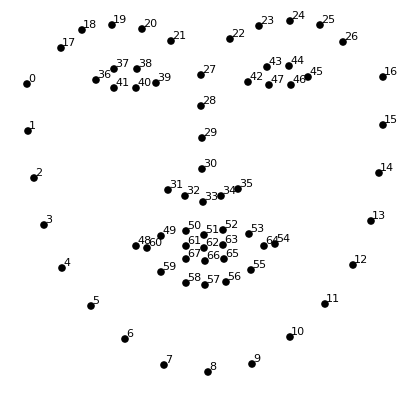
Once the installation process is done, the next step is to open and read the data from a webcam then use that camera data to detect if a face is in the screen. To do this, OpenCV uses its Haarcascade classifiers to determine if there is a face on the screen and if so, where it is.

The first step was to use OpenCV’s functionality to open up a camera then to use the detectMultiScale() function to determine how many faces are in frame. This function works with a Cascade Classifier that has a loaded and trained data model of faces to detect faces in real time. This then returns the tracked face in the form of a vector of rectangles, that shows where the face is on screen.

### Facial landmarks

Once the face is tracked, the next step is to track the facial landmarks on the face itself. Using OpenCV and the tracked face(s) from the previous chapter – the function fit() is called on our Facemark, our Facemark also used a trained data model to detect features of the face. The function fit then will fill a vector of Point2fs with 68 tracked data points, this is then called each tick and thus a face is being tracked with all its landmarks.

With these tracked landmarks working, the next stage of the face tracking pipeline is to communicate the tracked points to the engine.

****

## Sending the information into Unreal

When it comes to reading the tracked data in Unreal, there are several methods that were found in the research stage of this project. Two of the methods were tested in the case study.

### By reading and writing to a file

The easiest method of sending the data to Unreal would be to write the information tracked to a csv file then to read that file every frame from Unreal. The CSV file layout was simple: x,y(endline). By keeping the layout simple it made the data easier to parse in Unreal. Once the Face Tracker was writing the data to a file, we could read the data with Unreal and c++. [9]

To read the file from Unreal, a c++ Unreal solution was created. In the solution a ReadFile class was created, and functions for loading and saving a file were created too. These functions took the file name as a parameter and returned the text read from the .csv as a string. Because the data in the .csv was just the x value, y value then an endline – the data was simple enough to parse in Blueprints.

To parse the data in Blueprints, an Actor is created with a FileReader object as a member variable, in the event tick the Actor would read the data from the file and use the comma as a delimiter to save the tracked data to an array of Vector2Ds.

#### The issues

The game was running at 5FPS and the framerate kept dropping, because of reading and writing to a file.

Another issue was that the Face Tracker had to be started manually then the Unreal game would have to be started manually too. This was cumbersome and not ideal – starting the Face Tracker .exe from Unreal was also an option but it was not further researched. [10]

### Creating a DLL

The first step is to create a DLL with Visual Studio, then to create a class that exports the DLL functions that Unreal can call. These functions need to be written in c, thus the Face Tracker code should be separated from the exported DLL functions. The general idea behind creating the DLL was to wrap up the functionality of the face tracker and being able to call this compartmentalized code from the Engine itself. The benefit to using a DLL method is that theoretically it can be used by another Engine with few issues.

To create a DLL project, in Visual Studio create a DLL application type. Then a class is added such as a DLL class, this class will handle all the functions that the Engine would call to obtain the Face Tracker data. If c++ is being used it needs to be converted to c, to prevent C++’s name mangling of method names. To do this use extern “C” – then the functions that we want the engine to call would be defined in there. The main cavate with using c instead of c++ is that functions cannot return vectors, let alone vectors of a nonstandard type (i.e. a vector of point2fs). Therefore, if the engine were to call a function such as getFacialLandmarks, the function would return nothing and take the parameters of a pointer to the beginning of the array, and the size of the array. Thus, the array pointer is filled in the Face Tracker application.

Once we have solved the issue of how to return the tracked data points the DLL would be created in Unreal. To do this first a class is created in the Unreal project that would handle the importing of the methods from the DLL and create the proc that would execute the DLL. To execute the DLL the file path of the DLL should be found, which would be stored in the folder structure of the Unreal project. Once the DLL has been loaded the methods from the DLL are imported, to do this the DLL proc that has been loaded will then take the names of the functions that it has been given (ex. getFacialLandmarks) and the names must be the same. If the DLL proc finds these functions within the DLL, then they are considered imported and are now callable.

These now callable methods are called in different functions, and getFacialLandmarks would be called every tick, therefore returning the tracked facial landmarks from the DLL.

## Building a system to manipulate the face rig

### With Blueprints

For the prototyping stage of the facial rig manipulation system, Unreals Blueprint system was used. Using the Blueprints allowed for quick prototyping and creating the logic behind the system.

Essentially the system would need to know which morph targets correspond to what facial feature points that are being tracked. To do this a struct would be created that stores the information of the name of the morph and the indexes of the facial feature that manipulates the morph target.

The system could then calculate distances of the certain points of the face that are being tracked, and that distance would then move the morph target. The issue is that the system would not know what the maximum distance is that a point can move, such as when the mouth curls into a smile the corners of the mouth move to a certain maximum distance. That is when it was decided that the face and its features should be calibrated.

### Calibration

The calibration stage involves the user of the program making certain facial expressions (i.e. neutral, smiling, frowning), and the system could keep track of all the points of the face and how far the points move in the most extreme positions. This way the system can calculate a percentage of how far a point is towards its most extreme position. With this percentage the morph target can be manipulated accordingly.

The way the calibration stage was set up was as follows, in Blueprints it was programmed that a message is printed to the screen such as ‘happy’, when that message appears the user has 3 seconds to make a happy face, then a screenshot of their face is taken in that happy position. It then goes on to calibrating a sad, angry and surprised face. With these different facial expressions stored in the system, the system would then be able to know how far a point is to its most extreme position. For example, if the system is tracking the lower lip, and needs to know how far the users lower lip is close to being fully opened. By checking the distance between the lower lips neutral position and its most extreme position (such as a surprised face) the system can then know how far to move point along.

Morph targets use a value between 0 – 1 to change the way a certain facial feature moves. Now that the system can know how the current distance of the point compares to the maximum distance of that point, it can calculate a value between 0 – 1, and this value would be applied to the morph target.

### Accounting for head movement

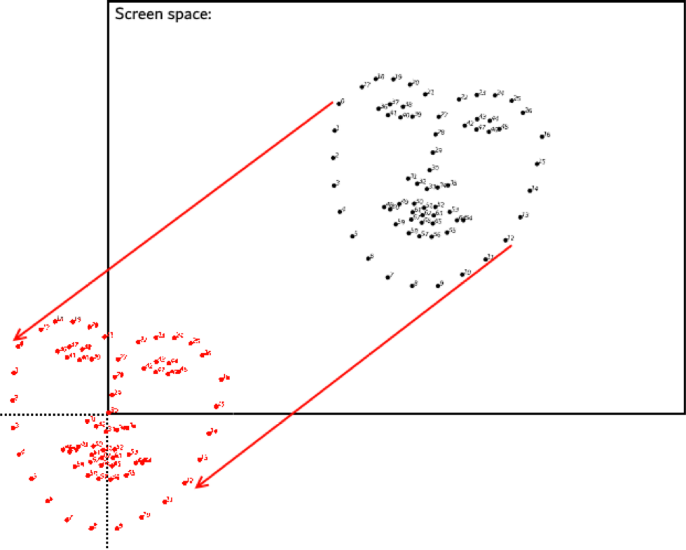
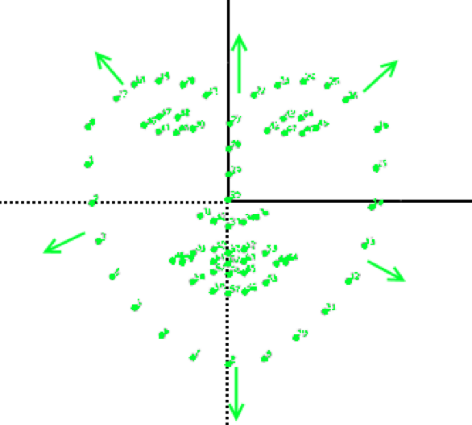
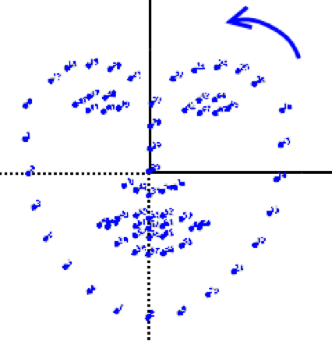
When calibrating an issue occurred that the users head would move closer/further away from the webcam, or the head would be tilted at a certain angle, which affected the way the distance calculations were done. To fix this the system would have to first translate all the points around the origin, then calculate the angle that the head is tilted at and rotate the points back, then calculate the scale of how close the user is to the webcam and scale the points up or down accordingly. [11]

#### Calculating translation, scale and rotation

To translate the points to around the origin, a point was chosen on the face that would be considered the origin, this point would have to be one that does not move much with the face if certain expressions are made thus, the tip of the nose was chosen. Then translating the points was a matter of obtaining the position of the tip of the nose in that frame and applying that vector to all the points of the face. This would make the tip of the nose move to 0,0 and all the other points would be moved around it.

For the rotation of the face, a vector was created from the tip of the nose to the point between the eyebrows, and another vector was created – a right vector. Using the dot product, the system could then calculate the amount that the head is tilted in that frame [14]. And since the points of the face have been translated to around origin, the rotation could be applied to each point accordingly.

Being closer and further from the screen also had a huge impact on how distance checks could be done, because if the user moved closer the points were further away from each other and if they moved further away the points were closer together. To fix this the system would have to create a scale of how far the user is from the screen, then that scale could be applied to each point of the face to normalize them. The scale would be calculated by taking the distance between the two temples of the face in neutral position, and that distance would be considered the ‘normal’ distance for the temples, if the user were to move closer in the next frame the distance would be greater, and the scale would increase.



The image above illustrates the translation, rotation and scale that the tracked face points go through, respectively.

### The results

By doing these calculations, the system no longer must worry about how close/far or tilted the head is. And the distance calculations that would be done to determine how far a point has moved from its neutral position, won’t be affected by the head being in a certain position. When the system was first being built, it was thought that a user could keep their head still for simplicities sake, but it soon became appareSnt that the user would move/tilt their head no matter what and this had a huge impact on how distance calculations were done, and the results of the face tracker were not desirable.

### Refactoring the code

The system was originally built in Blueprints, and once it had to start accounting for head movement (discussed in the previous paragraph), the Blueprint code became unreadable and difficult to debug. This is when it was decided that the system should be written in c++.

# Conclusion

In the past few years, real time facial tracking and mo-cap for faces and human bodies has become more and more important in the gaming industry. The industry has created a demand for realistic video games, with bigger environments and more gameplay. This has led to the demand for realistic video game characters, with believable facial features, expressions and movement. Thus, real time facial tracking and animations need to be heavily utilized to create these life-like characters quickly and efficiently. In the past motion capture data would be captured beforehand then the data would be cleaned up and applied to rigged characters in a game. These days we have the technology to capture the data in real time and apply it in real time to rigged in game characters – to obtain results as fast as possible.

When researching methods that companies use for real-time facial capture and animation, usually an actor is wearing a head mounted camera, and they have points drawn on their faces that the system would track. With a steady camera and tracked points on the face and good diffuse lighting on the actor, this would create a system of smoothly tracked face points. With these smoothly tracked points, the game companies would use an in-house system to manipulate the rigged character. The result is smooth and believable, with actors being able to do several takes to try and get the perfect result.

The pipeline for building a real time facial tracking and animation system from scratch proves to be tricky, with the main caveat being the libraries which are being used for the marker-less facial tracking and detection. When it comes to marker-less real time facial tracking, there is not a surplus of libraries out there – and the ones that do exist have their limitations, such as a low frame rate and inaccurate tracking. We see that there are accurate smooth face tracking systems that exist, but for this project building one from scratch was part of the learning process.

Once the data is tracked and sent to our engine of choice, building a system that perfectly manipulates a mesh and its morph targets in real time is tricky. Each feature of the human face reacts differently to how certain parts of the face move, and since there is no control over which points of the face are tracked, there is some data which the system would not have access to see if those certain features are moving. The best system that a singular person could come up with in such a short time span would be one that handles certain points of the face moving and manipulating the morph targets accordingly (i.e. smiling, frowning, lip moving) and when it comes to more complex movement of the face, more complex systems would need to be created that take many parameters into account.

Overall, this project was about seeing if it is possible to run through the entire pipeline of real time facial manipulation and animations as a singular person with only 6 weeks, and through my findings it is entirely possible. The result would be noticeably better if the decision was made to use an existing facial tracking software such as OpenFace, that provides a smooth working facial tracker that uses deep learning algorithms to create the most accurate results. By creating a Face Tracking system from scratch, skills were obtained on how to use OpenCV, CMake, how to build a DLL and link DLLs to Unreal and much more.

# References

1. SearchStorage. 2018. What is I/O contention (input/output contention)? - Definition from WhatIs.com. [ONLINE] Available at: <https://searchstorage.techtarget.com/definition/I-O-contention-input-output-contention>. [Accessed 10 October 2018].
2. Lifewire. 2018. What Is a DLL File? (Dynamic Link Library). [ONLINE] Available at: <https://www.lifewire.com/what-is-a-dll-file-2625852>. [Accessed 11 October 2018].
3. tutorialspoint.com. 2018. DLL How to Write. [ONLINE] Available at: <https://www.tutorialspoint.com/dll/dll_writing.htm>. [Accessed 11 October 2018].
4. Paint blend shape weights | Maya 2018 | Autodesk Knowledge Network. 2018. Paint blend shape weights | Maya 2018 | Autodesk Knowledge Network. [ONLINE] Available at: <https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2018/ENU/Maya-CharacterAnimation/files/GUID-950716A4-19CF-4C14-96F1-FD6B4D48663C-htm.html>. [Accessed 15 October 2018].
5. Overview | CMake. 2018. Overview | CMake. [ONLINE] Available at: <https://cmake.org/overview/>. [Accessed 29 October 2018].
6. GitHub. 2018. GitHub - opencv/opencv\_contrib: Repository for OpenCV's extra modules. [ONLINE] Available at: <https://github.com/opencv/opencv_contrib>. [Accessed 03 October 2018].
7. OpenCV: Deep Neural Network module. 2018. OpenCV: Deep Neural Network module. [ONLINE] Available at: <https://docs.opencv.org/3.4.3/d6/d0f/group__dnn.html>. [Accessed 01 October 2018].
8. Facial Landmark Detection | Learn OpenCV. 2018. Facial Landmark Detection | Learn OpenCV. [ONLINE] Available at: <https://www.learnopencv.com/facial-landmark-detection/>. [Accessed 01 October 2018].
9. Unreal Engine. 2018. File Management, Create Folders, Delete Files, and More - Epic Wiki. [ONLINE] Available at: <https://wiki.unrealengine.com/File_Management,_Create_Folders,_Delete_Files,_and_More#Create_Text_Files>. [Accessed 16 October 2018].
10. api.unrealengine.com. 2018. Create Proc. [ONLINE] Available at: <https://api.unrealengine.com/INT/API/Runtime/Core/GenericPlatform/FGenericPlatformProcess/CreateProc/index.html>. [Accessed 29 October 2018].
11. Geometrical Transformations. 2017. [ONLINE] Available at: http://www.cs.brandeis.edu/~cs155/Lecture\_06.pdf. 2017. [Accessed 23October 2018]
12. Github. 2018. Dlib: Repository for Dlib.[ONLINE] Available at: <https://github.com/davisking/dlib> . [Accessed 01 October].
13. How to build applications with OpenCV inside the Microsoft Visual Studio — OpenCV 2.4.13.7 documentation. 2018. How to build applications with OpenCV inside the Microsoft Visual Studio — OpenCV 2.4.13.7 documentation. [ONLINE] Available at: <https://docs.opencv.org/2.4/doc/tutorials/introduction/windows_visual_studio_Opencv/windows_visual_studio_Opencv.html>. [Accessed 02 November 2018].
14. Angle between two vectors. 2018. Angle between two vectors. [ONLINE] Available at: <http://onlinemschool.com/math/library/vector/angl/>. [Accessed 28 October 2018].