universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„Animation of soft-looking Facial Skin and Muscle Movements"

verfasst von / submitted by

Filip Vidović, BSc.

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of
Diplom-Ingenieur (Dipl.-Ing.)

Wien, 2018 / Vienna, 2018

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	A 066 935
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Medieninformatik
Betreut von / Supervisor:	Univ.-Prof. Dr. Helmut Hlavacs

Declaration of Authorship

I, Filip Vidović, BSc, declare that this thesis titled, “Animation of soft-looking Facial Skin and Muscle Movements” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

_____

Date:

_____

Acknowledgments

I want to express my sincere thanks to Univ.-Prof. Dr. Helmut Hlavacs, who had guided me through writing this master's thesis, always had very valuable hints when I got stuck and who had shown incredible patience.

I would also like to thank my family and friends for being supportive this entire time. Four of them deserve a special mention though. I thank my brother Ivo Vidović, Alexandra Mădălina Tamaș and Maximilian Egger, whose mathematical expertise I consulted. Furthermore, I thank Krzysztof Przybysz, who gave me an excellent crash course on mesh manipulation. Thank you all so much!

Abstract

English

In this master's thesis a simple, constraint-based skin simulation for faces is presented, which is independent of face meshes or the way they were obtained. Furthermore, a muscle simulation presented as well, which deforms a face mesh in an efficient and simple manner. The basis for both simulations is the master's thesis of Leon Beutl from 2011 and a paper written by Leon Beutl and Helmut Hlavacs in 2012. The approaches presented in these works had been revised and adapted with the goal of integrating it into a bigger project, namely a Virtual Reality conference room, in mind. In scope of this task, a showcase application was written to demonstrate the skin and muscle simulations, to examine the feasibility of a port from OGRE to Unity, as well as to highlight the difficulties which can be expected. Furthermore, to allow greater reusability, the mathematical calculations needed for the skin and muscle simulations had been detached from the showcase application, which will allow future projects to make use of the results of this thesis in a straightforward manner.

Deutsch

In dieser Masterarbeit wird eine einfache, bedingungsbasierte Hautsimulation für Gesichter präsentiert, welche unabhängig von Gesichtsmeshes oder deren Gewinnung ist. Desweiteren wird eine Muskelsimulation vorgestellt, die ein Gesichtsmesh auf effiziente und einfache Art deformiert. Die Basis für beide Simulationen bildete die Masterarbeit von Leon Beutl aus dem Jahre 2011, als auch das Paper von Leon Beutl und Helmut Hlavacs aus dem Jahre 2012. Die in diesen Arbeiten vorgestellten Herangehensweisen wurden überarbeitet und adaptiert mit dem Ziel der Integration in ein größeres Projekt, nämlich das Projekt Konferenzraum in der virtuellen Realität. Im Zuge dieser Arbeit wurde eine Vorzeigeapplikation geschrieben um die Haut- und Muskelsimulationen zu demonstrieren, die Durchführbarkeit einer Portierung von OGRE auf Unity zu untersuchen, als auch die Schwierigkeiten aufzuzeigen, die dabei zu erwarten sind. Darüberhinaus wurden, um größere Wiederverwendbarkeit zu ermöglichen, die mathematischen Berechnungen für die Haut- und Muskelsimulationen von der Vorzeigeapplikation entkoppelt, womit zukünftigen Projekten es ermöglicht wird, auf den Ergebnissen dieser Masterarbeit auf unkomplizierte Weise aufzubauen.

Table of Contents

List of Figures

1 Introduction

1.1 Motivation

One of the most intriguing and challenging usages of computer graphics is the realistic simulation of human faces. The application areas are numerous. Perhaps the biggest sector is the entertainment sector. Animated faces are a must-have for 3D animated films, while they are encountered more and more often also in live-action film productions for animated characters. Video games rely almost exclusively on animated characters, being another big field of application for animated faces. Recently, there has also been a surge of animated avatars for various communication channels, especially in combination with the growing Virtual Reality sector, like for example Facebook Spaces [25]. There are, however, also other sectors besides the entertainment sector, which can make good use of a simulation of a human face. For medical purposes it can for instance be a very helpful tool to have an animation of a patient before surgery, or having a schematic head, on which for instance the workings of the facial muscles or the characteristics of human skin can be analyzed and taught.

In particular in applications, where an actual human being shall be realistically represented, difficulties frequently arise. While making animated faces soon are by all means and purposes recognizable as human-like and thus eliciting a positive reaction, increasing with the quality of the animation, this reaction at one point drops sharply. While still unmistakably human, the now rather minor differences seem to feel more repulsive to the viewer, thus eliciting either a rather negative emotional response or simply disbelief. This phenomenon goes by the name "Uncanny Valley" and is known from the field of robotics and has been described already in 1970 [16]. This valley can be bridged by further increasing the quality of the animation, eliminating more of the imperfections of the animated face.

One of the most important factors for making a face believable, are wrinkles in the skin. Whenever a facial muscle is moved, human skin on the face wrinkles to a certain degree, which depends on the contracted muscle as well as how strong the muscle is contracted. These wrinkles are important for

conveying emotions and are perceived as natural for the human observer. Animated faces without or with a rather simplistic wrinkle generation may be able to convey what emotion is supposed to be shown, but such a face will not be perceived as natural. The smoothness appears too artificial and unnatural.

In [3] and its extension [12], a fairly convincing method for muscle and skin simulation was presented, with the advantage of being computationally cheap in comparison to several other methods. The muscles for a face need to be positioned by hand, which is a time-consuming and difficult process. However, from then on, the calculations for the muscle contractions and skin deformations are rather simple to perform. This is made possible by the absence of springs and by muscles which are not anatomically fully correct but are sufficient for the simulation. Another big advantage is that this approach works for every face in the same fashion, without a need to adjust anything besides the facial muscles for a new face.

As mentioned above, Virtual Reality is one of the application fields for facial animations and is the big newcomer. With its recent boom, a host of applications has been developed and is still in development, while the same can be said about VR glasses like the Oculus Rift [29] or the HTC Vive [26]. Given the technical possibilities offered by VR, the immersion is far greater than with "conventional media". The user feels to be in the middle of the scenery and is able to observe it in a natural way with head and eye movements, and not with rather unnatural input via consoles, keyboards or a mouse. Furthermore, the user can even interact with the scenery, often with motions much more truthful to real life than via aforementioned input devices.

However, right because of this greater immersion, it is very important to not allow the setting to break this immersion, which may happen easier with elements looking odd or unnatural. An unrealistic face of a character interacting with the user in this manner most likely can be counted among destroyers of the suspension of disbelief.

1.2 Research Question and Problem Description

The research question of this master’s thesis is: Is it possible to enhance the work of [3][12] in a way to use it with more modern technologies in order

to be usable for a VR setting with focus on a virtual conference room, and if yes, can it be improved?

The work presented in [3] was written in C++ as an OGRE application. OGRE did not offer any VR support at the time this thesis was written. Thus, a new environment needed to be found, which would allow these facial calculations to be run also as a VR application. Furthermore, to allow greater reusability, the mathematical calculations needed to be separated from the presentation itself, preventing the need to extract the mathematical aspect of this project another time in the future.

While the performance of the OGRE project was satisfactory, it was needed to determine whether at least the same performance could be kept, while an improvement in speed, believability of the face or both, was desired.

The idea behind the VR conference room is that a virtual conference room would be created, in which the user would partake in a conversation with other people. There would be two advantages over a video conference. The first advantage would be the enhanced experience. Instead of just seeing the conversation partner on a screen, the user would have the feeling to physically be within the same room as the other involved party or parties, giving an enhanced feeling of closeness. For this reason, it is important to have believable faces with believable facial expressions, lest the experience be perceived of lower quality, perhaps even inferior to classic video-telephony.

The second advantage would be the greatly reduced amount of data, which needs to be exchanged between the partaking parties. Instead of encoding, sending and decoding large video information, the virtual conference room would merely send parameters for facial expressions. This handful of values would then be taken as input for the communication partner, which would locally calculate the facial expressions. Naturally, the audio component would not be affected.

The creation of a face mesh was not a part of this thesis. While writing this work and the application accompanying it, the author made use of provided face meshes. Likewise, the definition of muscles is not done in the scope of this thesis - these were, again, provided.

However, it should be possible to work not only with a predefined set of faces, but instead with any given face. The results of course may vary, depending on the quality of the face.

2 Related Work

A good overview of the various approaches to facial animation and modelling has been laid out in [8], classifying them into several categories. However, there the focus was mainly on the modelling of faces, whereas skin wrinkles are given less attention.

2.1 Shape Interpolation

The oldest approach is shape interpolation, which also goes by the names of blend shapes or morph targets. As Parke describes [17], this technique can be accurately compared to classic animations, in which a head artist draws the key poses for a character, and then passes these on to an assistant, who draws the frames in-between. With Shape Interpolation, the artist creates the key poses, while the frames in-between are calculated.

The artist manually repositions vertices of the face to create facial expressions. These deformations of various parts of the face are stored. Each necessary facial action, like raising an eyebrow, pulling the corner of the mouth etc., needs to be considered in this process. With these deformations, blend shapes are created, to which certain weights are assigned. To obtain a facial expression, the face in its neutral state is deformed by using the blend shapes, making use of the weights [3][12]. The simplest way of calculating these frames between the neutral face and the desired facial expression, is linear interpolation [2][19], although cosine interpolation [23] can be used as well, as well as a host of other options.

The big advantage of shape interpolations is that, once all preparatory work has been set up, they are fast, and it is easy to create primitive facial animations with them. The disadvantages however are that either an artist must sculpt a face, or a face needs to be generated via live capturing. The key frames need to be created by an artist as well, which is a tedious and unforgiving work. This work is made harder the higher the resolution of the face is, since the number of vertices, which need to be repositioned, grows. This means that shape interpolation depends a lot on an artist's skill. Moreover, this process needs to be done for every used face [3][8][12].

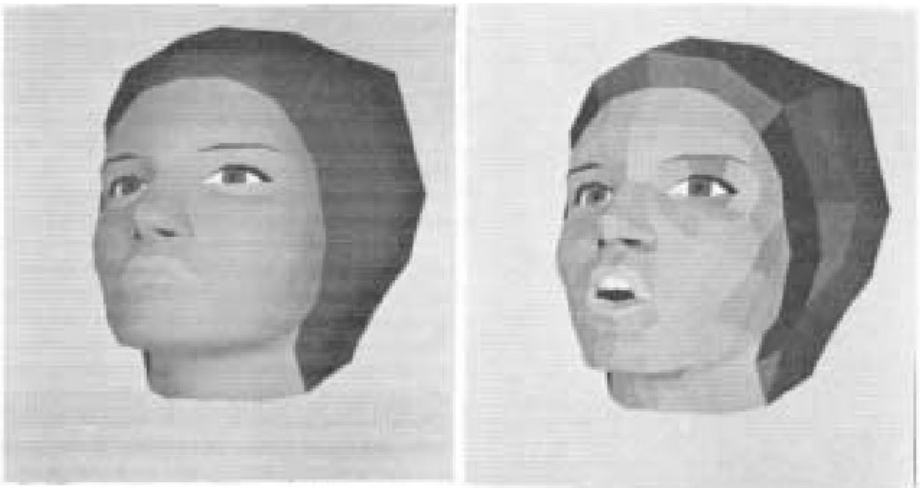

Figure 1 Bend shapes by Parke [17].

Shape interpolation is still commonly used. Most animation software like Maya or Blender offer this technique, with shape interpolation having seen use in blockbuster movies like Lord of the Rings or Star Wars [8].

2.2 Free Form Deformation

The approach of free form deformation (FFD) attempts to overcome some of the mentioned drawbacks of shape interpolation. The face is approximated by a simpler shape, which is embedded in a three-dimensional control point lattice. This control point lattice has several control points, via which the model can be deformed by changing the positions of the control points. Usually this lattice is of a cubic shape [20]. However, the method of extended free form deformation uses a cylindrical lattice for additional flexibility of shape deformation [7].

Rational free form deformation adds weight factors to the control points, allowing to deform the face via changing the weight factors instead of the positions of the control points [21].

In [11] a combination of FFD and MPEG-4 is proposed, making use of the Facial Animation Parameters and Facial Definition Parameters, which are supported by that video standard. That way not only the generation of facial expressions was achieved, but also the generation of different faces by deforming a base face.

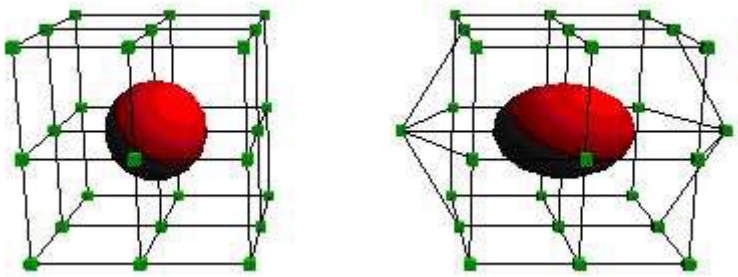

Figure 2 Free Form Deformation. A cubic control lattice and an embedded object can be seen, as well as the effects of manipulating the control points.

The main advantage of all three variants of this approach is that the deformation of the face is independent of the specifics of the model surface, given that it is completely calculation based. Thus, a higher resolution of the face does not mean more manual work. The disadvantage however is that no precise muscle and skin simulation can be achieved, given that only the surface is being manipulated [3][8].

2.3 Performance Driven Facial Animation

An entirely different route is taken by the approach of performance driven facial animation. For this approach, an actor is needed, whose facial expressions are recorded with the help of markers on the actor's face. The recordings of the marker movements are then translated to a face model, which has corresponding markers. Since these systems often run in real-time, the actors can watch the results of their work immediately, thus allowing them to adjust their facial expressions for more desirable results.

In [4] there can be seen an implementation of performance driven facial animation. Three optical properties were defined for the creation of facial expressions: course scale properties represented the movements of muscles (pulling a lip corner etc.); spatial scale properties were wrinkles created by skin compression; and fine scale properties were small details of the skin (freckles, pores, etc.). With commercial scanning software the face model was created at a resolution of 500k-700k vertices, while another scanning system, consisting of six cameras, was used to track facial motions. Facial markers, in the form of 80 to 90 blue dots, were painted on the face of the actor. With this setup, the facial expressions could be tracked and mapped to the face.

However, in order to also generate wrinkles, bright colours needed to be applied at the actor's face, so that the contrast would even out more.

Figure 3 Tracking of facial expressions and wrinkles [4].

In the extension [5] of [4], a data driven approach for the creation of wrinkles was proposed. Around 100 handling vertices were defined, which could either be matched with tracking data, or could be manipulated by an artist. The wrinkles were no longer created in real-time but were instead taken from an example database, filled with example poses of the actor.

Position differences of the handle vertices were captured during the facial expression performance of an actor. These position differences were then used to define a set of strains, with which skin compression would be represented and which would be used then to calculate wrinkles. The shorter the distance between the vertices, the more pronounced the wrinkles become.

The results of [4] and [5] are very satisfactory.

To obtain performance driven facial animation of higher quality however, facial motion capture is needed. For this a high number of cameras are needed, which track the facial expressions of an actor, which are then either converted into blend shapes or used to drive muscle simulation [8][12].

While this is not commonly used for real-time applications, it is an often-used method in the movie industry, although it has also seen some use in the video game industry.

A good example can be found in "The Digital Emily Project" [1], which had the goal to create a synthetic face of such a quality, that it would easily be mistaken for a real one. A female actor was placed into a lighting cage with 156 LED lights and 15 cameras, while 40 small markers were drawn on her face with a make-up pen. The actress performed 38 facial expressions, which were based on Paul Ekman's Facial Action Coding System [10].

Multiple photos were taken of the face, capturing the face and its skin details, down to the stretching of skin pores. These photographs were then converted into different texture maps (like specular maps), storing the amount of light reflections, as well as normal maps, storing fine skin details. Each facial scan provided a face model with approximately three million polygons.

However, from these face models, lower resolution meshes of about 4000 polygons were created. These lower-resolution meshes were to resemble the neutral face positions and naturally lost information due to the downsampling, which is why the texture maps were necessary in order to be able to preserve that information. Then blend shapes were created from the face scans, using the marker points on the actress's face.

Finally, the actress could drive these blend shapes by a video performance. The resulting video showed a phenomenal result.

The big disadvantage, that must be noted for this approach, is that it has a cost-intense setup.

2.4 Muscle Models

The vector muscle model developed by Keith Waters was another approach in 1987 [22]. The idea was to approximate the behaviour of facial muscles, using merely two muscle types for this: the linear muscle (sometimes also called parallel or vector muscle) and the sphincter muscle. The linear muscle, which is used for most of the facial muscles, consists of two parts – a surface part, which is pulled to the fixed point at the skull. Despite the not overly complicated principle, a good variety of emotions for a face could be created.

The linear muscle is conically shaped, with the tip of the cone being the point fixed at the skull, while the wide base of the cone is the surface part. To define such a linear muscle, two points are needed: one for the fixed endpoint at the skull, and one for the middle point of the base of the cone. Furthermore, an angle is needed, which defines the width of the cone, thus also the area of influence.

The sphincter muscle is of an ellipsoid shape, pulling towards the centre. The sphincter muscle is used only for the circular muscles around the eyes and the mouth. To define a sphincter muscle, a centre point is needed, as well as three radii for the ellipsoid.

The advantage of muscle simulations like Waters' vector muscle model is for one their independence of the resolution of the model, and on the other hand the ease of adjustment for different faces, since the facial expressions are not surface based.

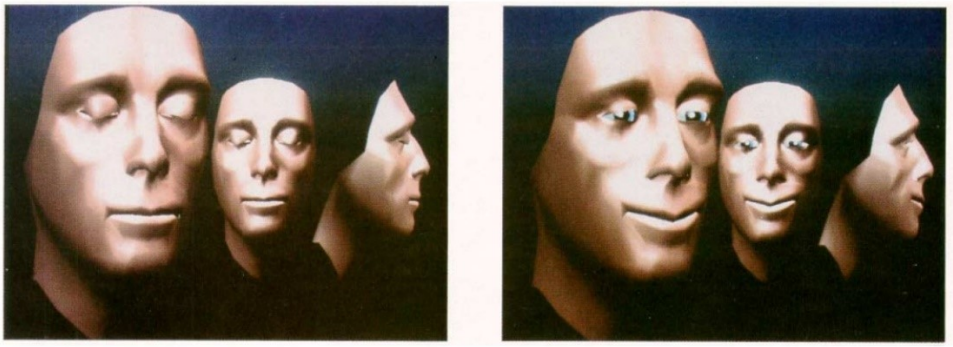

Figure 4 A face in the neutral position and displaying happiness [22].

The disadvantage of this approach is the time consuming and unforgiving task of positioning of the muscles, given that already minor mistakes can lead to very unrealistic deformations. Often it is a tedious task of trial and error.

A method similar to Waters' model was used for Pixar's short animated film "Tin Toy" in 1987 [18].

An improvement on Waters' work was presented in 2003 [6]. The main concern of this work were the unrealistic vertex deformations in cases where multiple muscles influence the same set of vertices. The entire head model was divided into eleven regions, and it was determined, that more expressive regions like the forehead need more vertices than rather static regions like the back of the head. Furthermore, another pseudo muscle was introduced to simulate jaw rotation.

The problem with multiple muscles affecting the same vertices, which [6] set out to rectify, was that vertices could leave the zone of influence of one of the affecting muscles, which would then cause abrupt changes in deformation, producing unrealistic results. To combat this occurrence, parallelism for the muscle contraction was suggested for the muscle contraction, thus not applying the full contraction forces of each muscle on top of each other, but rather applying smaller forces until no contraction force is left anymore. This approach turned out to be effective and produced more realistic deformations in cases where several muscles influenced the same vertices.

[6] also dealt with the question of wrinkle generation in combination with Waters' Muscle Model. The assumption is made, that muscles align with the skin. Thus, a certain number of wrinkles is predefined for each muscle, as are their heights. After applying muscle contraction, the wrinkle amplitude is calculated and added to the force repositioning the vertex. This rather simple technique adds a bit to the achieved realism.

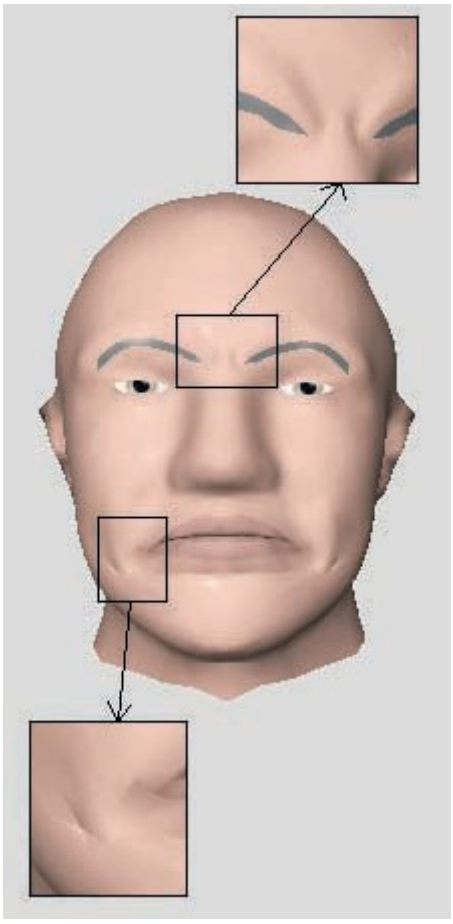

Figure 5 Application of [6], with wrinkles.

Waters and Terzopolous [24] later introduced a more complex muscle model, called layered spring model. It was designed to model anatomical facial features, which are implemented as a three-layer model and represent the bone layer, the fatty tissue layer (also called "dermis") and the skin layer (also called "epidermis"). The face is to be approximated by a point lattice,

with the points being connected by springs. Mimicking the properties of actual human skin, the points of the outer layer (skin/epidermis layer) are connected with rather stiff springs, offering moderate resistance to deformations. The middle layer (fat tissue/dermis layer) points are connected with highly elastic springs, making them highly deformable. The inner layer (bone layer) points are connected with highly resistant springs, allowing for very little deformation. Similarly to the linear muscles in [22], the layered spring model defines muscles with a point attached to the bone layer and another one attached to the skin layer. All muscle contractions are computed and the points moved, depending on the weighted sum of their influencing forces, for the animation of the face model.

In [3][12] the approach of [22] was taken a step further. Likewise an approach of muscle simulation which deforms the surface, one more muscle type was added – the rotation muscle. This muscle type was deemed necessary to perform the opening and closing of the lids, whereas using a linear muscle produced unrealistic results, giving the appearance of the lids being drawn inside the head, and for the rotation of the lower jaw. However, given that the shapes of the areas to be rotated varied greatly, two subtypes were devised, namely the lid muscle, and the jaw muscle. The difference between these two types is mainly the shape, though in terms of calculation they are almost identical. The lid muscle is, like the sphincter muscle, an ellipsoid, which means that the same parameters are needed to define it. The jaw muscle has the shape of a cube, meaning that an origin point is needed, as well as length, height and width. The muscles were based on the FACS model by Ekman [10].

To create believable wrinkles, a solution for the skin was needed. Soft-body simulation, commonly used to simulate cloth, was adapted for the skin simulation, with a few minor changes. Unlike cloth however, the human skin does not react that noticeable to external forces, like gravity or wind. The skin should therefor not hang loosely. Given that this work focused on the creation of expressive wrinkles, the influence of external influences was dropped entirely.

A toolbox was created to showcase the abilities of this approach, which shows satisfactory results.

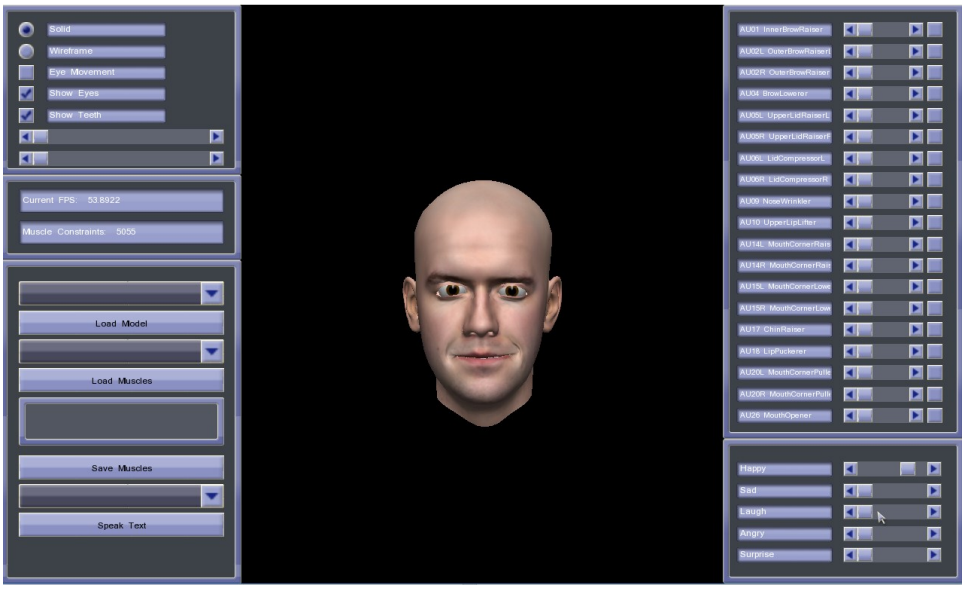

Figure 6 The Facetools toolbox, showcasing [3][12].

In [13] a similarly complex face simulation can be found with the aim to build animated, anatomically correct head models. Unlike [24], a five-component model was used. The first component was the skin surface, represented by the triangle mesh. It was built with the idea in mind to give more expressive regions of the face more polygons (see also [6]). Under the skin layer there is a layer of muscles, which is modelled as an array of fibres with the ability to contract in a linear and circular way. The next layer was built around a model representing the human skull. This is used only for the initialisation of the muscle layer and is not present during the animation. These three components, which form layers similar to [24], were then connected through a mass-spring system, with each vertex being assigned a certain mass and being connected to the underlying muscles by springs. The next component is represented by separate models for eyes, teeth and the tongue. Finally, during the creation of the head model, landmarks were placed on the skin surface, with corresponding landmarks also placed on the skull. The reason for this was to generate an offset between skin and skull, which would be maintained also during animation.

In order to animate the face, the contraction values of the muscles are changed, and the resulting forces directly applied to the skin mesh. Given that the muscles are automatically calculated from the space between skull and

skin, a recalculation after the skin surface deformation is used to visualize their contraction.

2.5 Wrinkle Creation

Wrinkle maps are an approach to generate wrinkles on animated faces and are commonly used in video games. In a similar way as normal maps, these wrinkle maps are laid over the texture of a face when the wrinkles should appear. By regulating the transparency of the texture, a smooth transition can be achieved, preventing a sudden appearance of wrinkles.

A wrinkle map is usually created by an artist during the modelling process of a character. A high-resolution sculpture is built first, from which later a low polygon version is derived. The high-resolution sculpture however contains such details, that it can provide wrinkle information of the skin, which in turn makes it possible to calculate the wrinkle maps from it.

However, this approach makes the wrinkle maps dependant on the character. In order to mitigate this problem, an easy wrinkle map acquisition technique was devised by [9], obtaining the information needed for wrinkle maps from a real person and mapping them to any mesh character.

The same face is recorded in different poses, with the wrinkles being calculated by the illumination differences on the face. To obtain the best possible pixel matches between two poses, a deformation algorithm was developed, which makes use of manually placed markers. On top of that, Gaussian smoothing is performed, with the aim of reducing noise in regions of interest, which are marked with a simple painting tool. The wrinkles are then calculated by approximating the light source as coming from the camera direction, while also assuming that the skin is a diffuse surface. Finally, via interpolation the marked space is mapped to the equivalent texture space of the character mesh, making once more use of manually placed markers.

Figure 7 Wrinkle acquisition by calculating the illumination differences.

It should be noted that this approach serves only the acquisition of wrinkles, but not of a whole face.

A solution for creating wrinkles by mesh deformation was presented by [14]. A planar wrinkle curve is defined by an origin point, a target point and the distance between these endpoints. By pushing the origin point towards the target point, wrinkles are gradually created. Control points can be defined between the origin and the target point, each of them representing a wrinkle to be created.

When the origin point is pushed towards the target point, the control point positions are recomputed with the aim to keep the rest length of the planar wrinkle curve constant. That way the surface is bulging.

Using this approach is quite straightforward. The control curve needs to be drawn over an arbitrary region of influence, no specific prerequisites need to be fulfilled. The curve anchors to the underlying mesh, with all vertices in the influence zone being attached to the nearest control point. The vertices are now manipulated depending on their nearest control point.

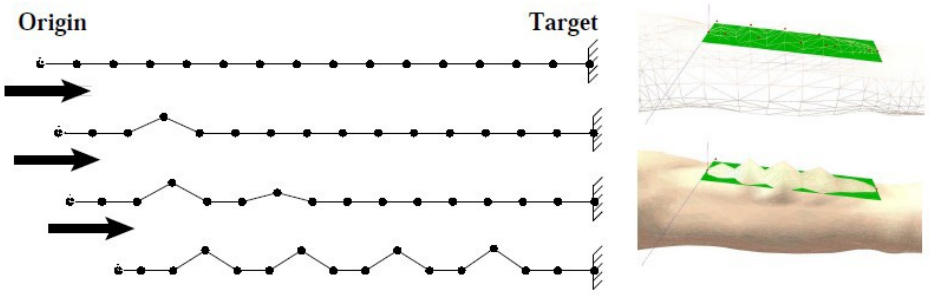

Figure 8 Wrinkle curves explained and in action.

3 Theoretical Discussion

Basing this thesis on [3] [12], the main approach and structure was found to hold great merit. The task at hand can be split up into the skin simulation and the muscle simulation, with the muscle definitions being based on the Facial Action Coding System [10].

This approach has several advantages. Compared to most other approaches laid out in the Related Works section of this thesis, it is not computationally expensive and does not require much preparation, save for the creation of the face mesh and the muscle data. It is also rather simple.

Additionally, by splitting the task into a skin and a muscle simulation, which are independent of each other, a great measure of exchangeability is achieved, which may prove very fruitful for eventual future work in this field.

Mathematically this thesis follows [12] more closely, since a number of improvements and corrections were undertaken by the authors after [3]. A few improvements are included as well, though this thesis largely follows the paths laid out by the mentioned works.

3.1 Skin Simulation

In order to be able to create a realistic animation of facial expressions, an appropriate skin simulation is needed. This simulated skin must not only be respondent to forces generated by facial muscles, but it must at the same time create wrinkles, so that believable facial expressions can be created. As described in the **Related Work** section of this thesis, there exists a number of solutions for this problem. Some of them, however, are computationally expensive, while others require a lot of manual preparation work for each face.

[3, pp. 17-22] points out that there is an object behaving somewhat similar to skin - cloth. Cloth, like skin, needs to wrinkle when compressed in order to look believable. For a, at least visually, correct simulation of cloth, jelly, or other soft bodies, an aptly named physics simulation called soft body simulation [15] is needed.

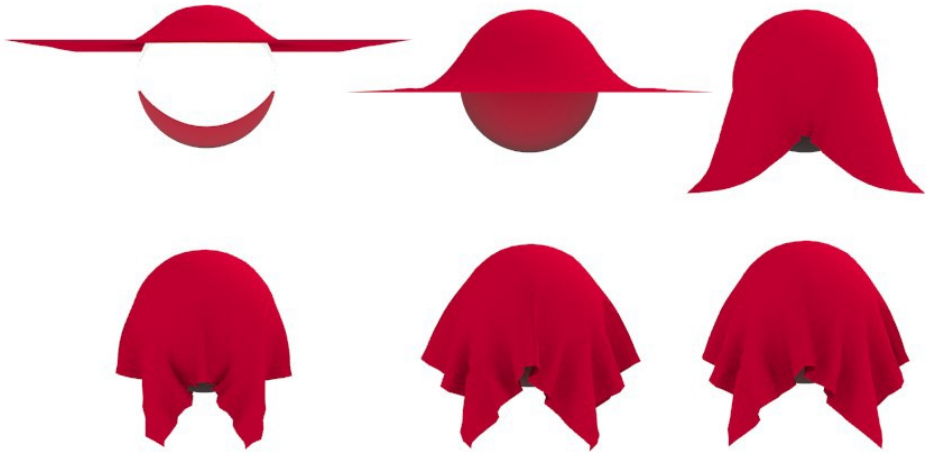

Figure 9 Cloth simulation in Blender [3, p. 21].

Soft body simulation differs in a number of points from solid body simulations. Forces which are applied on a solid body, move it as a whole. They are not elastic, meaning that they do not deform as a reaction to forces, nor do they deform upon collision with another object or let that other object pass through.

Soft bodies on the other hand behave quite differently. A soft body can very well be deformed or give in to a force or another object upon collision, meaning that they are elastic. This is achieved by treating every single vertex in the mesh of a soft body as a separate entity (or, indeed, a solid body), each of them being valid targets for forces to be applied on.

However, the description of soft body simulations above is not yet complete. Soft bodies need to be able to give in, but not allow other bodies to actually go through them. Furthermore, once a force stops affecting the soft body, it should move back into a rest position in order to truly fulfill the requirements of elasticity. To achieve this, it is common to use various springs and constraints.

Springs usually include structural springs, whose task it is to keep the horizontal and vertical distances between vertices, diagonally placed shear springs, which prevent the mesh squares from forming flat diamonds, and bend springs for the bending stiffness of the soft body [12, p. 2]. These springs react to external forces, like gravity or collisions with other objects.

Constraints are another element in this model, which generate forces to keep certain characteristics of mesh vertices or edges and which need to be

fulfilled at any given time. If they are not fulfilled, they generate a force to counter this state, until they are fulfilled. Most commonly used are point constraints, which keep a vertex fixed at its place (for example the part of the flag fixed to a flagpole), or linear constraints, which demand that the edges between vertices maintain their rest length.

The approach described above works very well for cloth simulations. In the case of skin simulations, a few adjustments need to be made. Since the scope of [3] was to create expressive wrinkles, which is also the goal of this thesis, external forces, for example gravity, were ignored. As pointed out in [3, p. 22], for aging skin this approach may need an update as it does indeed hang loosely, but for now this is out of scope. By eliminating external forces, the need for shear springs is not given, since there is no force which actually could flatten the squares to diamonds.

The linear constraints in fact already do what would be done by structural springs, which leaves no reason to keep them around.

Point constraints, as described above, are not usable for the skin simulation, since they fixate a vertex to its position, preventing it to ever leave it. Instead, a position constraint was developed, which tries to keep a vertex at its original position, but does not prevent it from moving away. Instead, a force is generated, which pulls the vertex back to its original position in case that it moves, which also satisfyingly implements elasticity [12, p. 3].

And finally, bend springs would not work well on faces, which may very well be asymmetrical. They would produce too different results for each half of the face [12, p. 4].

The described skin simulation, which is being used and adapted by this thesis, thus makes use of no springs at all. Instead, three different kinds of constraints are used: the aforementioned linear and position constraints, as well as the muscle constraint, which simulates the forces generated by the various facial muscles.

3.1.1 Common Math for Constraints

All of these constraints work essentially the same way, which can be shown with this base formula:

$$f = M \times Dir$$

With f, the force generated by a constraint is denoted. M means the magnitude of the force. Dir denotes the force direction.

Every force generated by the constraints affecting a particular vertex, needs to be added to a common net force. This net force is added to the vertex position after all constraint forces of an iteration had been calculated. However, some weighting must be introduced as well, which is done by simply dividing the resulting net force by the number of forces affecting the vertex.

$$V_{curr}{}' = V_{curr} + \frac{\sum_i f}{i}$$

This does not apply to muscle constraints though, for which no weighting must be introduced. The reason is that the muscles affecting the same vertex would otherwise cancel each other out, even if inactive.

Obviously, the constraint forces not affect each other within the same iteration during their calculations.

3.1.2 Linear Constraint

The linear constraint is a constraint which tries to keep mesh edges at an optimal rest length. It is thus the only constraint, which affects two vertices at the same time, given that an edge in the mesh must have exactly two vertices, which it connects.

As mentioned, the aim of this constraint is to keep the mesh edges at their optimal rest lengths. As the optimal rest length, the mesh length during initialization is taken. At this time, the face is still in a neutral, expressionless position. Whenever a force now moves one or both vertices of the linear constraint, the length of the edge is changed. The vertices will either move closer together, or drift further away. At this point the linear constraint develops a force, which counters this development, pushing the vertices either further away from each other when they got too close, or pushing them closer together when they drifted away from each other.

The role of the linear constraint is to actually create the wrinkles by preventing the vertices to get too compressed, which would mean that they would be allowed to overlay each other, which in turn would mean that surfaces would stay very smooth. Since human skin does not work that way, it has to evade the force to the sides. By enforcing the initial distance between

the vertices despite other forces affecting them, the vertices are bound to at some point evade to the side in order to fulfil this constraint, thus giving the skin its wrinkles.

Every mesh edge of the face must have exactly one linear constraint. Every vertex in the face must have a number of linear constraints. They must have at least two linear constraints and thus two edges, since otherwise mesh triangles could not be formed, or three when working with mesh rectangles. While there is no real upper limit, it appears to be rare to have more than eight linear constraints affecting it [3, p. 23].

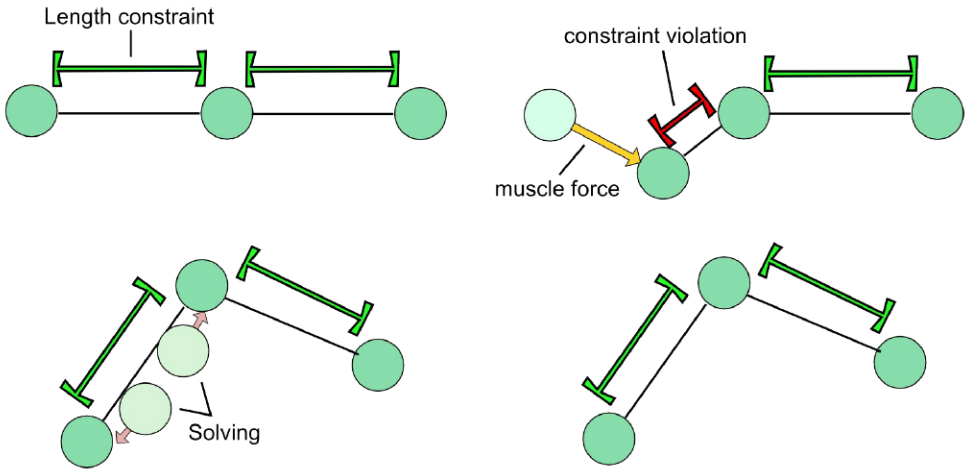

Figure 10 Linear constraints in action [3, p. 24].

3.1.2.1 Calculation of the Linear Constraint

For calculating a linear constraint, the initial and current positions of the two vertices connected by a mesh are needed, which will be called $V1_{init}$ and $V2_{init}$, as well as $V1_{curr}$ and $V2_{curr}$ respectively.

To calculate M for a linear constraint, the initial distance between the two vertices needs to be subtracted from the current distance:

$$M = Dist_{curr} - Dist_{init}$$

As a quick help, the formula for calculating the Euclidean distance between two vectors - the coordinates of a vertex ultimately are a vector - is presented here. The presented example works with the initial positions, though, obviously, this works the same way in all cases.

$$Dist_{init} = \sqrt{(V1_{initX} - V2_{initX})^2 + \left(V1_{inity} - V2_{initY}\right)^2 + (V1_{initZ} - V2_{initZ})^2}$$

The bigger the difference between $Dist_{init}$ and $Dist_{curr}$, the higher M will be, which translates to a stronger developed force. If M is a positive value, the two vertices of the mesh edge have drifted away and will, as a result, be pushed towards each other. If M is a negative value however, the two vertices have actually come closer to each other and will thus be pushed away from each other.

The direction Dir of the force is calculated from $V1$ to $V2$. For this $V1$ needs to be subtracted from $V2$, with the result being normalized. Naturally, the current positions are to be used.

$$Dir = \frac{V2_{curr} - V1_{curr}}{|V2_{curr} - V1_{curr}|}$$

In order to normalize a vector, each of its components need to be divided by the vector length. If the length happens to be zero, Dir becomes a zero vector.

$$V_{length} = \sqrt{{V_X}^2 + {V_Y}^2 + {V_Z}^2}$$

With M and Dir calculated, they can be put into the constraint base formula. There is however a peculiarity, which needs to be taken into account.

$$f_{lin} = 0.5 \times M \times Dir$$

Given that this generated force needs to push two vertices around, it needs to be multiplied by 0.5, since the whole force needs to be split evenly between both vertices.

Another peculiarity resulting from the fact that two vertices are affected, is that the generated force needs to be added to the net forces of both vertices.

$$F1_{net} = F1_{net} + f_{lin}$$

$$F2_{net} = F2_{net} - f_{lin}$$

Given that the force developed by a linear constraint needs to push the vertices into opposite directions, it needs to be added to the net force of one vertex, while subtracted from the net force of the other. It is important to keep in mind that the force needs to be added to that vertex, which has been used as the starting point for the direction, and consequently subtracted from the other. Otherwise the vertices would drift even further apart, if M is positive, or come even closer, if M is negative.

3.1.3 Position Constraint

The position constraint aims to keep a vertex at its original position. This means that whenever a vertex is moved away, the position constraint will develop a force to push the vertex back to its initial position. The further the vertex moves away from its original position, the stronger this push will be. That way the position constraint also keeps the model from deforming in completely unrealistic ways, i.e. by widening the face with a smile – the force generated by the position constraint prevents this.

The reason behind the position constraint is to simulate the elasticity of the skin. Human skin can be deformed but will always return to its rest state once the force which deformed it dissipates. Skin can also not be stretched endlessly, which is another point simulated by the position constraints.

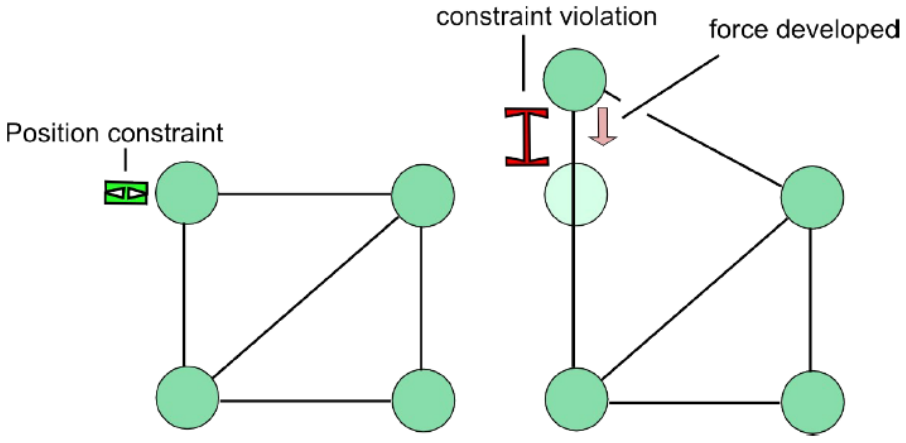

Figure 11 Position constraints in action [3, p. 23].

To calculate a position constraint, merely the force direction as well as the skin elasticity are needed. The direction is being calculated from the current to the initial position of the vertex, with the desired rest direction being zero,

meaning that current and the initial position are equal. The skin elasticity is a constant, which is determined before the actual simulation and does not change anymore.

Every vertex of the face must have exactly one position constraint.

3.1.3.1 Calculation of the Position Constraint

The only things that are necessary to be able to calculate the position constraint for a given vertex, are the initial position V_{init} and the current position V_{curr} of that vertex, as well as the skin elasticity, as mentioned above. The skin elasticity is designated α. For α the following applies:

$$\alpha \in [0, 1)$$

Obtaining M for a position constraint is easy, as it equals the skin elasticity. This means that the higher the value of skin elasticity, the stronger the force developed by the position constraint will be, while a lower value will allow further stretching, since the counter force will not be that strong.

$$M = \alpha$$

Dir is calculated simply by subtracting V_{curr} from V_{init}, since the vertex shall move from its current position towards its initial position. Unlike for the linear constraint, the resulting vector is not normalized.

$$Dir = V_{init} - V_{curr}$$

The formula for the position constraint can thus be written also in the following way:

$$f_{pos} = \alpha \times (V_{init} - V_{curr})$$

3.1.4 Muscle Constraint

The muscle constraint is what actually gets things into motion. It is the constraint responsible for letting muscle contractions take effect on the face.

Each muscle has an area of effect, with which it is determined during initialization, which vertices are being affected by the muscle. Upon contraction, all affected vertices are moved. This causes the position and linear constraints to stop being satisfied, in turn making them generate forces

to get back into a satisfying state. Once the muscle stops contracting, the affected vertices return eventually to their initial positions.

Every vertex can be affected by multiple muscles. Every muscle can affect an infinite number of vertices.

3.1.4.1 Calculation of the Muscle Constraint

To calculate a muscle constraint, a target position V_{tar} in addition to V_{init} is needed. The target position is the position, to which the vertex would be moved if the force of the muscle associated with the muscle constraint would be unopposed. It is obtained via the muscle calculation, which is done differently for every muscle type.

Dir is being calculated by simply subtracting V_{init} from V_{tar} as the vertex is supposed to move towards the target position. Like with the position constraint, Dir is not normalized.

$$Dir = V_{tar} - V_{init}$$

Since M is always 1 for muscle constraints, the constraint formula can be also written as:

$$f_{mus} = V_{tar} - V_{init}$$

3.2 Muscle Simulation

The facial muscles, which are represented in the muscle simulation presented by this thesis, need to be categorized into different types, based on their shape and effect upon contraction.

Since this work is based heavily on Waters Muscle Model, two of the three types are taken from [22]. These types are the linear and the sphincter muscle.

A third type was introduced by [3][12], namely the rotation muscle. The addition of this type had been necessary, since eye lid and jaw movement could not be handled well with Water's model, with the eye lids for example appearing to be straightly pulled into the skull [3, p. 30], giving an unrealistic look. However, due to the shape difference of the lids and the jaw, this muscle had to be split further into the lid muscle and the jaw muscle.

The result of each muscle calculation, triggered by muscle contraction, is the target position V_{tar}, which is needed for the muscle constraints. The base

for the calculation always has to be V_{init}, as otherwise the results would not be predictable.

The contraction strength is named K, sometimes also referred to as muscle spring constant. In this work it is assumed that the contraction strength is given as percentage in the following form:

$$K \in [0, 1]$$

3.2.1 Linear Muscle

The linear muscle, sometimes also called vector muscle, is the most common type of muscle in the face model. This muscle type is used for all muscles, which pull on a section of the skin surface. The linear muscle is one of the two muscles defined in [22].

The linear muscle consists of two points. The origin point of the muscle is attached to a bone, which means that it remains static during contraction. The endpoint is attached to the skin and is pulled towards the origin point during contraction. Furthermore, around the endpoint an area of influence is created, with all vertices finding themselves in it, being pulled towards the muscle origin point as well.

However, the generated force needs to be dampened for vertices which are too close to the origin point. This is necessary to firstly prevent affected vertices from being moved beyond the origin point and secondly this aids the skin simulation, given that skin, despite compression and evasion, at some point has nowhere left to go. To achieve this task, a muscle falloff needs to be defined, which is a distance shorter than the distance from origin to endpoint. If the distance between origin and affected vertex is equal or less than the muscle falloff, the dampening is triggered.

To define a linear muscle, an origin and endpoint are needed, as is a muscle falloff start and an angle. With the two points and the angle, a sphere segment is created, ultimately being the zone of influence of the muscle.

3.2.1.1 Linear Muscle Calculations

The formula for the calculation of the target position V_{tar} of a linear muscle, is the following:

$$V_{tar} = V_{init} + adv \times K \times rdv \times Dir$$

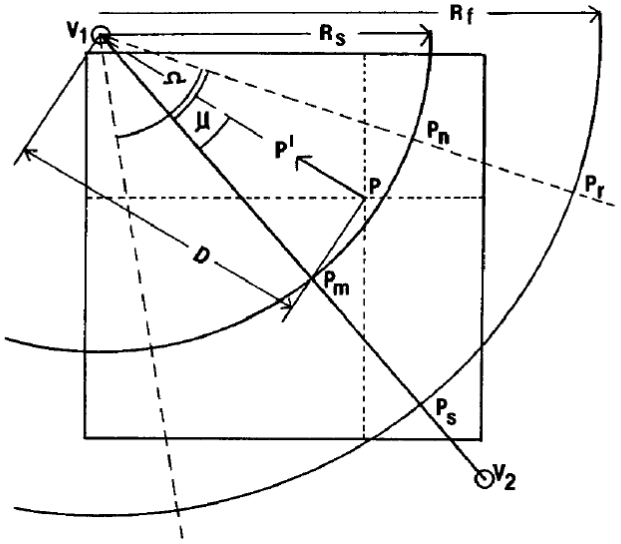

Figure 12 Schematics of a linear muscle by [22]. Note that *V1* is called P_{OP} in this thesis, and *V2* is called P_{EP}. While the drawing suggests otherwise, R_f indeed is $Dist_{OPEP}$. Ω is the whole angle of the linear muscle.

The Angular Displacement Value, in the formula denoted as adv, is a factor which depends on how far off the vertex V is from the line $\overrightarrow{P_{OP}P_{EP}}$. The further away it is, the weaker the developed strenght. It is calculated the following way:

$$adv = \cos \mu$$

The angle μ is the angle opened between $\overrightarrow{P_{OP}P_{EP}}$ and $\overrightarrow{P_{OP}V}$.

The Radial Displacement Value rdv is another factor, which diminishes the generated strength. It simulates the muscle falloff, which means that the closer V_{init} is to the origin point of the linear muscle, the weaker the force has to become. There are two formulas for the rdv, which depend on how far away from the origin point V_{init} is.

If the distance $Dist_{OPV}$ is below or equal R_S, which is the distance between P_{OP} and the muscle falloff start, rdv is calculated the following way:

$$rdv = \cos\left(\left(1 - \frac{Dist_{OPV}}{R_S}\right) \times \frac{\pi}{2}\right)$$

However, if $Dist_{OPV}$ is greater than R_S, the formula changes:

$$rdv = \cos\left(\left(\frac{Dist_{OPV} - R_S}{R_f - R_S}\right) \times \frac{\pi}{2}\right)$$

R_f is the maximal distance the muscle can affect. Thus, it is $Dist_{OPEP}$.

The direction Dir points from V_{init} to P_{OP}, since this is the direction towards which the vertes shall be pulled. It needs to be normalized. Thus, the formula is:

$$Dir = \frac{P_{OP} - V_{init}}{|P_{OP} - V_{init}|}$$

3.2.2 Sphincter Muscle

The second of Waters' muscles is the sphincter muscle. This type of muscle takes care of the circular compression around eyes and mouth, which for example are used for lip puckering.

The area of influence of the sphincter muscle has the shape of an ellipsoid and it pulls vertices towards its center. This center represents the bone attachment of the muscle.

To define a sphincter muscle, its origin point is needed, as well as its three radii, with which the ellipsoid can be formed.

As with the linear muscle, a muscle falloff exists, which prevents vertices from being dragged across the origin point. To achieve this, a set of inner radii needs to be provided, creating an ellipsoid within the ellipsoid. If a vertex is within the inner radii, the muscle falloff is activated.

Furthermore, a scale factor needs to be added as well. This scale factor is needed to enhance the muscle behaviours for their specific areas of use. To give a realistic impression, the effect of the horizontal muscle contraction must have a far less pronounced effect than the vertical one, if the sphincter muscle is responsible for the area around the eye. For the area around the mouth however, it is the exact opposite – the vertical muscle contraction needs to be less prominent than the horizontal one [3, p48].

3.2.2.1 Sphincter Muscle Calculations

The formula for the sphincter muscle is very similar to the formula for the linear muscle:

$$V_{tar} = V_{init} + K \times rdv \times Dir \times S$$

Dir and *rdv* denote the very same factors as for the linear muscle and are calculated the very same way. The Angular Displacement Value *adv* is not needed anymore, while a new Scale Factor *S* appeared.

S is a three-dimensional vector which, when factored in, negates the x-value of the force vector, when working with the muscles around the eyes, while same is done with the y-value when working with the muscles around the mouth. In both cases, the z-value is somewhat dampened. That way the desired effect of having a less pronounced effect of horizontal/vertical muscle contraction, as described above, is achieved.

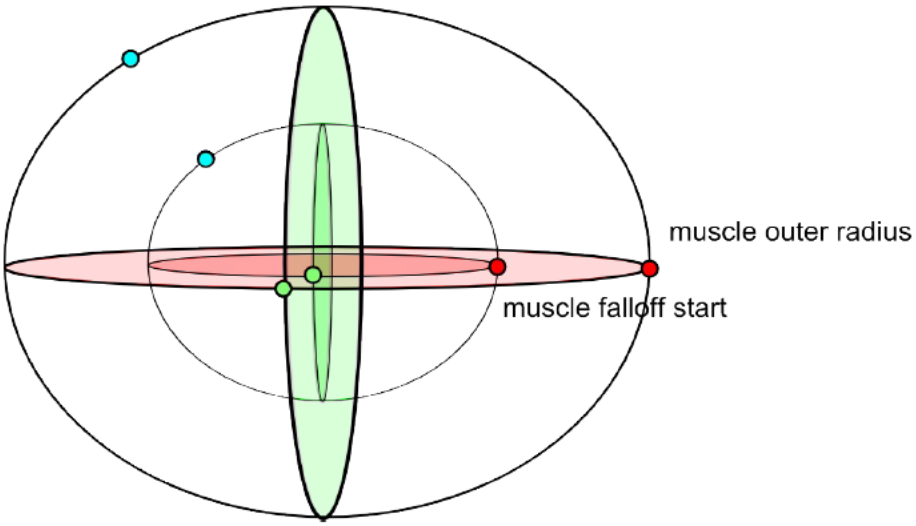

Figure 13 Sphincter muscle [3, p. 48].

3.2.3 Lid Muscle

The lid muscle is a rotation muscle and one of the two muscles, which were added to the model proposed by Waters [22].

As mentioned in 3.2, the problem with simulating the eyelids with linear muscles was that they appeared to be pulled straigthly into the head, giving an unrealistic feel. It was decided to instead let the lid rotate inside.

Since the eyelids can be approximated as a quarter of a sphere, [3] made the decision to make the lid muscle spherical in shape, exactly like the sphincter muscle.

To be able to define a lid muscle, merely its origin point and its three radii for the area of influence, which is shaped like an ellipsoid, need to be provided.

3.2.3.1 Lid Muscle Calculations

The target position provided by the lid muscle is calculated with the following formula:

$$V_{tar} = P_{Joint} + Dir \times m_{rot}$$

The joint point P_{Joint} is calculated by subtracting R_Z, the ellipsoid radius for the z-axis:

$$P_{Joint} = P_{OP} - R_Z$$

Here the direction is calculated as follows:

$$Dir = \left(V_{Init} - P_{Joint}\right)$$

The rotation matrix m_{rot} is the part, where the changes happen. The permitted angle ranges from $0°$ to $90°$, any further rotation is unnecessary and in fact would look unrealistic. This is also where the contraction strength comes into play for the rotation muscles. The angle of $90°$ in m_{rot} needs to be multiplied with K, before calculating V_{tar}. One thing to keep in mind is whether the initial position of the lids is closed or opened, thus if the muscle needs to pull them down or up. For adjustment, it may be necessary to multiply the angle with -1. See **4.5.2.3.** for further details.

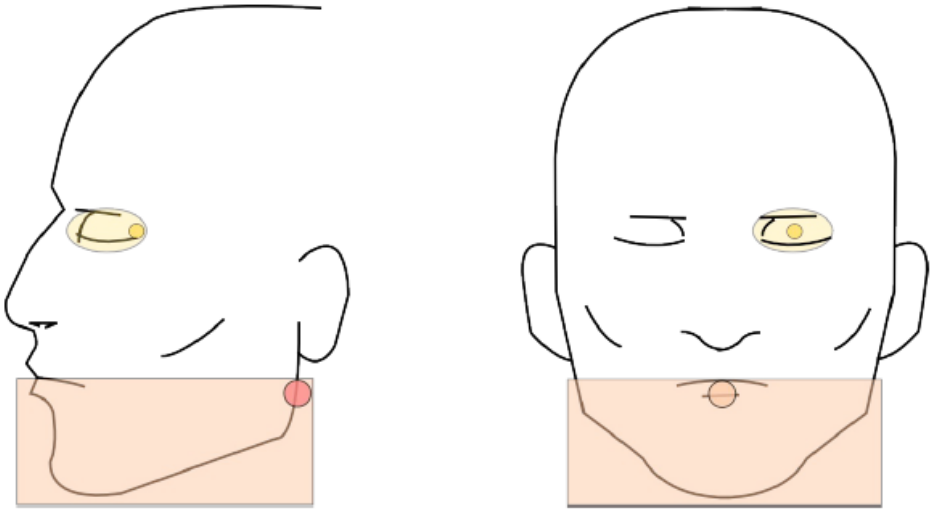

Figure 14 The two types of rotation muscle - the lid and the jaw muscle. Note the positions of their joint points [3, p. 30].

3.2.4 Jaw Muscle

The jaw muscle is the second muscle added to Waters' Muscle Model and is likewise a rotation muscle and working very similarly.

The main difference is that the shape of the muscle is approximated as a cube, which for this purpose is precise enough (see **figure 14**).

3.2.4.1 Jaw Muscle Calculations

Calculating V_{tar} for the jaw muscle is very similar to calculating it for the lid muscle. The formula is:

$$V_{tar} = P_{Joint} + Dir \times m_{rot}$$

As mentioned in the theoretical section for the jaw muscle, its shape is approximated by a cube. A bit peculiar, the origin point of the jaw muscle is assumed to be located centrally on the top plane of the cube. This means that, since the joint point is meant to be located on the middle of the upper back edge on the cube, the joint point is calculated again by:

$$P_{Joint} = P_{OP} - R_Z$$

The direction Dir is calculated the same way as before:

$$Dir = \left(V_{Init} - P_{Joint}\right)$$

However, there is one additional step to be taken when working with the jaw muscle. To give a somewhat more realistic look, the lower lip needs to curve slightly more when opening the mouth. To achieve this, y-position of the V_{tar} needs to be adjusted after calculation. To achieve this, the following formula was used:

$$V_{tarY}{}' = V_{tarY} + \sqrt{\left|L_X - \left(V_{tarX} - P_{JointX}\right)\right| \times \left|L_Y - \left(V_{tarY} - P_{JointY}\right)\right|} \times N$$

The variable L denotes the cube lengths of the jaw muscle, while N denotes a necessary strength multiplier. For N the following formula was found to give satisfying results during experimentations:

$$N = \frac{K}{15}$$

For the jaw muscle it was found that $90°$ mean a too big mouth opening, looking very unrealistic. In trials it was found that around $35°$ are enough to simulate the opening of the mouth realistically. Like with the lid muscle, the angle of $35°$ needs to be multiplied with K to obtain the correct effect. Once again attention must be paid to what the idle position of the jaw is, though it is highly unlikely that it will be open.

3.3 Facial Action Coding System

The Facial Action Coding System (FACS) [10] is a manual for human facial expressions, as well as which muscles are needed to achieve them and how they need to be used. Muscles are made part of Action Units, which define the behaviour of the muscles and their visual impact on the facial expression. An Action Unit combines all muscles needed for the behaviour it characterizes, usually creating small groups of muscles, whereas sometimes it can also be a single muscle. These Action Units can be combined as desired to create any possible facial expression. Altogether the Facial Action Coding System knows a total of 46 Action Units.

The Facial Action Coding System describes each Action Unit verbally, with a set of still images, and a short video clip. The name of each Action Unit describes concisely its function. In addition, each Action Unit is assigned an arbitrary number. Most Action Units are symmetric, influencing both sides of the face equally. Some units, however, exist twice, once on each side of the face. In these cases, the letter 'L' is put before the Action Unit number to denote it being on the left side, or the letter 'R' for the right side.

The description of each Action Unit is split into three sections. The first section is called "Appearance Changes" and deals with all visual deformations of the face, which may occur due to the Action Unit in question, be it coarse deformations like raising an eyebrow, or finer deformations, like wrinkling of the skin. The second section is called "How to" and details how to perform an Action Unit. The third and last section is called "Intensity Scoring". It contains a scoring scale, ranging from A to E, with A denoting the weakest and E the strongest deformation.

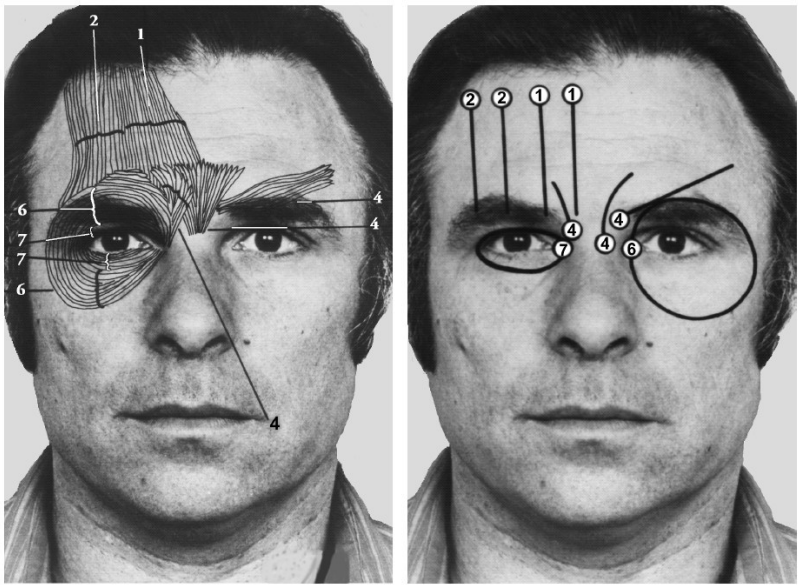

Figure 15 Left: image with anatomically correct muscles. Right: image on muscular actions of an Action Unit [10].

The description of each Action Unit is accompanied by two images (see **figure 15**). One image shows the relevant muscular anatomy for the Action Unit. The muscles are drawn anatomically correct and very detailed. These muscle drawings are superimposed on a photograph of a face, giving a better

understanding of the sizes, locations and shapes of the muscles. The other image shows the muscular action of the Action Unit, which schematically shows how the muscles of the Action Unit affect the face.

The Action Units are divided into upper face and lower face Action Units. The Upper Face Action Units are responsible for the forehead, the eyebrows and the eyelids and are not categorized any further. The Lower Face Action Units are however categorized in five further groups: Up/Down, Horizontal, Oblique, Orbital and Miscellaneous Actions.

4 Implementation

This chapter is dedicated to explaining how the theory laid out in **3.** was implemented, which limitations of the chosen technology were found and what challenges presented themselves during this work.

4.1 Technical Starting Position

In this section the technical starting position is detailed. It is explained why certain tools were chosen, and which data had been provided and was used as a basis.

4.1.1 Unity

One of the requirements for this master's thesis was to port the work of [3] from OGRE to Unity.

OGRE is a very popular 3D rendering engine based on C++ [30]. While it is in the meanwhile somewhat old, with having its beginnings at the end 1999, it is nevertheless still continuously updated and maintained. While it does its job very well, it is quite cumbersome and unwieldy to be used in a project. OGRE needs to be integrated in a time-consuming way into the project to be able to use it.

Unity is a somewhat newer game engine [32]. Similar to OGRE, it has been continuously developed and is very popular in the gaming industry, in particular among indie developers. However, being a game engine, Unity is way simpler to integrate into a project - or rather, to have a project built around it. Despite staying a separate development tool, it has an excellent interface for working with MonoDevelop or Visual Studio (which was the tool of choice of the author). Furthermore, the scripting is done in C\# instead of C++, which has lately seen a great expansion with the removal of platform dependability as well as an increase of speed.

The biggest advantage of Unity however is its readily available support of Virtual Reality, which follows the direction this thesis is pointing in.

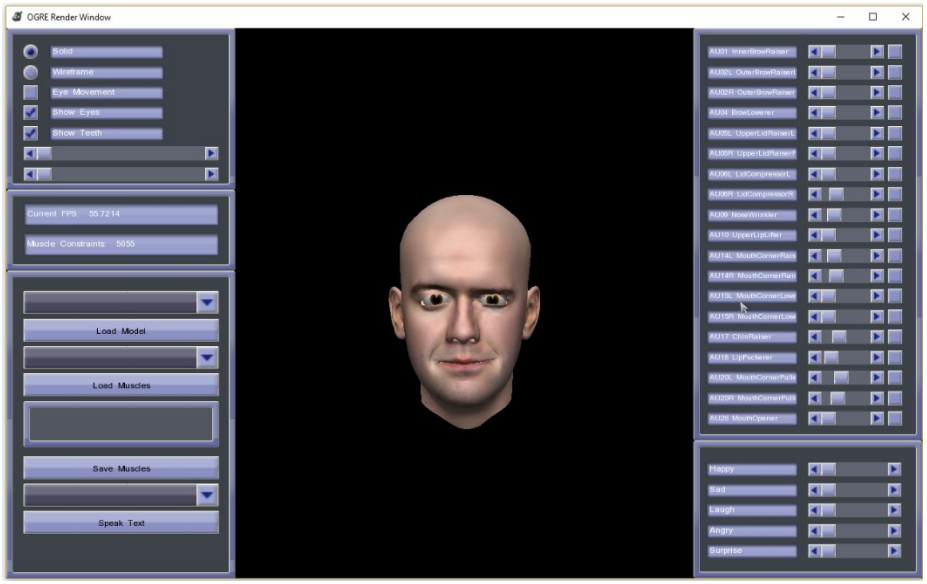

Figure 16 The Facetools toolbox of [3], using OGRE.

Both, OGRE and Unity, continue to be used for various projects, be it for game development or for scientific or experimental applications.

4.1.2 Provided Data

Obtaining face models or determining meta data for muscles, such as positions, is explicitly not part of this thesis. The approaches laid out in this thesis require at least one face model, and all necessary muscle information to be obtained in-beforehand.

4.1.2.1 Faces

Three head models had been provided, with two of them being provided by the Research Group Entertaining Computing of the Faculty of Computer Science at the University of Vienna, while another one was obtained from [27]. By gathering different face meshes from different sources, it could be made sure that the approach continued by this thesis is not dependant on some arbitrary parameters or ways of obtaining a face mesh, nor that the approach is specifically tailored for a specific face.

4.1.2.2 Muscle Data

Muscle data is information which is necessary for initializing muscles properly so that they can work as intended. The data needed varies from muscle type to muscle type and is explained in detail in **4.5.2.**

Determining this muscle data is unfortunately a very time-consuming process, since it has to be determined for each face, manually. This is however necessary, since every face is different and even small errors in e.g. positioning the muscle origin points can have grave consequences.

4.2 Implementational Overview

Accordign to the author of this thesis, one of the major shortcomings of the implementation in [3] was the strong coupling of the mathematical implementation to the presentation model. It was a showcase application with all necessary calculations built in, but it was not readily reusable.

It was thus decided to split the application into two parts: a presentation part, and a calculation part. The presentation part is the Unity application, responsible for the GUI, handling the user input, presenting the results of the calculation on the face mesh, and handling of the eyes and teeth, which, after all, are not part of the skin and muscle simulation.

The calculation part was made as a library in pure C#. While this prevented the usage of predefined Unity classes for vectors, vertices and calculations associated with them, this decision also gives the flexibility to reuse the DLL with whatever application in the future might need it. The downside of this is of course the fact that the aforementioned classes for vectors etc. needed to be implemented for the DLL.

All requests from the presentation part are handled by one controller, which then calls the appropriate methods for the constraints and muscles upon initialisation, contraction and force application. This way a model-view-controller pattern was implemented, with the model being represented by the vertices of the face, the view being the presentational application in Unity, and the controller being the aforementioned controller in the DLL, handling all muscle contractions and force calculations [34].

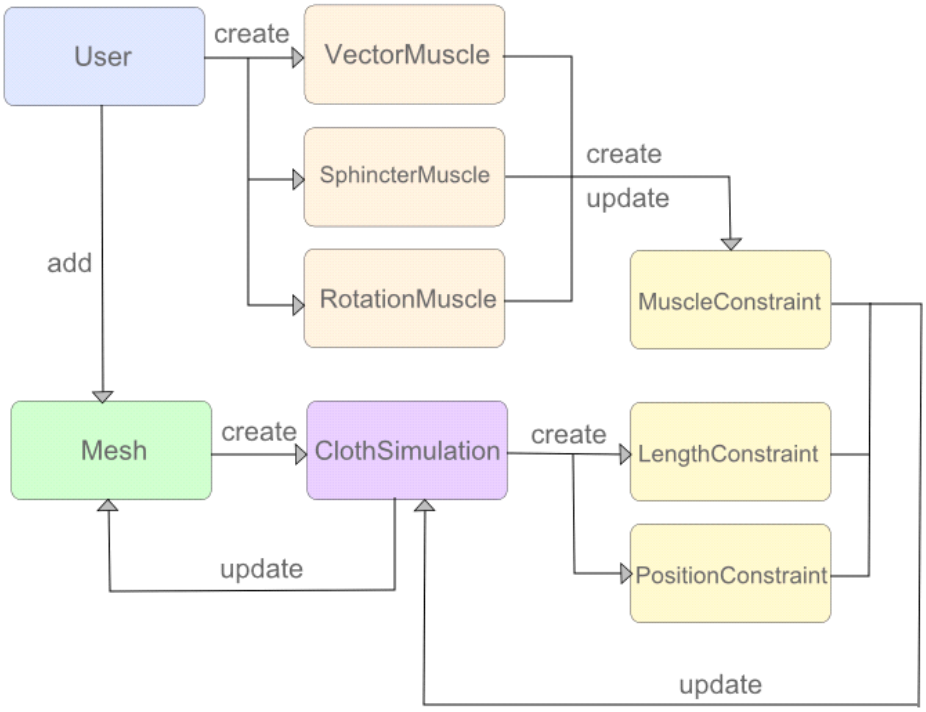

Figure 17 Note that while the implementation principles changed a lot to [3], the idea of how the application works has remained the same [3, p. 39].

4.3 Face Meshes in Unity

The first challenge faced was the usage of the provided face meshes. These were received in the native OGRE mesh format .mesh, which cannot be used by Unity nor Blender. Luckily, with the help of an online converter [28], the mesh could be converted into the Wavefront .obj format, which many mesh editors can interprete.

However, this solution is not ideal, as the converter seems to downsample the mesh, with the resulting face mesh having only slightly over half of the vertices the original has. See **5.1.2.** for further details.

After converting and importing the face meshes into the Unity project, the textures had to reassigned to the meshes as a finalising step. After that, the meshes were imported.

4.3.1 Challenges with Unity

A big issue needs to be kept in mind when working with engines like OGRE or Unity, and that is the handedness of the coordinate system used. OGRE and Unity unfortunately differ here, with OGRE using a right-hand coordinate system, while Unity uses a left-hand coordinate system.

Luckily, Unity handles this issue fairly well with imported models by automatically converting the mesh from right-hand to left-hand, keeping texture intact and without mirroring the face. It goes by unnoticed, without any further input required.

This becomes problematic however for additional external data, which refers to the positions in the face. In the case of this master's thesis this additional external data were the origin points of the muscles and, in the case of the linear muscles, also of their endpoints. These needed to get their x-coordinate inverted in order to really be where they were actually meant to be.

Failing to invert the aforementioned muscle positions does not prevent the calculations from working. The results are in fact even still fairly good, given that the human face is relatively symmetric. However, since human faces are not completely symmetric, minor differences do occur, giving in some cases, when muscles are contracted fully, somewhat unnatural wrinkles.

Another issue that comes up with the usage of Unity is that it by default sets an option called "Weld Vertices" for imported meshes. This option takes vertices, which have the same coordinates, and replaces them with only one vertex. While this makes perfect sense for a game engine, where in most cases those vertices are rather a nuisance and their unification yields gains in performance, for the needs of this master's thesis this was detrimental. Since the skin and muscle simulations are based on the vertices of the model, clusters of vertices in a particular area may actually be desired to, for instance, allow for somewhat finer results of the muscle constraints in very expressive areas, like around the mouth. Furthermore, unifying all these vertices means to also unify their edges, which then would reduce the number of generated wrinkles or their quality, since less linear constraints are generated. Luckily, just by unchecking that option, the welding of vertices stops being an issue.

There is however also a benefit from the Unity conversions. Likewise by default, Unity sets for imported meshes an option called "Optimize Mesh".

This option lets Unity rearrange vertices in order to lower caching times during animation. Since no vertex is actually moved and no edges are created, changed or removed, this option does not affect the calculations and can thus be left checked.

4.4 Implementation of the Skin Simulation

As was described in **3.1.**, the skin simulation is based on three constraints: the linear, position and muscle constraints.

When a face and a muscle set are loaded, the constraints are initialized. For the position and linear constraints, the face mesh itself is sufficient, while for the muscle constraints the muscle data is needed as well. It needs to be kept in mind for creating the linear constraints, that Unity saves mesh edges as triangles in the form of a one-dimensional integer array, with three consecutive values denoting the IDs of the vertices making up a mesh triangle. Consequently, the length of that array must be a multiple of three.

The approach of [3] and [12] demands that every frame the effects of all constraints are calculated over a set number of iterations. However, the author of this thesis found that since the approach demands that for each frame the calculations are started anew from the initial positions of the vertices, the results are bound to be the same as in the last frame. Because of this, it was decided to not kick-off the calculations every frame, but instead do them on demand, which is whenever the contraction strength of a muscle changes, which then sets off a chain-reaction among the constraints. (The DLL responsible for the mathematical calculations does support either approach though.)

The approach of having iterations within such a cycle were found to be very important. They were however tweaked.

In [3] and [12] a common net force F is calculated for each vertex during every iteration by summing up the forces of all constraints which affect the vertex. During the implementation of the ideas discussed in **3.1.**, it was found that the forces of the muscle constraints are suffocated by the other constraints due to the much larger amount of generated linear and position constraints. It was decided to calculate two different, independent net forces.

The muscle constraints must go first, since they are the ones whose results let all the other constraints lose their resting positions. Not following this

order of operations leads to at least the first iteration of position and linear constraints being entirely pointless, since as long as no change took place, they will never not be fulfilled.

Unlike with the position and linear constraints, which need a weighting of forces, the forces of the muscle constraints must not receive any weighting whatsoever. The reason behind this is that otherwise the muscle constraints may very well cancel each other out, even if inactive. Weighting only active muscles also does not give a desirable result either, as the vertices make sudden jumps at the activation of each muscle, since a force active so far all at once is cut in half by a second muscle constraint suddenly affecting a common vertex.

Next come the position and linear constraints with their own net forces. Their task is to counteract the muscle forces. However, since they disturb each other in this task by not letting each other come to rest, vertices are moved to the sides and not directly to their resting positions, which results in wrinkles. The more iterations there are, the closer does the result come to a smoother and more realistic looking appearance. Too few iterations may result in rather unnatural looking wide cracks in the face, while too many on the other hand may become too smooth, ironing out the wrinkles.

The constraints need to be checked every iteration and as long as they are not fulfilled, they must develop forces with the aim to bring about a state in which they are fulfilled.

No matter how many iterations are set, the muscle constraints are calculated only once in their own iteration. This is imperative, since they are not weighted, their full forces are applied to the vertices, which then would never allow the other constraints to achieve meaningful results.

It is imperative to let every constraint work with the unaltered positions of its affected vertices in the current iteration. Only after all constraints had been calculated and added to the appropriate net force of a vertex, the net force of that iteration shall be applied to the vertex. The resulting new position is then valid for the entire next iteration. As noted in **3.1.1.**, it is important to give the forces generated by linear and position constraints a weight, with dividing the netforce by the number of added forces being sufficient for this purpose. Otherwise the skin simulation leads to very disfiguring results, ranging from a neverending “wobbly face” with constraints never coming to rest, to outright completely destroying the face.

As stated above, for each cycle, these operations must be performed over a number of iterations. The number of iterations is not fixed and while [3, p. 73][12, p. 7] found three iterations per frame to be optimal in terms of results and performance, the author found eight iterations to actually provide acceptable results, with only three iterations not sufficiently ironing out some artifacts related to the jaw and lid muscles, while still providing a tolerable performance. The library, however, was written in a way to allow a departure from this way and to add an own number of iterations, though eight is indeed the default value. It should be noted that due to the lower quality of face models, the need for more iterations for a smoother appearance may have very well be caused by the models and not the approach itself.

After each cycle, the vertex normals need to be recalculated to allow the mesh to actually reflect the created wrinkles. Without this step, the wrinkles become not nearly as pronounced as they should be [3, p. 43].

It is important to start calculating at every cycle from the initial vertex positions and to forget the current vertex positions, as otherwise the results are barely predictable, and muscle constraints may pull way beyond their designated target. This approach is also faster and does not give the facial movements a lagging appearance.

As noted in **3.1.3.1.**, the value for the skin elasticity needs to be between 0 and 1. During the implementation of the showcasing application, a value of 0.75 was chosen, since it allowed the skin to deform notably, but did not stretch to unreal proportions. In general, the lower the value, the more pronounced the deformations will be.

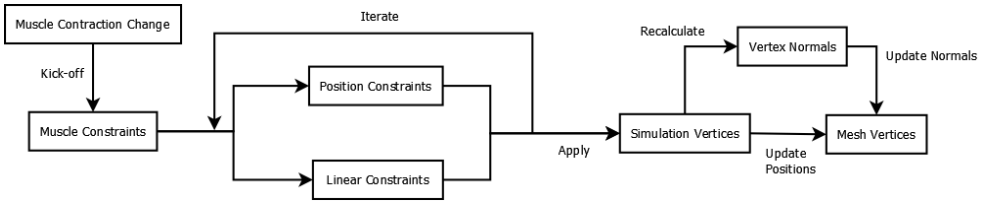

Figure 18 The order of constraint calculation and force application.

4.5 Implementation of the Muscle Simulation

The skin simulation is responsible for delivering the target positions V_{tar} for the muscle constraints and are active whenever a muscle contracts.

4.5.1 General Approach

The centrepiece of the muscle simulation are the muscles and the constraints they update. Given that the skin simulation is at rest as long as no muscle is contracted, the muscles are what gets things going.

Each muscle has its predefined set of muscle constraints which does not change during runtime. It was decided, that vertices wandering into an area of influence of a muscle as a result of contraction by another (or indeed, also the very same) muscle should not create new muscle constraints. The results would not be satisfactory, pulling for example eyebrows much further up, because after the contraction of the lower brow raisers they would be in the area of influence of the upper brow raisers.

Like with the constraints, the contractions of all simultaneous muscle contractions are performed in a way to not interfere with each other. This was however not a too big issue, since the muscles always use the initial positions of the vertices for their calculations and never the current positions.

The reason as to why initial positions and not current positions were used, is to simply have a deterministic outcome of the mathematical operations. Otherwise the muscles would allow for being boosted by each other, with one muscle pulling a vertex out and kickstarting the next one to pull it out way further, since the current position was already much further away to begin with.

Once the target positions are set, the muscle constraints are not fulfilled anymore, thus beginning to develop forces to deform the mesh, to which the other constraints of the skin simulation need to react.

4.5.2 The Muscles

To initialize a muscle, a certain set of information needs to be passed in form of a string. This set of information varies, depending on the type of the muscle. The various passed values need to be separated by semicolons, which need a leading and a trailing white space. The sets will be discussed further below in the appropriate sections with examples.

During the initialization the affected vertices are calculated. For each affected vertex, a muscle constraint is established, and every time a muscle contracts, it modifies these muscle constraints by determining new target positions for them.

As mentioned in **4.3.1.**, Unity switched the coordinate system from right-hand to left-hand, which meant that position data of the muscles needed to get their x-coordinates inverted as well.

4.5.2.1 Linear Muscle

Linear muscles need these informations for initialization:

```
VM ; AU1_left_inner_brow_raiser ; 1.2 4.2 0.2 ; 0.7 1.3 0.65 ; 0.6 ; 45
```

The first section is the type section, which for a linear muscle needs to be "VM" (for vertical muscle, an alternative name for this muscle type, needed to distinguish from the lid muscle).

The second section is the name. In this case the muscle in question is used for the left side application of AU1, which is the inner brow raiser.

The third section contains the coordinates of P_{OP}, while the fourth gives the coordinates of P_{EP}.

The fifth section contains the area not affected by the muscle falloff in percent – the given example thus means 60%. This value tells us, how much of the length, looking from P_{EP}, does not belong to the muscle falloff area. Differently put, this is the percentage of $R_f - R_s$.

The last section gives the muscle angle Ω in degrees.

With this information provided, the search for the affected vertices can begin. There are two criteria a vertex needs to fulfil in order to be affected by a linear muscle.

The first criterion is that $Dist_{OPV}$ is shorter or equal to $Dist_{OPEP}$. Any vertex that is further away of P_{OP}, can never be affected by that linear muscle.

The second criterion is that the angle μ is smaller or equal to the passed angle Ω. This was calculated using the following formula:

$$\mu = \cos^{-1}\left(\frac{Dir_{OPEP} \cdot Dir_{OPV}}{Dir_{OPEP\_Length} \times Dir_{OPV\_Length}}\right)$$

The directions must not be normalized. Another very important note is to keep in mind whether the angles are calculated in radians or degrees. C# math libraries usually work in radians, which is why the author of this thesis also used them in calculations. However, the provided data had the muscle angle given in degrees, which made a conversion necessary.

Vertices, which fulfil both criteria, are affected by the linear muscle, and a muscle constraint is created for them.

To calculate R_s, the inverse percentage of the appropriate passed parameter needs to be taken. The actual value of the distance R_s is then that percentage of R_f.

4.5.2.2 Sphincter Muscle

Sphincter muscles need their muscle information in this form:

```
SM ; AU6_left_lid_tightener ; -1.42 1.2 0.5 ; 1.4 1.6 1.5 ; 1 0.1 1.3 ; 0 1 0.5
```

The first section is the type section, with sphincter muscles being identified via the code 'SM'.

The second section contains the name. The given example is for the left lid tightener, Action Unit AU6.

The third section is also the same as for the linear muscle, as it contains the muscle origin.

The fourth section contains the outer radius R_o, which denotes the area of influence of the sphincter muscle.

The fifth section contains the inner radius R_i, from which on there is a muscle falloff.

The sixth section contains the scale factor S, which is needed to tweak the results somewhat to better fit their purposes.

Since the sphincter muscle is ellipsoidical in shape, the affected vertices must be calculated differently. For this purpose, the ellipsoid standard equation can be used:

$$\frac{{V_X}^2}{{R_X}^2} + \frac{{V_Y}^2}{{R_Y}^2} + \frac{{V_Z}^2}{{R_Z}^2} = 1$$

Any Vertex, for which the above equation results in a value of 1 or less, is within the boundaries of the muscle. As a hint, it is suggested to translate the ellipsoid to the origin point of $(0, 0, 0)$ for the calculation.

The rest of the calculation behaves exactly like with the linear muscle, sans the lacking angular displacement value. However, calculating the distances

R_S and R_f was much more challenging, given that merely the ellipsoids were given. It was now important in which direction the vertices were located, as R_S and R_f varied accordingly, due to the ellipsoid proportions.

Both, R_S and R_f, are calculated the same way. For starters, the direction from V_{init} to P_{OP} needs to be calculated, with normalisation.

The next step is to find a vertex which is definitely outside of the radius in question. It is recommended to simply seek for a vertex outside of R_o, as it then by definition is also outside R_i. This can be done by recursively multiplying the normalised direction vector recursively by some scalar (the author used the factor 10) and check during each recursion, whether the vertex is outside the boundaries of the sphincter muscle or not, until it is outside R_o. To make matters easier, it is recommended to start not from the P_{OP}, but instead from the coordinate *(0, 0, 0)*. Calculations are much easier this way, while the direction vector and the radii are unaffected by this. It is however important to stick to this new origin for the rest of the calculation of R_S/R_f.

Taking the helper vertex P_H and the inverse direction – which then must be the direction from P_H to P_{OP} or V_{init} – the function of the ellipsoid can be written like this:

$$P_R = P_H + t \times dir_{HOP}$$

P_R is the vertex on the radius in the same direction as the affected vertex is, looking from P_{OP}, while t is the factor with which the direction vector needs to be multiplied, so that after adding it to P_H, P_R is calculated.

This factor t is calculated with the quadratic equation.

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The three variables in the formula (a, b and c) can be calculated as seen below. For mathematically inclined readers, **Appendix A)** shows how to come to this final formula.

$$a = \left({R_Y}^2 \times {R_Z}^2 \times {dir_X}^2\right) + \left({R_X}^2 \times {R_Z}^2 \times {dir_Y}^2\right) + \left({R_X}^2 \times {R_Y}^2 \times {dir_Z}^2\right)$$

$$b = 2 \times \Big(\left({R_Y}^2 \times {R_Z}^2 \times V_{HX} \times dir_X\right) + \left({R_X}^2 \times {R_Z}^2 \times V_{HY} \times dir_Y\right) + \left({R_X}^2 \times {R_Y}^2 \times V_{HX} \times dir_X\right)\Big)$$

$$c = \left({V_{HX}}^2 \times {R_Y}^2 \times {R_Z}^2\right) + \left({V_{HY}}^2 \times {R_X}^2 \times {R_Z}^2\right) + \left({V_{HZ}}^2 \times {R_X}^2 \times {R_Y}^2\right) - \left({R_X}^2 \times {R_Y}^2 \times {R_Z}^2\right)$$

Finally, with the calculated t, P_R can be calculated. Once this is done, the distance between *(0, 0, 0)* and P_R has to be calculated. Whether R_s or R_f was calculated, depends on the radius used in the calculations. If R_i was used, R_s was calculated. If R_o was used, R_f was calculated.

One last word of notice: R_i is set to *(0, 0, 0)* when working with the lip pucker Action Unit (AU18). This needs to be kept in mind, as a check is needed to prevent divisions by zero while calculating R_s. R_s can safely be assumed to simply be *0*, since no muscle falloff exists in that case.

4.5.2.3 Lid Muscle

Lid muscles require their muscle information in the following form:

```
LM ; AU5_left_lid ; -1.7 1.48 0.3 ; 1 0.45 1.5 ; 1 1 1
```

In the type section, the lid muscle is identified with the letters 'LM'.

The second section tells the name of the Action Unit, in this case AU5, the lid, on the left side.

The third section is, as always, the coordinates of P_{OP}.

The fourth section gives R, the radii of the ellipsoid, with which the lid muscle is approximated.

The last section is actually entirely unused and may be a leftover from experimenting with the shape of the lid muscle, which seems to have been at one point based on the sphincter muscle.

Since both muscles are in the shapes of ellipsoids, the same formula for calculating the affected vertices can be used:

$$\frac{{V_X}^2}{{R_X}^2} + \frac{{V_Y}^2}{{R_Y}^2} + \frac{{V_Z}^2}{{R_Z}^2} = 1$$

Any Vertex, for which the above equation results in a value of *1* or less, is within the boundaries of the muscle. As with the sphincter muscle, it is suggested to translate the ellipsoid to the origin point of *(0, 0, 0)* for the calculation.

In **3.2.3.1.** and [3, p. 50] it was stated that the rotation of the lid muscle needs to be at a maximum of $90°$. However, since the lid rotates upwards, it was necessary to take $-90°$ for implementational purposes.

For the m_{rot}, the rotation matrix for the x-axis could be used, since the model was arranged in a way that the face is perpendicular to the x-axis. With Θ being $-90°$, the rotation matrix looks like this:

$$m_{rot} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(-90° \times K) & -\sin(-90° \times K) \\ 0 & \sin(-90° \times K) & \cos(-90° \times K) \end{pmatrix}$$

Once again it must be considered, which measuring units are being used for the angle. The values here are given in degrees. However, the implementation in C# demanded radians, due to the C# math libraries. K is the contraction strength, which regulates how far the eye lids really are to be opened.

4.5.2.4 Jaw Muscle

Jaw muscles need their information set in the following form:

```
JM ; AU26_jaw_drop ; 0 -1.9 -3 ; 5 4 5
```

The type section identifies this row as belonging to a jaw muscle with the letters ‘JM’.

The second section is again the name section, containing the only Action Unit dealing with the jaw, namely AU26.

The third section too follows the well-known pattern and gives the coordinates of P_{OP}.

The fourth coordinate contains the dimensions of the cube, with which the muscle area of influence is approximated.

One very important information about the provided data for the jaw muscle, is that P_{OP} is not in the center of the cube. While its x-coordinate is located centrally, it is situated on the upper plane of the cube, meaning that the y-value is subtracted as a whole from its position, when the lower end of the cube needs to be determined. Similarly, it is located on the back-side on

the z-axis, meaning that the z-value needs to be added to determine the front plane of the cube.

As was mentioned in **3.2.4.1.**, the optimal maximum of degrees for opening the mouth was determined to be somewhere around $35°$. Given that the mouth rotates downwards, Θ needs to be taken as a positive value. Once again, during implementation they needed to be converted from degrees to radians, given that the C# math libraries work with radians. Furthermore, since the jaw also rotates on the x-axis, the x-rotation matrix could be used, which looks like this for the jaw muscle:

$$m_{rot} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(35° \times K) & -\sin(35° \times K) \\ 0 & \sin(35° \times K) & \cos(35° \times K) \end{pmatrix}$$

Again, as mentioned, the necessary strength multiplier N was during trials to give satisfactory results with this calculation:

$$N = \frac{K}{15}$$

4.5.3 Usage of FACS

As described in **3.**, the muscle simulation was based on the Facial Action Coding System, which knows 46 different Action Units. However, [3] did not use all Action Units. Instead, only thirteen were actually used, with nine having separate versions for the left and right half of the face.

Since for this thesis the same facial data was provided as for [3], the proof of concept implementation used the same Action Units. However, in the scope of this thesis it was made possible to add all 46 Action Units in the accompanying application, provided that they can be simulated with one of the four muscle types presented in this thesis. To add more muscles, simply further lines of muscle data need to be added in the same style as presented in **4.5.2.**

This chapter will present the Action Units with provided data and via which muscle type they were implemented. For a more in-depth analysis and explanation of the decisions made, the reader is referred to [3, pp. 51-63].

4.5.3.1 Implemented via Linear Muscles

The Action Units AU1 (inner brow raiser), AU2 (outer brow raiser), AU4 (lower brow), AU9 (nose wrinkler), AU10 (upper lip raiser), AU14 (dimpler), AU15 (lip corner depressor), AU17 (chin raiser) and AU20 (lip stretcher) were implemented as linear muscles. With nine Action Units, the linear muscle is by far the most prominent in the face simulation, and is used for muscles all over the face, in particular at the sides of the face and the upper portion.

Except for two Action Units, all mentioned Action Units have two muscles attached to them. The exceptions are AU17, which is implemented with only one muscle, and AU10, which is implemented with four muscles.

Of the seven Action Units with two muscles, three work simultaneously on both sides of the face, namely AU1, AU4 and AU9. AU10 likewise contracts all four muscles simultaneously.

The remaining four Action Units (AU2, AU14, AU15 and AU20) may see each side contract independently. Thus, separate sliders were introduced for each side.

4.5.3.2 Implemented via Sphincter Muscles

The Action Units AU6 (lid tightener) and AU18 (lip pucker) were implemented as sphincter muscles, with AU6 having variants for left and right. As can be seen, the sphincter muscle is already somewhat more specialised than the linear muscle, with only two Action Units making use of it.

4.5.3.3 Implemented via Lid Muscles

The lid muscle was created with a way more specific field of applications in mind than either the linear or the sphincter muscles. Thus, it affects merely one Action Unit, AU5 (lid).

Since both lids can be contracted separately, once again two sliders were introduced, one for each side.

4.5.3.4 Implemented via Jaw Muscle

The jaw muscle, like the lid muscle, was created with a very narrow field of applications. It is used only for Action Unit AU26 (jaw drop), having only one muscle.

4.6 Further implementation details

A few things were implemented in the showcase application, which, while not within scope of the muscle or skin simulation, deserve to be highlighted.

4.6.1 Rendering mode

The rendering mode of the face can be changed from the normal face material, to a wireframe material [35]. The reason for this is to allow the user to see how muscle contractions affect the actual vertices in the face.

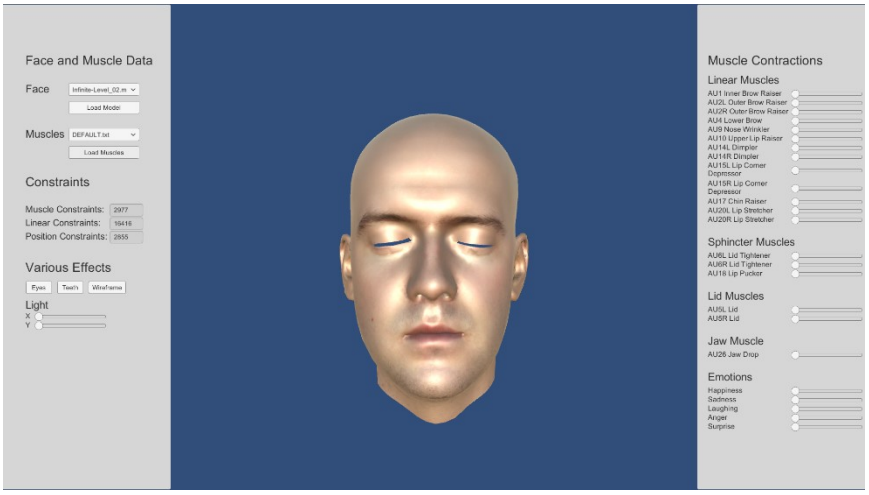

Figure 19 The Showcase Unity Application for this thesis. Standard rendering of the face.

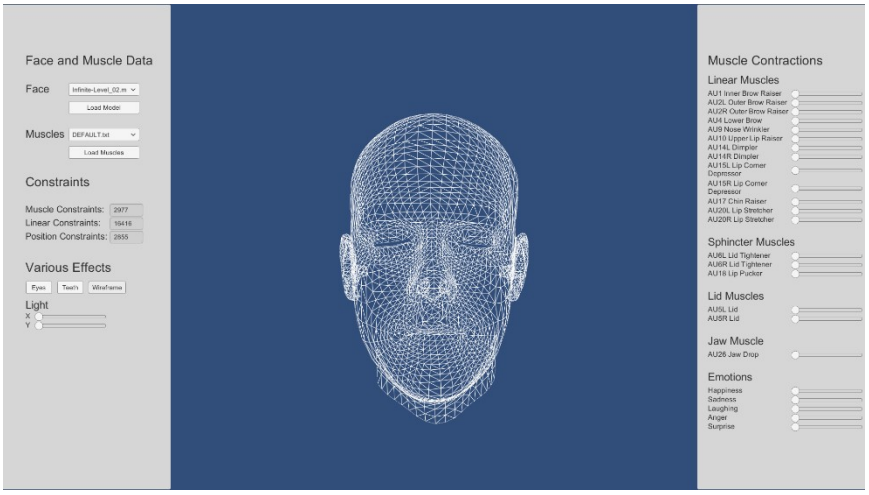

Figure 20 Wireframe rendering.

4.6.2 Emotions

Following the example set in [3, pp. 65-66], the Showcase Unity Application for this thesis is also capable of generating a few predefined emotions. This is, as in [3], done by combining specific Action Units, which then give the desired facial expression.

Multiple emotions can be combined at will, though the stronger the generated forces are, the more extreme the results will be. If the emotions affect the same Action Units, the last set emotion cancels out the results for Actions Units affected by the previously active emotion.

The following emotions were predefined:

- Happiness: AU2 + AU5 + AU6 + AU14 + AU20
- Sadness: AU1 + AU4 + AU5 + AU9 + AU15 + AU17 + AU20
- Laughing: AU1 + AU2 + AU6 + AU9 + AU14 + AU20 + AU26
- Anger: AU2 + AU4 + AU5 + AU6 + AU9 + AU15 + AU20 + AU26
- Surprise: AU1 + AU2 + AU5 + AU9 + AU14 + AU17 + AU26

4.6.3 Extras

There were a few extras implemented in the Showcase Unity Application, which are not part of the skin or muscle simulation and are thus not within scope of this thesis, but are still needed to make the face much more belieavable.

4.6.3.1 Eyes

For the eye models, the ones provided by [33] were found satisfactory. They were placed roughly at the same place as the lid muscles have their origin points, although very slightly further away from the centre, a bit higher up and more to the back.

While not part of the skin nor muscle simulation, the eyes are needed to give the face a more human look. They can however be toggled off to show clearer how the skin deforms.

4.6.3.2 Teeth

The teeth model was based on [31], but a few changes needed to be done. The model did not allow for the movement of the lower jaw, and there was an issue with the materials, which needed to be reintroduced as textures.

The teeth are, like the eyes, needed to give a more human look to the face, as otherwise the mouth would, upon opening, be a big hole with nothing behind it, which does not look natural at all. The lower jaw also follows the jaw muscle's contraction. Like the eyes, they can be toggled off to show skin deformations clearer.

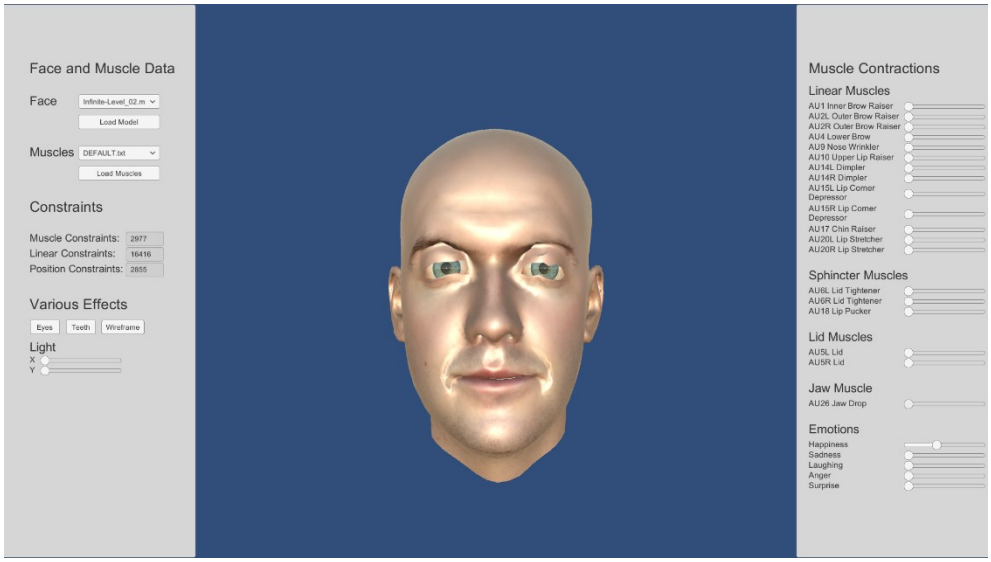

Figure 21 Screenshot of the *Showcase Unity Application*.

4.7 Showcase Unity Application

One of the big tasks of this thesis was to port the skin and muscle simulation algorithms presented in [3] and [12] from OGRE to Unity. This proof of concept was made with a showcase application, which allows a user to load face meshes and muscle data and to deform the face by manipulating the loaded muscles.

This showcase application will be described in this chapter in detail. On the left side of the application are the control elements for the faces and muscle data, as well as for effects not related directly to the skin or muscle simulation. On the right side are all control elements for the muscle contractions, which are categorized by their muscle types and sorted within each category by their Action Unit numbers. The control elements for the emotions are located on the right side as well, since they too are responsible for muscle contractions.

Figure 22 The Showcase Unity Application right after starting it, all empty.

4.7.1 Face and Muscle Data

The “Face and Muscle Data” section on the top left is the most important one for getting started. With the control elements of this section, the user can choose from all available faces and muscle data and by doing so, the user enables the other control elements.

Figure 23 The "Face and Muscle Data" section, used to pick face meshes and muscle data sets.

4.7.1.1 Face

The Face drop-down list shows all currently available faces, from which the user can choose. The faces are read from the folder *Assets/Resources/FaceModels*, with the folder *Assets* being in the same folder as the Showcase Unity Application. It automatically reads all usable meshes from the folder upon start and lists them in this drop-down list.

For a mesh to be usable, it needs to be provided in the .obj format, which is a mesh format Unity can interprete. There are no naming conventions.

Upon selecting a mesh and pressing the “Load Model” button, the face mesh is loaded. This action also unlocks the “Wireframe” button.

The user may load another face model at any given time, which will replace the currently shown face mesh.

Figure 24 The Showcase Unity Application after loading the face mesh "Debbie". Note that in comparison to **figure 22** the “Wireframe” button is active.

4.7.1.2 Muscles

The Muscles drop-down list shows all currently available muscle data sets. They are read from the location *Assets/Resources/SavedMuscles*. Like described in **4.7.1.1.** with the face meshes, the muscle data sets are read automatically upon start.

For a muscle data file to be read by the Showcase Unity Application, it merely needs to be in .txt format. No naming conventions exist.

Loading a muscle data set enables the muscle simulation to get going, as all information needed to initialize the muscles is provided. All so far still locked control elements, which are the eyes and teeth buttons, as well as the sliders for the muscle contractions, are enabled. Furthermore, since selecting a muscle data set starts the initialization of the skin and muscle simulations, the constraints are created and their numbers are presented in the Constraints section (see **4.7.2.**).

Like with the face meshes, the user may choose to load another set of muscle data at any time. Upon doing so, all muscle contraction sliders will be reset to zero, and the face mesh is reloaded in order to give a clean start.

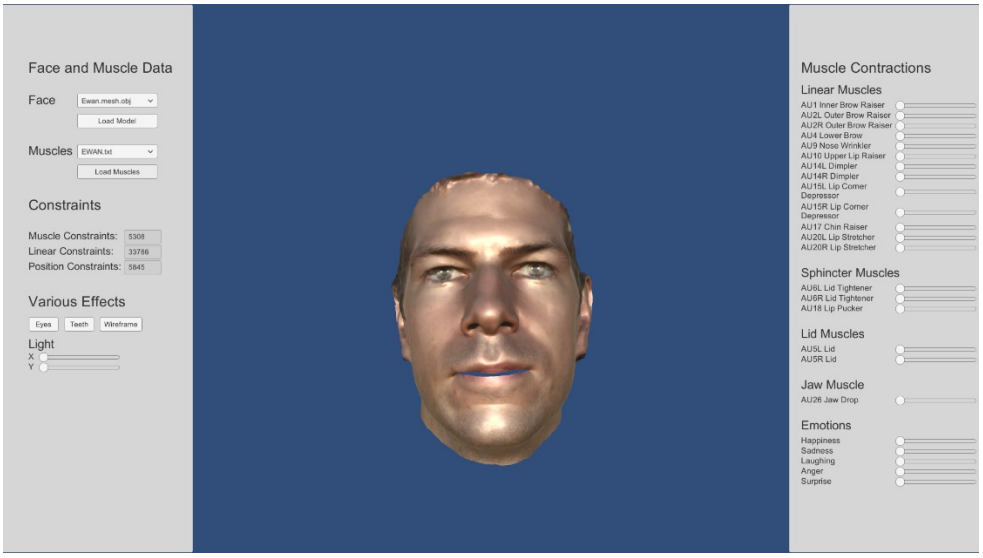

Figure 25 Face mesh "Ewan" and the appropriate muscle data set are loaded. Observe that the teeth and eyes buttons are enabled now, as are the sliders for the muscle contractions. The numbers of the constraints can be seen as well.

4.7.2 Constraints

The “Constraints” section on the centre left shows how many constraints of each type currently exist. This data is obtained after a muscle data set is chosen. Besides showing the number of each type of constraints, some other information can be deduced.

The field “Position Constraints” shows how many vertices the currently selected mesh has, since for each vertex there needs to be one position constraint (see **3.1.3**).

The field “Linear Constraints” shows in a simular manner how many edges the face mesh has.

The field “Muscle Constraints“ shows the sum of how many vertices all each muscle affects, with vertices being counted once for each muscle affecting them.

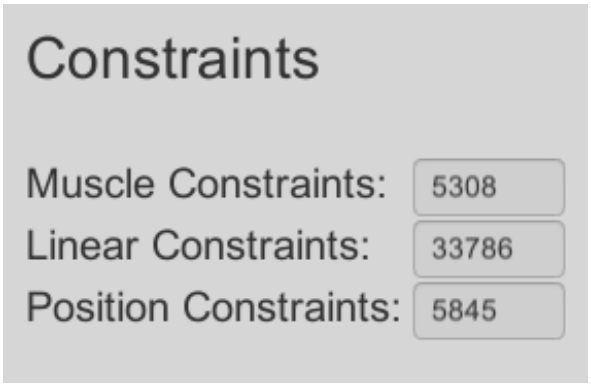

Figure 26 The numbers of the constrainst for mesh and muscle data set "Ewan".

4.7.3 Various Effects

The “Various Effects” section on the bottom left includes various toggleable effects and controls for the light source, none of which are directly influences by the skin and muscle simulations nor are influencing the simulations. They do, however, change the visual aspect of the Showcase Unity Application.

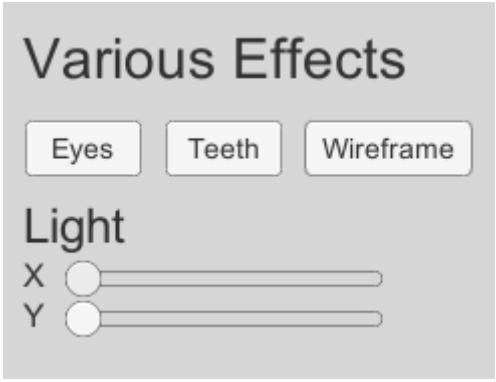

Figure 27 The "Various Effects" section with the "Eyes", "Teeth" and "Wireframe" buttons, as well as the light sliders.

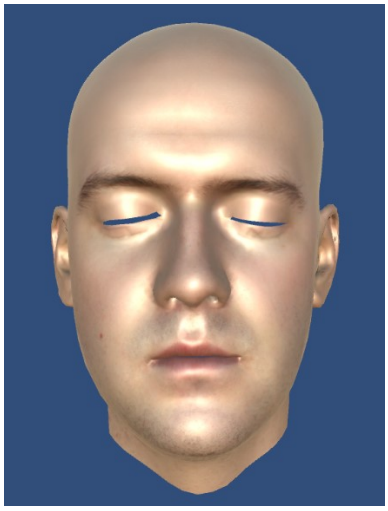

Figure 28 The "Infinite-Level_02" mesh with no extra effects.

4.7.3.1 Eyes

The “Eyes” button toggles the visibility of the eye meshes. This button is not enabled until a muscle data set is selected, since the positioning of the eyes requires the coordinates of P_{OP} the lid muscles.

Figure 29 The "Infinite-Level_02" mesh with eyes.

4.7.3.2 Teeth

The “Teeth” button toggles the visibility of the teeth mesh. Like the “Eyes” button, it is not enabled until a muscle data set is selected, since the coordinates of P_{OP} of the jaw muscle are needed for the positioning of the teeth mesh.

Figure 30 The "Infinite-Level_02" mesh with teeth. Mouth opened to show them.

4.7.3.3 Wireframe

The “Wireframe” button toggles the materials of the face from the ones normally assigned to the face mesh, to a wireframe material. This is useful for examining how the vertices of the face mesh change their positions upon muscle contraction.

The “Wireframe” button is enabled as soon as a face mesh is loaded.

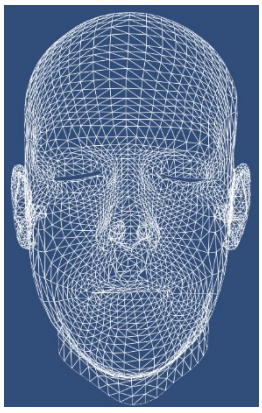

Figure 31 The "Infinite-Level_02" mesh with wireframe materials.

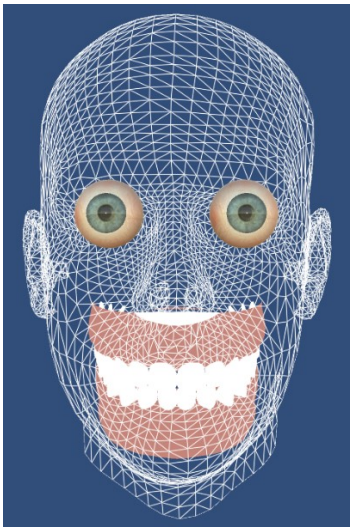

Figure 32 The "Infinite-Level_02" mesh with wireframe materials with the eyes and teeth activated.

4.7.3.4 Light

The light source in the Showcase Unity Application is not fixed. It can be rotated via two sliders on on the x-axis and the y-axis, with the sliders being marked appropriately.

Figure 33 The direction of the light changed by rotating around the x-axis (left) and the y-axis (right). Rotations can be performed simultaneously as well.

4.7.4 Muscle Contractions

The “Muscle Contractions” section on the right side contains the sliders for all muscle contractions. The categorization follows the muscle types presented in this thesis: linear, sphincter, lid and jaw muscles. There is one more category called “Emotions” (see **4.6.2.**). The muscles and emotions available in the Showcase Unity Application are predefined.

Each slider will now be presented with a screenshot to show its effect, while the description of the image will contain the name for the portrayed muscle, which needs to be included in the muscle data file. Some may need several muscles. In this case all names are provided. See **3.2.** for the structures of the muscle data sets for the various muscle types.

Of course, muscle contractions can be combined at will. For the muscle data sets it does not matter in which sequence the various muscles are delivered.

Figure 34 The "Infinite-Level_02" mesh in idle pose. The light source was rotated on the x-axis for better visibility of the effects.

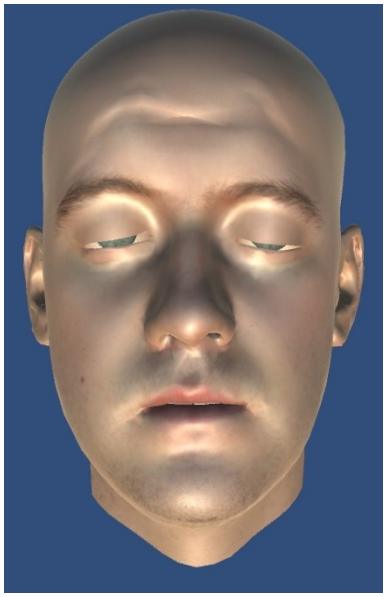

Figure 35 Action Unit 1, Inner Brow Raiser. Required muscle names: "AU1_left_inner_brow_raiser" and “AU1_right_inner_brow_raiser”.

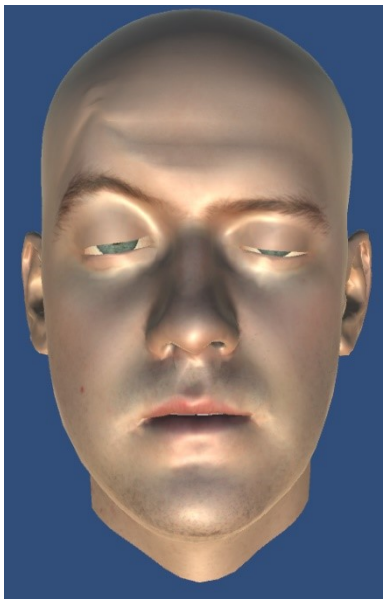

Figure 36 Action Unit 2, Left Outer Brow Raiser. Required muscle name: "AU2_left_outer_brow_raiser".

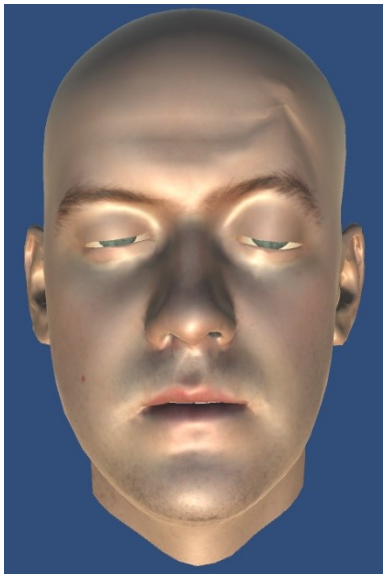

Figure 37 Action Unit 2, Right Outer Brow Raiser. Required muscle name: "AU2_right_outer_brow_raiser".

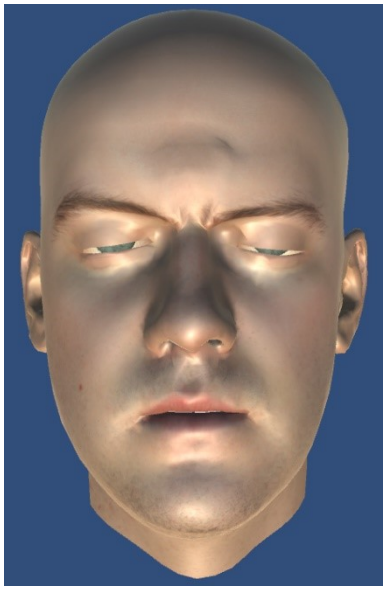

Figure 38 Action Unit 4, Lower Brow. Required muscle names: "AU4_lower_left_brow" and “AU4_lower_right_brow”.

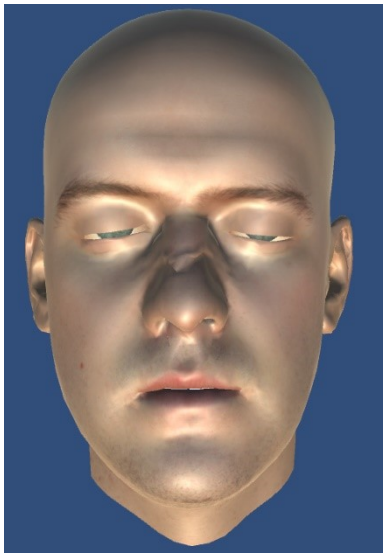

Figure 39 Action Unit 9, Nose Wrinkler. Required muscle names: "AU9_left_side_nose_wrinkler" and “AU9_right_side_nose_wrinkler”.

Figure 40 Action Unit 10, Lip Raiser. Required muscle names: "AU10_left_side_upper_lip_raiser", "AU10_right_side_upper_lip_raiser", "AU10a_left_side_upper_lip_raiser" and “AU10a_right_side_upper_lip_raiser”.

Figure 41 Action Unit 14, Left Dimpler. Required muscle name: "AU14_left_dimpler".

Figure 42 Action Unit 14, Right Dimpler. Required muscle name: "AU14_right_dimpler".

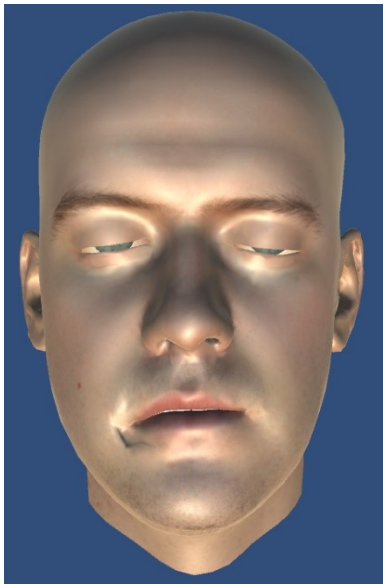

Figure 43 Action Unit 15, Left Lip Corner Depressor. Required muscle name: "AU15_left_lip_corner_depressor".

Figure 44 Action Unit 15, Right Lip Corner Depressor. Required muscle name: "AU15_right_lip_corner_depressor".

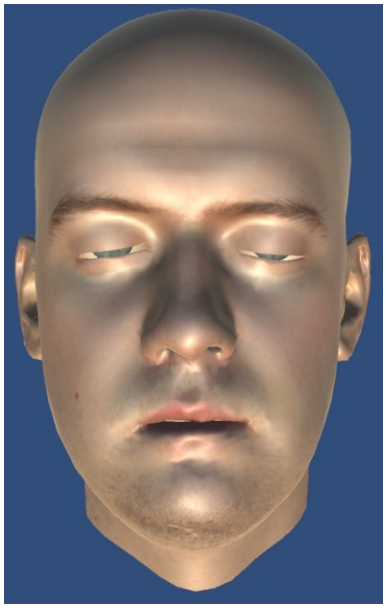

Figure 45 Action Unit 17, Chin Raiser. Required muscle name: "AU17_chin_raiser".

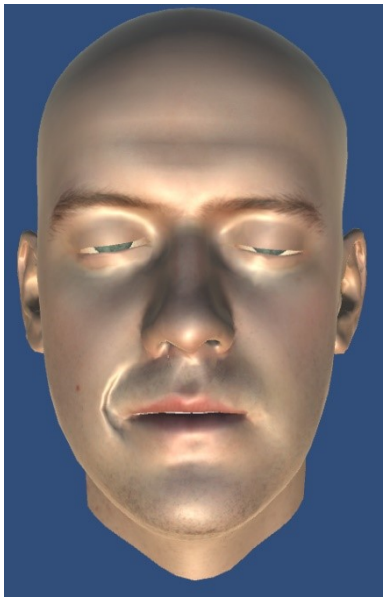

Figure 46 Action Unit 20, Left Lip Stretcher. Required muscle name: "AU20_left_lip_stretcher".

Figure 47 Action Unit 20, Right Lip Stretcher. Required muscle name: "AU20_right_lip_stretcher".

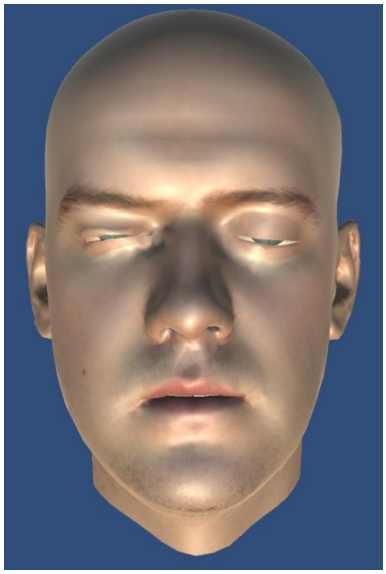

Figure 48 Action Unit 6, Left Lid Tightener. Required muscle name: "AU6_left_lid_tightener".

Figure 49 Action Unit 6, Right Lid Tightener. Required muscle name: "AU6_right_lid_tightener".

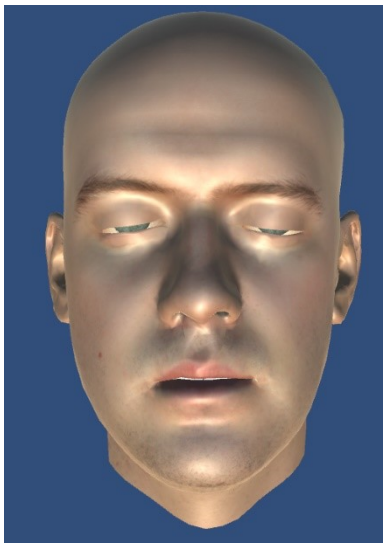

Figure 50 Action Unit 18, Lip Pucker. Required muscle name: "AU18_lip_pucker".

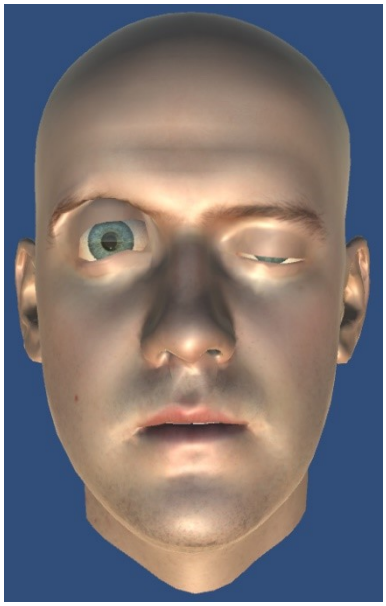

Figure 51 Action Unit 5, Left Lid. Required muscle name: "AU5_left_lid".

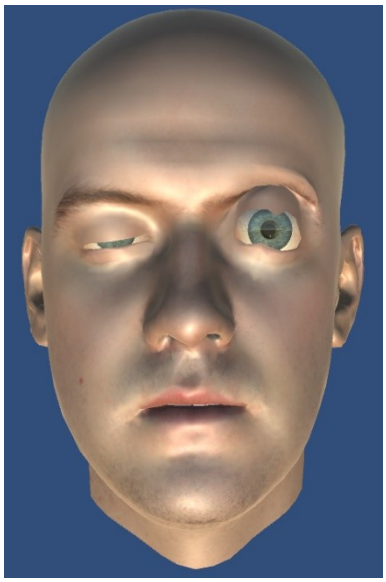

Figure 52 Action Unit 5, Right Lid. Required muscle name: "AU5_right_lid".

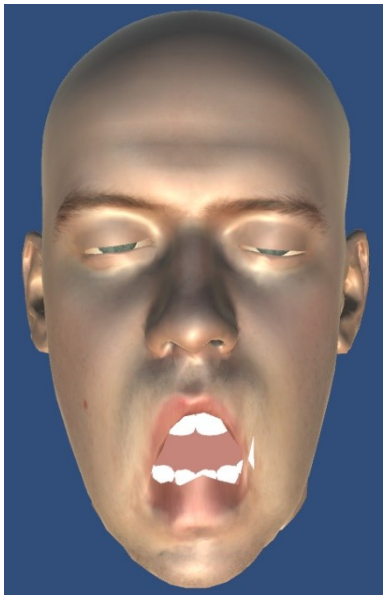

Figure 53 Action Unit 26, Jaw Drop. Required muscle name: "AU26_jaw_drop". Notice how the teeth move as well.

Figure 54 Happiness.

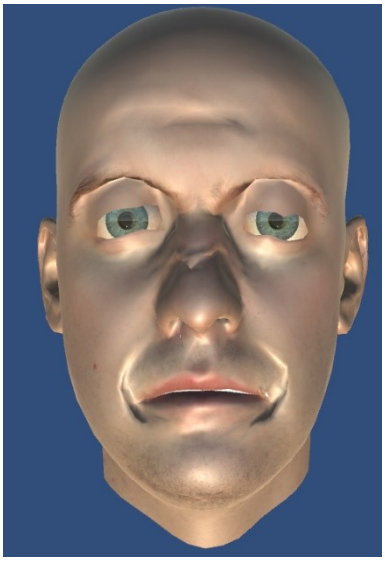

Figure 55 Sadness.

Figure 56 Laughing.

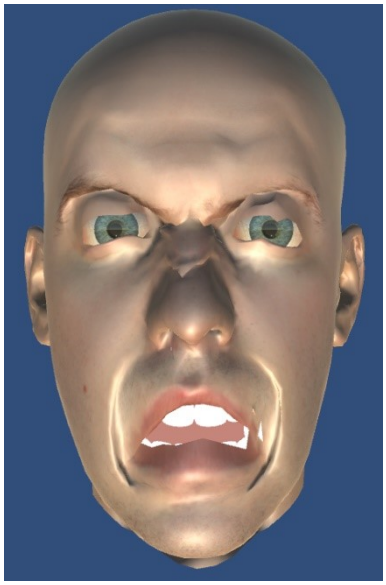

Figure 57 Anger.

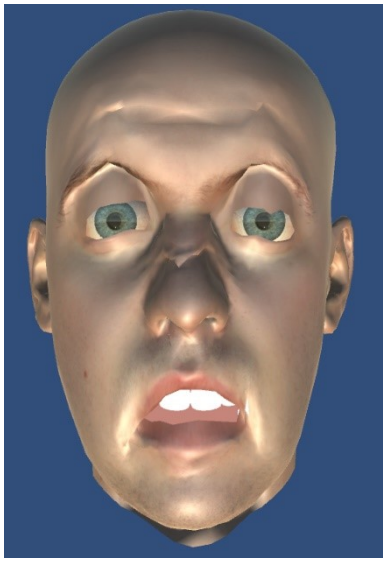

Figure 58 Surprise.

5 Results and Future Work

In the scope of this master's thesis the approach presented in [3][12] has been implemented anew with more modern technology. By using Unity, the field of Virtual Reality has been opened to the skin and muscle simulation presented in [3], [12] and this thesis.

By separating the GUI from the mathematical logic, a much higher degree of reusability has been achieved. The DLL responsible for the calculations can be reused in any project of any kind in a straightforward manner, there is absolutely no coupling to Unity or any other external dependencies.

The DLL can be configured by parameters to use either a left-hand or a right-hand coordinate system, which makes it usable for any kind of environment. The number of iterations is another parameter, which can be set arbitrarily.

There is no limited amount of muscles, which can be used at the same time with the DLL. The simulation can be made as fine or granular as the user desires or the possibilities allow. The only real prerequisite is that all muscles, which are supposed to take part in the simulation, must fit one of the four muscle types presented in this thesis.

The Showcase Unity Application was likewise designed with flexibility in mind. The offered faces are not hardcoded, they are read during runtime from a designated folder, as are muscle data files. This means that experimenting with a new face mesh can be done quick and easy – provided the user has appropriate muscle information.

One peculiarity was noted during experimentations with the approach presented in [3][12], and that was the demand to recalculate the constraints during every frame. Since every cycle is to begin with erasing the former forces, the produced results are bound to always be the same, as long as no muscle contraction changes. The Showcase Unity Application of this thesis went a different approach and recalculates the constraints only when there is a need to do so, which means whenever any muscle contraction is being changed by any of the sliders responsible for either a muscle or an emotion.

The logic for this was kept in the GUI section of the project, since it cannot be excluded that there may very well be valid use cases demanding a

continuouos calculation. Therefor the responsibility for handling this issue has not been given to the DLL, but rather the calling program.

The author also believes that by separating muscle constraints from position and linear constraints more strictly, unnecessary iterations are prevented, while the system behaves more as it would be expected.

5.1 Difficulties

Several difficulties were encountered during the implementation of the presented skin and muscle simulation. Interestingly, most of them had rather little to do with the actual algorithms, but with the meshes, which had proven to be the weakest link.

Converting a mesh in OGRE's native format to a more common format seems to have received so far little attention. This was quite surprising, given that Unity is more tolerant to different format types than OGRE, and tools like Blender can work with a plethora of formats. The only possibility the author found to convert a mesh made for OGRE to a format usable by Unity, had been [28].

Sadly, this was not a perfect conversion. The mentioned converter seems to downsample a mesh to a big degree. To illustrate this point: the Facetools toolbox of [3] could create 5055 muscle constraints for the face mesh "Infinite-Level_02", whereas the Showcase Unity Application of this thesis could create merely 2977. The reason for this discrepancy is that during the conversion either some vertices are lost, or several vertices are united.

This problem shows the great dependency of the presented skin and muscle simulations on the number of vertices. The more vertices there are in a mesh, the better the results of the simulation. Movements look smoother, while wrinkles look finer. The fewer vertices there are, the higher the probability that a muscle activation will give rather unbelieavable results, appearing as if a rigid chunk of flesh is moving.

The author also suspects that some vertices may have actually been moved slightly during the conversion. In most cases this should not even be discernible or disturb too much. However, each of the rotation muscles had difficulties with stray vertices. The lid muscles for example are prone to catch one single vertex from the lower lid, which should not move upwards. By being pulled upwards though, it creates a triangle which starts to cover the eye from the lower half of the eye, and the stronger the lid muscle contracts,

the more of the eye is actually hidden again! The jaw muscle has the inverted problem – sometimes vertices on the top section of the lower lip are not caught directly, which means that they are lagging very visible behind, in fact moving downwards only because of the linear constraints affecting them, as they are pulled closer to their edge partners which did get pulled down. However, since the dominant muscle constraint force is missing, the effect of lagging behind is still very visible.

Both issues can be mitigated by letting more iterations be performed per cycle. This, however, is at some point detrimental to the actually desired deformations, as they are slowly eroded.

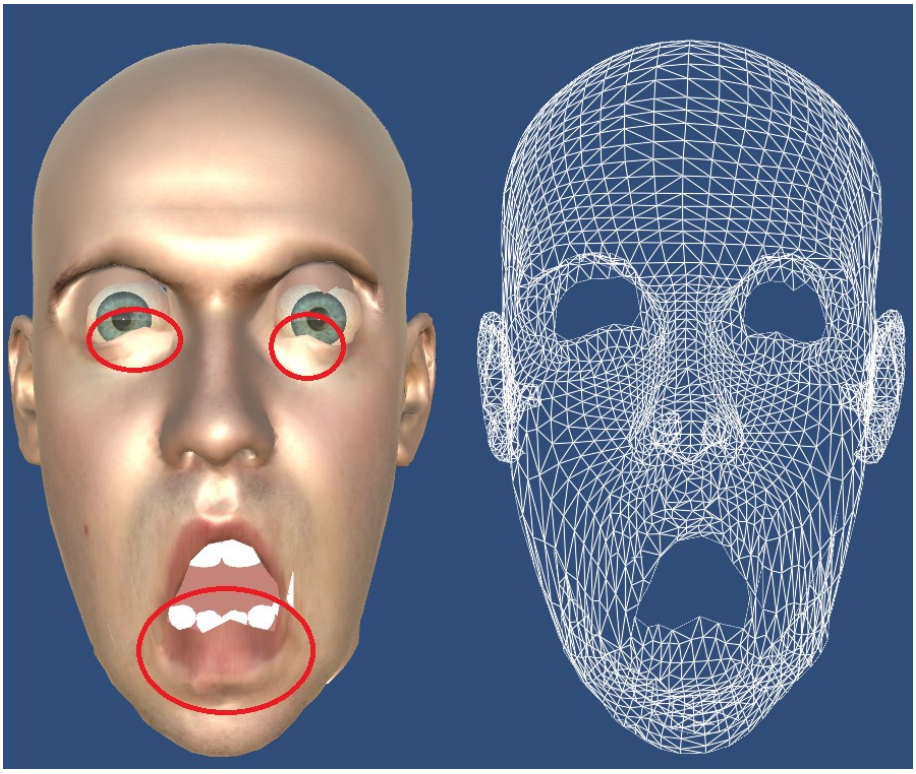

Figure 59 Demonstration of the lagging lower lip vertices and the falsely affected lower lid vertices.

All in all, due to the downsampling of the meshes, the results are somewhat worse. It can be argued though, that this comparison is problematic, since the meshes ultimately are only superficially similar.

Iterations are another difficult aspect of the skin simulation. The idea of corrective iterations proved to be useful and the right way to go. What is not so clear-cut, is the number of iterations.

Sadly, there are several factors which need to be considered for this question.

One important aspect is of course speed. The fewer iterations there are, the faster the calculation will be. This however produces worse results.

If the mesh in question has many vertices, then the calculation of the constraints for it can be truly time-consuming. Here it might be better to actually perform less iterations.

Furthermore, the more of these corrective iterations, which after all try to mitigate the effects of the muscle constraints, the less pronounced will the effect of the muscles be.

Perhaps it can also be said that with more vertices, there will be also more constraints, which may influence each other in such a way, that for the wrinkle generation a too high number of iterations is not even necessary.

Conversely, having a mesh with fewer vertices, the results of fewer iterations will allow more artifacts (like in **figure 59**) to be preserved. Such artifacts can be mitigated by more iterations. However, the aforementioned danger of cancelling out the muscle constraints, still persists.

5.2 Future Work

The porting of the skin and muscle algorithm of [3] and [12] to Unity and making them usable for VR applications, naturally opens many doors which can be taken. However, there are a few issues which may be more pressing than others.

One issue is speed. It is most likely that with future progress, the face meshes will be more detailed and contain more and more vertices. The logical consequence is, that the calculations for such faces will be more computationally expensive.

One approach to combat this problem is to introduce threads. With multi-core CPUs being a defacto standard today, and the number of cores still growing, a skilful load balancing might greatly improve the performance of the algorithm, making it usable even for highly detailed faces.

Another approach would be to pre-calculate as many values as possible. Given that the muscle calculations make use of fairly many constants, with effectively only the contraction strength changing, it would be beneficial to reduce the calculations for a muscle from several operations to merely one, where the contration strength is factored in. This would work particularly well for linear muscles, which also happen to be the most common muscles in the human face.

[3, p. 22] brought up an interesting thought, which sooner or later should be tackled. Human skin does not stay the same over the years. With age permanent wrinkles appear, skin starts to hang more loosely, gravity then has to be considered, even if it has a rather low impact. Perhaps medical conditions or other factors affecting the state of the human skin can be interesting to be factored in, since such characteristics and imperfections make a model appear much more real.

A topic which has not much to do with skin and muscle simulation, but is still nevertheless important for the setting of the VR conference room, is a good simulation of eyes and teeth. In particular the eyes are an essential and very expressive component of the human face. Simulating eye movement, eye focus and blinking are all topics which give an animated face a certain warmth, with which the Uncanny Valley [16] can be bridged.

To maximize the results of the eye and teeth simulation it will most likely be necessary to store information for individual faces. This means more unforgiving work, like with collecting and setting the muscle data.

Perhaps most important, as this thesis has shown, is the supply with good face meshes. A mesh with an insufficient or suboptimal number of vertices seriously inhibits the skin and muscle simulation in achieving the goal of a realistic portrayal of human skin.

The vertex number alone is not enough either. Another requirement are well placed edges. Areas, in which there are rather few edges, despite a high number of vertices, are prone to breaking, leaving visible gaps in the face. See **figure 60** for an example.

Figure 60 Face mesh "Debbie" with a crack across the forehead, the nose and the upper lip.

6 Conclusio

The goal of this master's thesis was to modernize the skin and muscle simulation algorithm presented in [3] and [12]. These algorithms were to be ported to Unity in order to be usable for a future project called "Conference Room in Virtual Reality". If possible, the algorithms were to be improved.

The porting has succeeded, the algorithms are now rewritten in C# and used in a Showcase Unity Application. Furthermore, by decoupling components from each other, a much greater reusability has been achieved. By tweaking the algorithms, in particular the exection order of some steps, the danger of pointless iterations has been greatly lowered. By offering an "on-demand" approach instead of a per-frame repeat of calculations, the performance could be increased as well. However, for cases where per-frame might be desirable, it was decided to leave the decision by the user of the mathematical DLL, who would call the calculation methods as is seen fit.

Difficulties and pitfalls of the conversion from OGRE to Unity have been mapped, with workarounds explained where they were found.

One big issue remains, which is not directly part of the skin or muscle simulation, but greatly affects their effectivity: mesh conversion. The currently existing methods to convert meshes from OGRE's native type to a type readable by Unity seem to work lossy, meaning that the models optically look alright, the number of vertices shrinks, which is highly detrimental to algorithms which base their entire functionality on vertices and their connecting edges. This, however, is a problem which was out of scope of this master's thesis.

References

Literature

1. Alexander, Oleg et al. “Creating a Photoreal Digital Actor: The Digital Emily Project”. In: *SigGraph ’09.* Association for Computing Machinery, New York, NY, USA, 2009
2. Bergeron, Philippe and Lachapelle, Pierre. “Controlling facial expression and body movements in the computer generated short ”tony de peltrie””. In: *SigGraph ’85 Tutorial Notes.* Association for Computing Machinery, New York, NY, USA, 1985, pp. 61–79
3. Beutl, Leon. “A simulation for the creation of soft-looking, realistic facial expressions”. MA thesis. University of Vienna, 2011
4. Bickel, Bernd et al. *Multi-Scale Capture of Facial Geometry and Motion.* 2007
5. Bickel, Bernd et al. “Pose-Space Animation and Transfer of Facial Details”. In: *Proceedings of the 2008 Eurographics/ACM SIGGRAPH Symposium on Computer Animation.* Eurographics Association, Airela-Ville, Switzerland, 2008, pp. 57–66
6. Bui, The Duy and Heylen, Dirk and Nijholt, Anton. “Improvements on a simple muscle-based 3D face for realistic facial expressions”. In: *Proceedings 11th IEEE International Workshop on Program Comprehension.* IEEE, 2003
7. Coquillart, Sabine. “Extended free-form deformation: A sculpturing tool for 3d geometric modeling”. In: *SigGraph ’90.* Association for Computing Machinery, New York, NY, USA, 1990, pp. 187–196
8. Deng, Zhigang and Noh, Junyong. *Computer Facial Animation: A Survey*. 2007
9. Dutreve, Ludovic and Meyer, Alexandre and Bouakaz, Saďda. “Easy Acquisition and Real-Time Animation of Facial Wrinkles”. In: *Computer Animation and Virtual Worlds, Wiley* Volume 22, Issue 2-3 (2011), pp. 169–176
10. Ekman, Paul and Friesen, Wallace and Hager, Joseph. *Facial Action Coding System.* 2002

11. Escher, Marc and Pandžić, Igor S. Thalmann, Nadia M.. “Facial deformations for MPEG-4”. In: *Computer Animation 98. Proceedings.* IEEE, 1998
12. Hlavacs, Helmut and Beutl, Leon. *Simulation of soft-looking Facial Expressions.* 2012
13. Kähler, Kolja et al. “Head shop: generating animated head models with anatomical structure”. In: *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation.* Association for Computing Machinery, New York, NY, USA, 2002, pp. 55–63
14. Larboulette, Caroline and Cani, Marie-Paule. “Real-Time Dynamic Wrinkles”. In: *Computer Graphics International (CGI’04).* IEEE Computer Society Press, 2004, pp. 522–525
15. Lee, Skeel Keng-Siang. *Introduction to Soft Body Physics.* 2008
16. Mori, Masahiro. “The Uncanny Valley”. In: *IEEE Robotics & Automation Magazine* (June 2012), pp. 98–100
17. Parke, Frederic I.. *Computer generated animation of faces*. 1972
18. Parke, Frederic I. and Waters, Keith. *Computer Facial Animation, Second Edition.* A K Peters, 2008
19. Pighin, Frederic et al. “Synthesizing Realistic Facial Expressions from Photographs”. In: *SigGraph ’98.* Association for Computing Machinery, New York, NY, USA, 1998, pp. 75–84
20. Sederberg, Thomas W. and Parry, Scott R.. “Free-form deformation of solid geometric models”. In: *SigGraph ’86.* Association for Computing Machinery, New York, NY, USA, 1986, pp. 151–160
21. Thalmann, Nadia M. and Thalmann, Daniel. *Interactive Computer Animation.* Prentice Hall, 1996
22. Waters, Keith. “A muscle model for animation three-dimensional facial expression”. In: *ACM SIGGRAPH Computer Graphics* Volume 21 Issue 4 (July 1987), pp. 17–24
23. Waters, Keith and Levergood, Thomas M.. DECface: *An Automatic Lip-Synchronization Algorithm for Synthetic Faces.* Tech. rep. Digital Equipment Corporation Cambridge Research Lab, 1993
24. Waters, Keith and Terzopoulos, Demetri. *The computer synthesis of expressive faces.* 1992

Online sources

25. *Facebook Spaces Homepage*. Aug. 2018. URL: https://www.facebook.com/spaces/
26. *HTC Vive Homepage*. Aug. 2018. URL: https://www.vive.com/eu/
27. *Infinite Realities*. Aug. 2018. URL: http://ir-ltd.net/
28. *Mesh Converter*. Aug. 2018. URL: http://www.greentoken.de/onlineconv/
29. *Oculus Homepage*. Aug. 2018. URL: https://www.oculus.com/
30. *Ogre Homepage*. Aug. 2018. URL: https://www.ogre3d.org/
31. *Teeth Assets*. Aug. 2018. URL: https://www.cgtrader.com/free-3d-models/character/anatomy/human-teeth-6b7374ef-8c18-4f47-aee2-539ba0266185
32. *Unity Homepage*. Aug. 2018. URL: https://unity3d.com/
33. *Visus Eyes*. Aug. 2018. URL: https://assetstore.unity.com/packages/3d/characters/visus-realistic-eyes-and-motion-119035
34. *Wikipedia, MVC-Pattern.* Aug. 2018. URL: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller
35. *Wireframe*. Aug. 2018. URL: https://assetstore.unity.com/packages/vfx/shaders/directx-11/ucla-wireframe-shader-21897

Appendix

A) Finding a Point on an Ellipsoid

The ellipsoid standard equation is:

$$\frac{x^2}{a^2}+\frac{y^2}{b^2}+\frac{z^2}{c^2}=1$$

The variables *a*, *b* and *c* denote the radii of the ellipsoid, whereas *x*, *y* and *z* denote the coordinates of a point. If the equation above equals *1* for a certain point, then this point is located exactly on the surface of the ellipsoid.

The equation for a line in three-dimensional space is:

$$\vec{V}=\begin{pmatrix}A\\B\\C\end{pmatrix}+t\begin{pmatrix}j\\k\\l\end{pmatrix}$$

$\vec{V}$ denotes the line. *A*, *B* and *C* are the coordinates of the origin point, whereas *j*, *k* and *l* are the components of the direction vector of the line from the origin point. The scalar t is the variable, via which any point on the line can be calculated.

To find a point which is situated right on an ellipsoid and to which a specific direction vector points from the origin point of the ellipsoid, the two equations above need to be combined.

It is assumed that the origin point of the line is also the origin point of the ellipsoid. Thus, the line equation needs to be plugged in the ellipsoid standard equation:

$$\frac{(A+tj)^2}{a^2}+\frac{(B+tk)^2}{b^2}+\frac{(C+tl)^2}{c^2}=1$$

The goal now is to rearrange the equation above to solving for the *t*, which gives the point on the ellipsoid, when plugged in the line equation:

$$\frac{(A+tj)^2}{a^2}+\frac{(B+tk)^2}{b^2}+\frac{(C+tl)^2}{c^2}-1=0$$

$$(A+tj)^2b^2c^2+(B+tk)^2a^2c^2+(C+tl)^2a^2b^2-a^2b^2c^2=0$$

$$\begin{gathered}(A^2+2Atj+t^2j^2)b^2c^2+(B^2+2Btk+t^2k^2)a^2c^2\\+(C^2+2Ctl+t^2l^2)a^2b^2-a^2b^2c^2=0\end{gathered}$$

To solve now for *t*, the quadratic equation needs to be used:

$$ax^2+bx+c=0$$

Variable *x* in the quadratic equation is *t*. The variables *a*, *b* and *c* now need to be determined from the last step of rearranging. They have nothing to do with the radii, it is just customary for the quadratic equation to name the variables that way.

The next step is to rearrange the equation in a way to separate by values including t^2, those including *t* and those which have no *t*.

$$\begin{gathered}A^2b^2c^2+2Atjb^2c^2+t^2j^2b^2c^2+B^2a^2c^2+2Btka^2c^2+t^2k^2a^2c^2\\+C^2a^2b^2+2Ctla^2b^2+t^2l^2a^2b^2-a^2b^2c^2=0\end{gathered}$$

$$\begin{gathered}t^2(j^2b^2c^2+k^2a^2c^2+l^2a^2b^2)+2t(Ajb^2c^2+Bka^2c^2+Cla^2b^2)\\+A^2b^2c^2+B^2a^2c^2+C^2a^2b^2-a^2b^2c^2=0\end{gathered}$$

Now *a*, *b* and *c* of the quadratic equation can be formed:

$$a=j^2b^2c^2+k^2a^2c^2+l^2a^2b^2$$

$$b=2(Ajb^2c^2+Bka^2c^2+Cla^2b^2)$$

$$c=A^2b^2c^2+B^2a^2c^2+C^2a^2b^2-a^2b^2c^2$$

Finally, they need to be plugged in the quadratic formula:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

B) Glossary

Variable name	Meaning
adv	Angular displacement value
Dir	Direction of a force
$Dist$	Distance between two vertices/points
f	Constraint force
F	Net force
K	Contraction strength/muscle spring constant
L	Jaw muscle cube side length
M	Magnitude of a constraint force
N	Necessary strength multiplier for a jaw muscle
m_{rot}	Rotation matrix
P_{EP}	Muscle endpoint of a linear muscle
P_H	Helper point for sphincter muscle R_s / R_f calculation
P_{joint}	Joint point of a rotation muscle
P_{OP}	Muscle origin point
P_R	Point on radius of sphincter muscle
R	Lid muscle radius
R_f	Distance P_{OP} to furthest end of muscle area of influence
R_i	Inner radius of sphincter muscle
R_o	Outer radius of sphincter muscle
R_s	Distance, in which muscle falloff exists
V_{curr}	Current vertex position
V_{init}	Initial vertex position
V_{tar}	Vertex target position
α	Skin elasticity
Θ	Rotation angle for rotation muscle
Ω	Angle of linear muscle

Summary

English

Realistic simulations of human faces are in high demand, especially since the boom of Virtual Reality technology. However, most approaches are either computationally very expensive, or require excessive preparation work. The master's thesis of Leon Beutl addressed this problem in 2011. Writing his application in the OGRE framework, it is not usable for current VR technologies though.

Therefor this thesis aims at reimplementing the algorithms presented in 2011, and their improvements in 2012 by Helmut Hlavacs and Leon Beutl, in the state-of-the-art game engine Unity, which has excellent VR support.

The approach laid out by Hlavacs and Beutl is a constraint based skin simulation, which develops forces as soon as the constraints are not fulfilled anymore, in order to return them to their rest states. A muscle simulation, based on Waters' muscle model from 1987 but enriched by one more muscle type with two subtypes, is used to cover all facial muscles needed, allowing to portray emotions on a set of face meshes.

This thesis decoupled the presentation layer from business layer by having a DLL for mathematical calculations, which is merely included by a showcase Unity project. Both, the DLL and the Unity project were made flexible to allow quick and easy experimenting and high reusability, which was particularly important for the mathematical aspect.

This thesis, while still allowing for it, abandoned the approach of calculating the constraints every frame and adopted an "on-demand" approach instead, keeping calculations at a minimum.

Problems arose mostly with peripherical topics, in particular the obtainment of meshes usable for Unity from the OGRE meshes, which is a problem which so far cannot be solved satisfactorily. However, this does not affect the actual algorithms, which work with what they receive.

Deutsch

Es besteht eine hohe Nachfrage nach realistischen Simulationen menschlicher Gesichter, insbesondere nach dem Boom der Technologie der Virtuellen Realität. Jedoch sind die meisten Ansätze entweder sehr rechenintensiv, oder verlangen exzessive Vorbereitungsarbeiten. Die Masterarbeit von Leon Beutl nahm sich 2011 dieses Problems an. Da er seine Applikation jedoch im OGRE-Framework geschrieben hatte, ist sie für gegenwärtige VR-Technologien nicht verwendbar.

Deswegen hat diese Arbeit als Ziel, die Algorithmen, die 2011 vorgestellt wuden, als auch deren Verbesserungen von Helmut Hlavacs und Leon Beutl aus dem Jahre 2012, auf den Stand der Technik zu bringen mit der Unity Game-Engine, welche exzellente VR-Unterstützung bietet.

Der von Hlavacs und Beutl vorgestellte Ansatz ist eine bedingungsbasierte Hautsimulation die Kräfte entwickelt, sobald eine der Bedingungen nicht mehr erfüllt wird, mit dem Ziel, sie wieder in einen Ruhezustand zu versetzen. Eine Muskelsimulation, die auf dem Muskelmodell von Waters aus dem Jahre 1987 basiert, jedoch um einen weiteren Muskeltypen mit zwei Subtypen erweitert wurde, wird für alle benötigten Gesichtsmuskel verwendet und erlaubt es, Emotionen an einer Palette von Gesichts-Meshes zu zeigen.

Diese Arbeit entkoppelte die Präsentationsschicht von der Businessschicht, indem eine DLL für mathematische Berechnungen enthält, die von einem Unity Vorzeigeprojekt lediglich eingebunden wird. Sowohl die DLL als auch das Unity-Projekt wurden flexibel genug gestaltet, um schnelles und einfaches Experimentieren zu ermöglichen, als auch eine hohe Wiederverwendbarkeit zu erlauben, was insbesondere für den mathematischen Part von großer Wichtigkeit ist.

Obwohl sie es noch immer zulässt, wurde mit dieser Arbeit der Ansatz des Berechnens der Bedingungen in jedem Frame aufgegeben. Stattdessen wurde ein „nach Bedarf“-Ansatz eingeführt, womit die Anzahl der Berechnungen auf ein Minimum gesenkt wird.

Probleme traten hauptsächlich mit Randthemen auf, insbesondere mit dem Gewinnen von für Unity nutzbaren Meshes aus OGRE Meshes, was ein Problem darstellt, welches bis jetzt nicht zufriedenstellen gelöst werden

konnte. Dies beeinträchtigt jedoch nicht die eigentlichen Algorithmen, die mit dem arbeiten, was sie bekommen.