universität
wien

MASTERARBEIT

Titel der Masterarbeit

A simulation for the creation of soft-looking, realistic facial expressions

Verfasser

BSc. Leon Beutl

angestrebter akademischer Grad

Diplom-Ingenieur (Dipl. -Ing.)

Wien, 2011

Studienkennzahl lt. Studienblatt:	A 066 935
Studienrichtung lt. Studienblatt:	Medieninformatik
Betreuer:	Ao. Univ.-Prof. Dipl.-Ing. Dr. Helmut Hlavacs

Abstract

In dieser Arbeit wird ein Weg zur Erstellung von realistischen, weich wirkenden, virtuellen Gesichtsausdrücken vorgestellt. Um dies zu ermöglichen wurde eine modifizierte, von Waters 1987 entwickelte, Simulation der Gesichtsmuskeln mit einer Simulation der menschlichen Haut kombiniert. Die Hautsimulation basiert auf einer Reihe simpler Bedingungen und dient zur Faltenbildung im Gesicht. Für die Gesamtsimulation wurde daraufhin ein Tool erstellt, welches das Austesten und die Manipulation der Muskeln ermöglicht.

0. Index

1. Introduction

The simulation of a virtual human face as realistic as possible and in real time, is a difficult challenge which can be broken down into three sub tasks. A polygon model with a realistic topology is the foundation, and can either be created by an artist, or acquired by a scanning system from a real person. On this model one, or often multiple detailed textures are applied for a realistic rendering. The textures have to include the slight color variations of the skin, and fine details, like pores and moles. The final task is to bring the human face to life by animating it. In my thesis I will focus on this last task, the animation of a human face, and present a solution for the generation of soft looking facial expressions.

Realistic facial expressions are needed in many virtual simulations, such as video games and avatar applications. However, the achieved results are often far away from convincing the user that he is actually interacting with a human being. The faces of virtual humans tend to be not soft enough, they seem to have more in common with a mask than with an actual face. Related to this problem is, that the faces are also not expressive enough, showing no real emotions. This destroys the illusion of interacting with an 'alive' human being rather fast. The name 'Uncanny Valley' [1] was assigned to a similar problem, when the viewer tends to feel repulsion or rejection towards the virtual character rather than emotional engagement.

In my opinion, to simulate a human face realistically, solutions for two main problems have to be developed. On one side a suitable simulation for the complex arrangement of facial muscles has to be found, on the other side a way has to be developed to simulate human skin and to give the face an overall soft look.

The task of simulating facial muscles can be solved in two ways. One is to simulate the positions, the form and the number of the facial muscles as well as their contraction behavior. The second way is to only focus on the muscle contraction, or rather the visible effects of the facial muscles on the surface. However, any good solution for a facial muscle simulation has also to be flexible enough, to be easily adjustable for different face models and has to be fast enough to allow the calculation in real time.

The main problem by the simulation of human skin is the creation of small wrinkles, that

facial expressions generate. These wrinkles do not only let us perceive a face as soft, but also help to make the emotions shown in the face more expressive. Therefore it is important to find a way for creating wrinkles in real time applications. Again, there are two ways to solve this problem. One way is to create the wrinkle information beforehand by either capturing it from a real person or by letting an artist create it. This generated static information is then mapped onto the model, similar to a texture. The second way is to create wrinkles dynamically from the mesh, either by letting a person define them in real time, or by simulating forces on the surface. However, a good skin simulation has again to be flexible enough to be used on different head models, and has to work in real time.
In my thesis, I will present a solution, that will allow the creation of soft looking, realistic facial expressions.

2. Related Work

Research in the field of facial animation has already brought up many different solutions to the mentioned problems above. In this section a general overview is provided over the most common techniques, making heavily use of a survey done by Zhigang Deng and Junyong Noh [2].

The oldest and still the most commonly used solution is the use of 'blend shapes', or also known under the name 'shape interpolation'. This technique relies on an artist to create facial expressions through the manual repositioning of the points a 3d model consists of, its vertices. Different parts of the face are deformed and these deformations then stored. Every possible facial action the application needs has to be considered, like closing an eye lid, raising the eye brow, etc. These partial deformations can then be combined to create the desired facial expressions, by assigning each of the created blend shapes a certain weight. The neutral facial expression is then deformed depending on these weights, using some sort of interpolation. Most of the common 3d animation softwares, like Autodesk's Maya, 3D Studio Max and Blender support this technique.

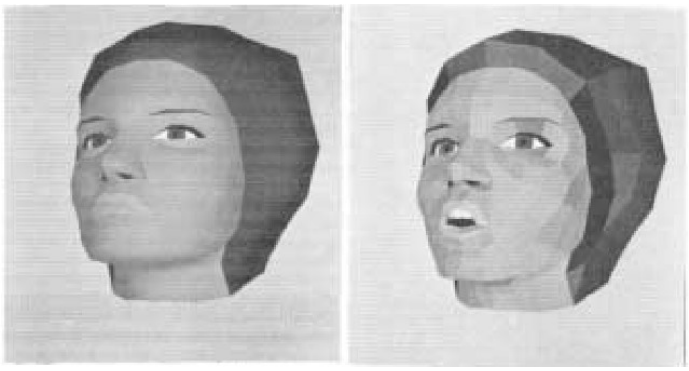

Figure 1: An example of a facial animation by Parke [3].

As a pioneer in this field, Frederick I. Parke has to be mentioned and his paper 'Computer Generated Animation of Faces' [3], in which he proposed this idea of animating faces by interpolation. He compared it to the traditional style of animation, in which the head animator only defines the key poses of a character, and then passes them on to an assistant,

who draws the frames in between. Blend shapes also need an artist to create the key frames, but the positions for the points in the rest of the frames can be calculated. Unfortunately has the use of blend shapes a few disadvantages. First of all it needs a lot of preparation work before being usable in the actual application. A software for the animation of 3d models is needed, and the result heavily depends on the artists skills. The work also depends on the resolution of the model. A high resolution model makes this technique hard to use, because of the high number of points which have to be repositioned. Also, the preparation work has to be done for each individual face, and if the application supports two different characters, the work already doubles. For video games, in which often many different characters have to be simulated, modeling their facial expressions takes a lot of time.

Another solution is the use of Free Form Deformations, which try to overcome some of the drawbacks blend shapes have. It basically approximates the model through a simpler shape, which consists of a few control points. Areas of the model can be deformed by changing the position of these control points. Rational Free Form Deformations expand this technique by adding weights to the control points. A simple mesh cage has to be created to use this technique for producing facial expressions. The cage has control points for all the important parts of a face. For example, there has to be one control point for the left eyebrow, which is then used to raise or lower it.
Escher et al. [4] developed a facial deformation system using free form deformation in combination with the Mpeg-4 video standard, in which Facial Animation Parameters (FAP) and Facial Definition Parameters (FDP) are supported. Their first step was to define a generic head model consisting of 1500 polygons, on which a fixed set of vertices, corresponding to the FDP feature points were put. These control points did not only allow the generation of facial expressions, but also made face deformations possible. Through the manipulation of the control points completely different faces can be generated. They used the Dirichlet Free Form Deformation technique, which was especially developed to allow mesh deformations while keeping the surface continuity.

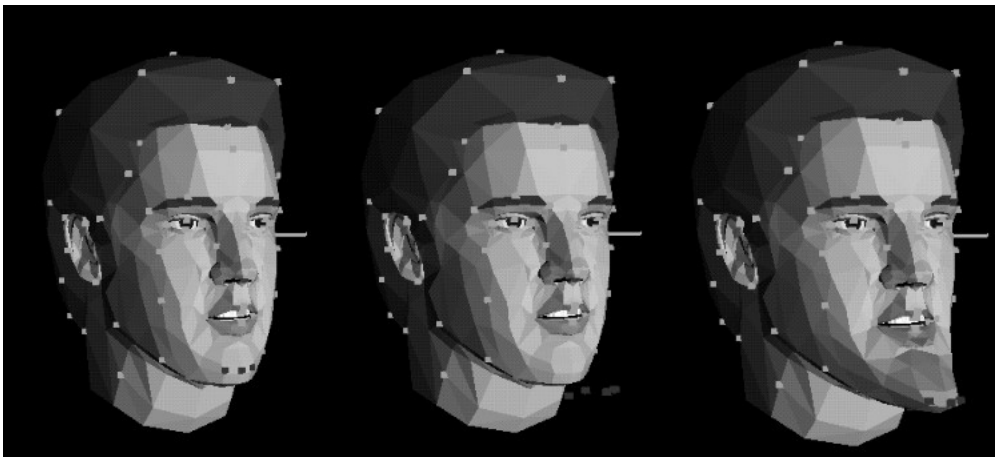

Figure 2: Facial deformations based on FDP [4].

An advantage of this solution over blend shapes, is the independence from the mesh resolution. Also, if the mesh cage is generic enough, it is possible to use it for different face models. However, this technique still depends on surface manipulation only, and does not take the actual facial muscles and their behavior in account.

A completely different approach to the problem is the use of performance driven facial animation. With this method, an actor is needed to create the facial expressions. Simple, but cheap systems can be used to track certain markers on the actors face, and convert them into a desired format. The model on which the animation is applied to has corresponding control points for the tracked markers. These systems often run in real time, allowing the actor to watch his facial expressions on the model while creating them.
In 2007 a real time, performance driven facial animation system, which was also able to create wrinkles dynamically, was presented by Bickel et al. [5]. They divided the task of creating facial expressions into three optical properties, from fine scale over spatial scale to coarse scale properties. Coarse scale properties were the movements of the muscles, for example raising an eyebrow or pulling down a lip corner. Spatial scale properties were the resulting wrinkles that are created due to skin compression, for example on the forehead or around the mouth corners. And fine scale optical properties represent the small details the skin consists of, like skin pores, or freckles.
A commercial scanning software was used to create the face model, which was delivered at a resolution between 500k ~ 700k vertices. Another scanning system, consisting out of 6

cameras was used to track the facial motions. For this task about 80 to 90 facial markers were tracked, which were painted blue dots on the face of the actor. With this setup the facial expressions could be tracked and mapped to the simulated face, but still no wrinkles are generated. To calculate them, the self shadowing of the real wrinkles was used in combination with bright colors, which were applied on the face to make the contrast more even.

A year later in a different paper [6] they proposed a data driven approach for creating wrinkles, based on their previous work. They expanded their work and defined ~100 handling vertices, which can be matched with tracking data or be manipulated by an artist.

The wrinkles were no longer created from real time data, but were taken from an example data base. They captured the position differences of the handle vertices during the facial expression performance of an actor and used these to define a set of strains, representing the skin compression. These strains were then used to calculate the wrinkles.

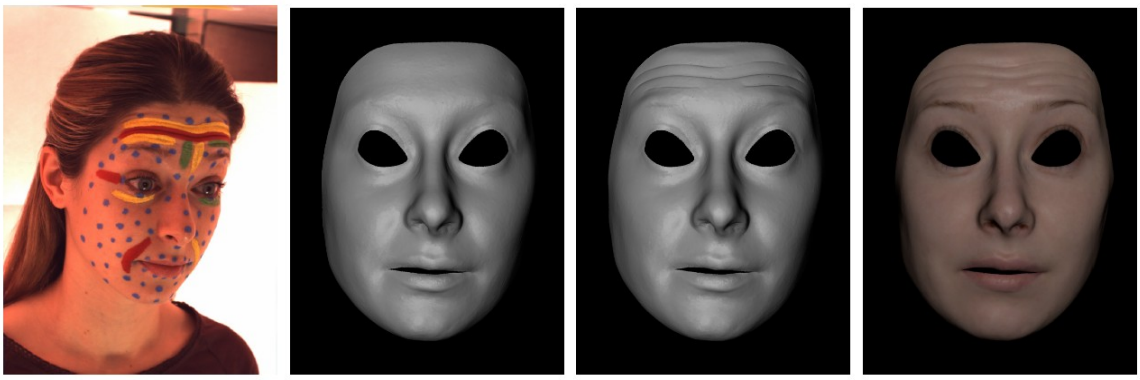

Figure 3: Facial expression tracking by Bickel [5].

A simple example for this technique can be imagined as the raising of an eyebrow, which leads to a length difference between the handle vertices of the eyebrow and the ones on the forehead. The smaller the distance between these points is, the clearer are the winkles visible due to the bulging of the skin. With this knowledge the wrinkles are created based on the wrinkle sample taken from the actors expression. The results that were achieved by using this technique are very impressing,

For even a higher quality performance driven animation, facial motion capture is used. This

technique requires rather expensive equipment, but gives very satisfying results. Usually a high number of cameras are used to track the facial animations of an actor, to either convert these to blend shapes, or to drive a muscle simulation with them. Facial motion capture was used in the animated movies 'The Polar Express' and 'Monster House', and is in general not a common real time solution. It is however often used in the movie industry. Recently though it made its debut on the video games market with Rockstars L.A Noire, in which actual actors and their facial expressions were scanned and then converted into real time clones.

A good documented example of a scanning process was provided in 2009 with 'The Digital Emily Project' [7]. This project, an cooperation between the well known Image Metrics company and the Graphics Laboratory at the University of Southern California's Institute for Creative Technologies, had the goal to create a synthetic face, that would look like and easily be mistaken for a real one.

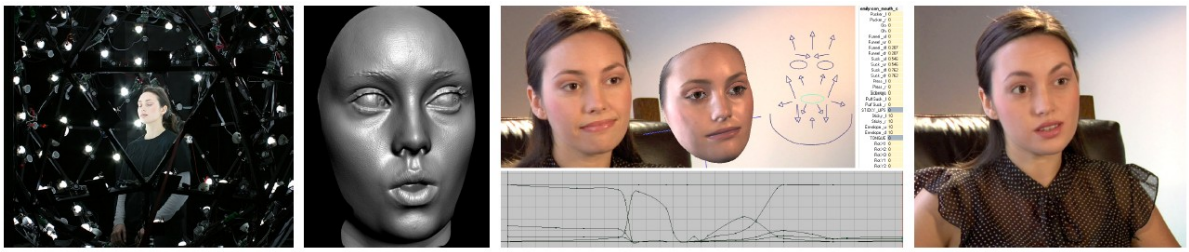

Figure 4: The Digital Emily project [7].

For this purpose a female actor was placed into a lighting cage consisting of 156 LED lights and 15 cameras. This futuristic looking setup confirms, that facial motion capture needs rather expensive equipment to provide the desired, realistic results. For better tracking results 40 small markers were drawn on the actresses face using a make up pen. The actress was then asked to perform 38 expressions, based on Paul Ekmans Facial Action Coding System [8], and multiple photos were taken to capture the face and its skin details, down to the stretching of skin pores. These were then converted into different texture maps, like specular maps, which store the amount of light reflection and normal maps, which save the fine skin details. In the end each facial scan provided a face model consisting of approximately three million polygons.

The next step was to build a lower resolution mesh consisting of about 4000 polygons,

resembling the neutral position of the face scans. Through this process the fine details of the scan were lost, but were put back in later through the texture maps which were retrieved during the scans. Using this model, blend shapes were created from the face scans, by using the marker points which were drawn on the actresses face. These resulting blend shapes then could be driven by a video performance of the actress, and the final video showed tremendous realism.

Another approach was developed in 1987, when Keith Waters presented a way to approximate the behavior of facial muscles, with his Vector Muscle Model [9]. It only consists of two muscle types, the linear or parallel muscle and the sphincter muscle. The linear muscle pulls its surface part to its root, to the point where it is fixed at the skull. it is used to simulate most of the facial muscles. On the other hand the sphincter muscle, which contracts to its center, is only used for the circular muscles around the eyes and the mouth.

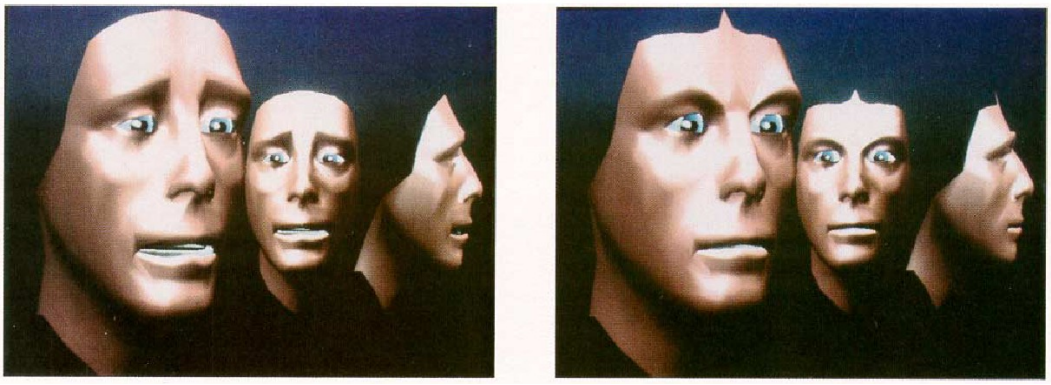

Figure 5: Example of facial deformation with Waters Muscle Model [9].

When visualized, the linear muscle is represented by a cone shaped object. It is defined by two points and an angle, which is representing the width of the cone. The form of the sphincter muscle is an ellipsoid, defined through a center point and the radii for each axis.
It is a simple, but powerful model, and Waters was able to create a face with a variety of different emotions using it.
The advantages of using an actual muscle simulation are, that the resolution of the model has no influence, and that it can be easily adjusted for different faces, since the facial expressions are no longer surface based. Unfortunately there is also a disadvantage, which

is that the positioning of the muscles can often be a time consuming task of trial and error. This is however necessary, since only a small position difference can easily be decisive whether it results into a realistic or unrealistic deformation. This muscle model can also be seen in action in one of Pixars early movies 'Tin Toy'. The face of Billy the baby has forty-seven of Waters muscles integrated in it.

2003 a paper [10] discussing some improvements on this model was presented. It was mainly concerned with the unrealistic vertex deformations, which occur when multiple muscles influence the same set of vertices. A head model retrieved from a 3D scanner was used for this purpose, subdivided into eleven regions. The reduced face consisted of 4744 polygons, and was calculated considering that the expressive regions, for example the forehead, has to consist of more vertices than more static regions, like the back of the head. Additional to Waters muscle model, a pseudo muscle was used to simulate the jaw rotation. When multiple muscles influence a set of vertices, the problem that can occur is, that some vertices leave the zone of influence of one of the muscles, abruptly resulting in unrealistic deformations. To solve this problem, the paper suggests the use of parallelism for the muscle contraction. Instead of applying the full contraction force of each muscle on top of each other, smaller forces are applied until no more contraction force is left. Using this method, more realistic deformations can be achieved when two or more muscles are influencing the same region.

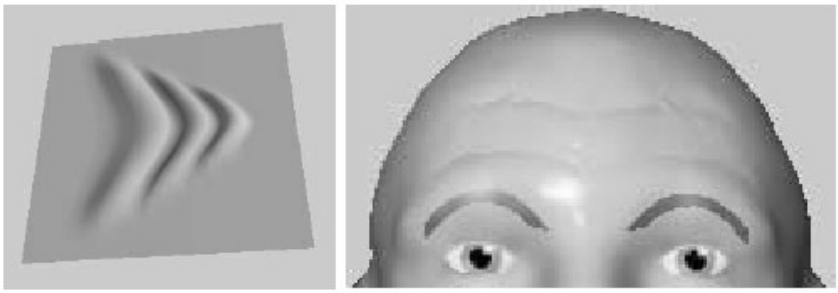

Figure 6: Simple Wrinkle generation [10].

The paper also proposed an idea for creating a simple wrinkle simulation in combination with Waters muscle model. Assuming that the muscles align with the skin, a certain number of wrinkles and their height is predefined for each muscle. Then, after the muscle contraction is applied, the wrinkles amplitude is computed and added to the vertex

deformation. The results prove, that by using this simple technique, the facial expressions achieve a bit more realism.

A different, more complex muscle model was later introduced by Waters and Terzopoulos [11]. The Layered Spring Mesh Muscle Model was designed to model all the anatomical facial features, and consisted of a three layer model. These three layers corresponded to the muscle tied to bone layer, a fatty tissue layer or also called dermis, and the skin layer, the epidermis. The idea behind it was to approximate the face through a point lattice connected by springs. Derived from the actual properties of the human skin, the most outer layer was connected with rather stiff springs, rendering it moderately resistant to deformations. On the other hand the springs in the layer representing the fatty tissue are highly deformable. Very restricted were the nodes connecting to the bone in the lowest layer. The facial muscles then were approximated by defining a point attached to the bone, and one attached to the skin layer, resulting in a muscle vector similar to the ones used in [9]. For the animation of the model, all the muscle contractions are computed and the nodes deformed depending on the weighted sum of their influencing forces.
A similar complex simulation was used by Kähler et al. [12] for the purpose to built animated, anatomically correct head models.

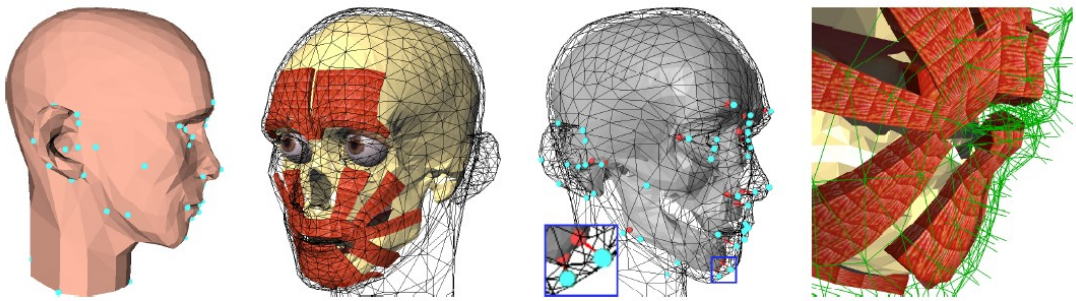

Figure 7: Anatomical correct head [12].

Their model consisted of five major components. The skin surface, which was represented through the triangle mesh, and was built to have more polygons in more expressive regions of the face. Following under the skin was a layer of muscles, modeled as an array of fibers, which were able to contract in a linear and circular way. This layer followed a model

representing the skull, which was only used for the initialization of the muscle layer, and was not present during animation. These three layers were then connected through a mass-spring system, assigning every vertex a certain mass and connecting it to the underlying muscles through springs. Finally, separate models for eyes, teeth and tongue were added.
While creating the head, a set of landmarks was placed on the skin surface with corresponding counterparts on the skull. These were used to generate an offset between the skin and the skull, which is also maintained during animation. To animate the structure, the contraction values of the muscles are changed and the resulting forces directly applied on the skin mesh. Since the muscles are automatically calculated from the space between skull and skin, their contraction is visualized by a recalculation after the skin surfaces deformation.

A commonly used solution to simulate wrinkles in video games, is the use of wrinkle maps. These maps are very similar to normal maps, which are laid over the texture of a characters face when wrinkles should appear. The transparency of the texture can be regulated, to prevent the sudden appearance of the wrinkles, and allow a smooth transition. Wrinkle maps are usually created by an artist during the modeling process of a character. A high resolution sculpture is built for each character, and then based upon it a low polygon version is created. The high resolution sculpture is so detailed that it can provide the wrinkle information of the skin, and the wrinkle maps are computed from it. Unfortunately, this process makes the wrinkle maps depending on the character, however an easy acquisition technique was developed by Dutreve et al. [13]. They describe a solution for acquiring the information needed for the wrinkle maps from a real person and a way to map them to any mesh character. Since their goal was not to acquire the whole face, but only the wrinkles in it, they calculate them using the illumination difference between different poses of the same face.

Figure 8: Wrinkle acquisition [13].

To get the best possible pixel match between the two poses, they developed a deformation algorithm that uses a set of hand placed markers. A Gaussian smoothing is done too, to reduce the noise in the regions of interest, which can be marked with a simple painting tool. The winkles are finally calculated by approximating the light source as coming from the camera direction and assuming that the skin is a diffuse surface. At last an interpolation is done to map the marked space to the equivalent texture space of the character mesh, again using a set of hand placed landmarks.

Yosuke Bando et al. [14] described in their work algorithms for the creation of fine and large scale wrinkles. Fine scale winkles, being small furrows covering the whole skin, were realized through bump maps. These bump maps were created by letting a user specify direction vectors directly on the mesh surface. An algorithm then matched the mesh structure to the direction vectors, resulting into a gray scaled height-map representing the skin furrows. Large scale wrinkles on the other hand were not implemented with maps, but directly modeled into the mesh. The users task is to draw lines again, but this time these represent the actual wrinkles. The wrinkle lines were then converted into a mesh deformation using the mesh structure and a predefined height. The height value specified the amplitude that was used to deform each vertex depending on its normal.

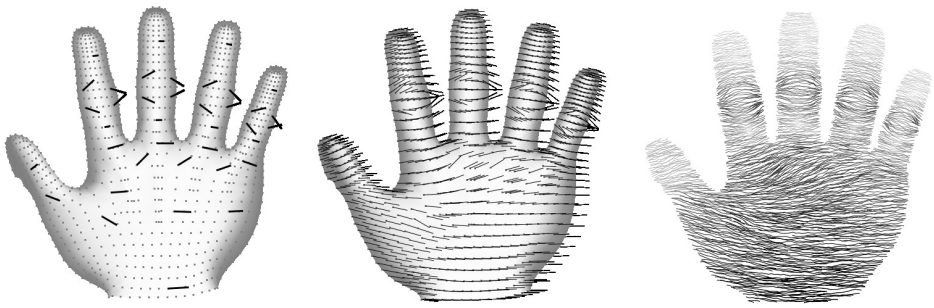

Figure 9: fine scale wrinkles [14].

The deformations were created in real time, allowing the user to immediately see the changes happening. This allowed easy adjustments by redrawing the wrinkle lines or by changing the height parameter. However, this method was designed and only used for static

face meshes so far, and not for the dynamic creation of facial expressions.

Another solution for creating wrinkles due to mesh deformation was presented by Larboulette and Cani [15]. Their approach used a planar wrinkle curve, which gradually wrinkles depending on the distance between its endpoints. This curve is defined through an origin point, a target point and their rest length value. The user is also able to define the number of control points in between, which is equal to the number of wrinkles that are created. When the origin point moves in the direction of the target point, the position of the control points is recomputed to keep the rest length constant, which results into the bulging of the surface.

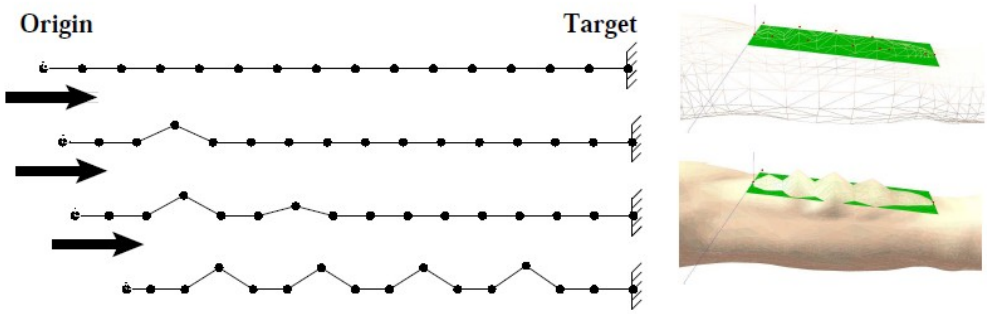

Figure 10: Wrinkle curve [15].

To use the control curve, the user simply draws it over a region of influence, which can be any part of a mesh. The curve then automatically anchors to the underlying mesh, which is done by attaching all the vertices in the influence zone to their nearest control point. Each vertex is then deformed depending on its nearest control point position.

3. Theoretical Discussion

The main goal of my master-thesis is to create facial animations in a way that result into soft looking facial expressions. Therefore the two mentioned problems have to be solved. Finding a simulation for the human skin, that would allow the creation of wrinkles during deformation, and a solution for approximating the facial muscles and their contraction behavior. In my opinion the harder task of these two is to find a suitable skin simulation, which is not only able to create wrinkles dynamically in real time, but also does so while needing as little preparation work as possible and which is also flexible enough to be easily adjustable for different face models.

None of the techniques which I encountered during my research really convinced me as being able to fulfill all of these criteria. Many of the presented solutions needed some preparation work or were in my opinion not flexible enough. Wrinkle maps on one hand require an artist, or with the capturing technique of [13] an actor to acquire the information from. Also the ideas by [15] and [14], needed someone to define the wrinkles beforehand, even though it was the simple task of drawing them onto the mesh. Still, preferably would be a simulation which would actually create wrinkles on its own without the need of any additional work.

In my opinion a perfect solution would be a simulation that would take the given mesh model, and convert it into a skin surface that automatically creates wrinkles when compressed. A glance at related work in the computer graphics field brought up a probable solution. Soft-body simulations, which are often used to simulate clothing, were able to create realistic wrinkles for any kind of fabric. They provide the kind of dynamic wrinkle generation which the simulation of human skin needs too. However there are some differences that have to be addressed. Clothes react to external forces, like gravity and wind, and hang rather loosely on the human body. The human skin on the other hand should not move at all, until triggered through muscle contraction, and also should not have that 'jiggly' effect, most cloth simulations produce. A modified soft-body simulation was therefore chosen to be implemented to simulate human skin behavior visually sufficient.

Because of this decision, to simulate the human skin by calculating forces on the surface, a special solution was needed for the animation of the face model. This solution is not allowed to deform the surface of the mesh directly, but has to simulate forces that can be integrated into the skin simulation. The general requirements however, that it has to be flexible enough to be easily adaptable for different head models and fast enough to run in real time, do not change.

Blend shapes were eliminated rather fast, fulfilling none of these criteria, and also performance driven animation due to its additional gear and preparation. Free form deformation was considered as an option, but in my opinion preferable was an actual simulation of the facial muscles, and not only of the resulting surface deformations. The best choice seemed therefore to be one of the muscle simulations I encountered during my research. While the three layer muscle model used by [11], and the one used by [12], are reproducing the facial muscle contraction and the visual aspects on the surface very accurately, it seemed a bit calculation expensive to be used in real time.

After this elimination process the one remaining model was picked, which was Waters Vector Muscle Model [9]. His model, consisting of only two types of muscles, does not only entice due to its rather simple design, but also due to its proven powerful performance.

Still missing though was the needed knowledge of the positions and the forms of the facial muscles. A guideline was needed, which would be the foundation for the muscle simulation. During my research I came across two possibilities, the Mpeg-4 model [4] and the Facial Action Coding System [8]. However, the Mpeg-4 model was rather fast eliminated because it only defines a set of feature points. These points can easily be converted to any mesh model, but are not of much use for a muscle simulation.

Therefore the Facial Action Coding System (FACS) created by Ekmann et al. was chosen, which was also used in Waters original paper. This manual does not only explain the position of the facial muscles accurately, but also focuses on their visible deformations, which are described through Action Units. By combining these Action Units, every possible facial expression can be created. The Facial Action Coding System was therefore chosen to provide the foundation for the muscle integration, and to help with the extraction

of the most important muscles, which have to be implemented to allow a wide range of facial expressions.

3.1 Skin Simulation

The developed skin simulation was derived from a simple soft-body simulation. Some of its basic principles, such as constraints and their iterative solving where kept, while others, like springs, were discarded. In the following section, I will begin by shortly explaining what exactly a soft-body simulation is and how it works, on the example of a mass spring simulation, and will then continue to discuss what changes had to be done, to get a visually sufficient skin simulation as result. All the modifications of the simulation were done, considering the performance of the final application.

3.1.1 Softbody Simulation

A soft-body simulation is a type of physic simulation that allows the, at least visually correct, physically simulation of soft objects, such as a piece of jelly or gum. It works similar to a regular physics simulation, which uses a set of external and internal forces to move an object realistically through a virtual world. Common external forces are gravity, air resistance or buoyancy which are derived from the real world. Internal forces are for example the weight and mass of the object, or its size. Important is also the collision with other objects, which also produces forces that stop or redirect the movement of the model.
All these forces and more importantly their impact on the object are calculated and then combined and processed to get the objects velocity as a result, which is added to its position. This delivers a new position, the current one for the object. The calculation and the update of the models position is done periodically, which finally results into a visible movement.
This is how a general physics simulation works. The difference to a soft-body simulation is, that whereas in a physics simulation the simulated forces influence the whole object, in a soft-body simulation every vertex of a model is considered an individual object. So each

vertex is assigned a mass and a new position is calculated depending on the applied forces. This results in soft looking deformations, for example when the bottom of a ball hits the floor. The movements of the lower vertices suddenly stop, while the upper ones still move due to gravity. Visually the result would be a compression of the lower half of the ball. There have to be some additional internal forces that prevent the vertices from passing through each other and maintain the form of the model. However, because of these resulting deformations, the object gives away a soft feeling that can simulate soft objects like a piece of jelly very accurately from the visual point of view. An even more common use for this simulation is in the implementation of virtual clothes.

Similar to virtual human skin, simulated clothes only look realistic if they form wrinkles upon deformation. A cloth simulation has therefore be able to do three things. Allow the deformation of the model by external forces, such as gravity. Maintain the volume of the cloth, so that upon deformation the object deforms but does not change its size in the process and provide some sort of distance preservation, so that neighboring vertices cannot pass through each other.

A simple way to create a soft-body simulation is by using a mass-spring simulation. This type of simulation assigns each vertices of the mesh a certain mass, and connects all the vertices with springs. This results in a net of individual objects, who are only connected loosely to each other. The assigned springs are derived from springs in the real world, and try to keep their original length, which is equal to the distance between the two involved vertices. If compressed, the spring develops a force to move the vertices in a direction that allows it to return to its rest length.

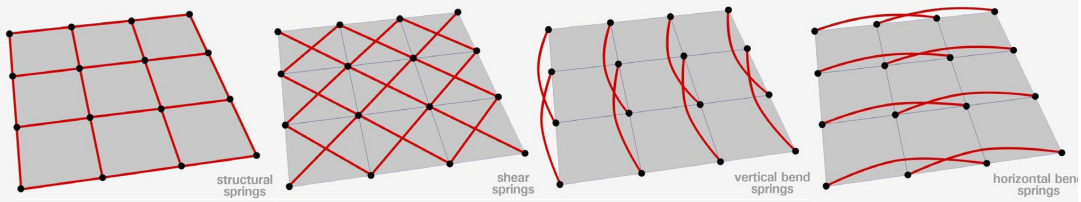

Figure 11: Cloth simulation springs [23].

Usually three types of springs are integrated into a cloth simulation, which can be seen in

Figure 11. Structural springs are integrated between neighboring vertices and are used to keep the horizontal and vertical distance between them. Shear springs are placed between diagonally positioned vertices to prevent a cuboid of a mesh to form a flat diamond. Bend springs or flex springs are placed horizontally and vertically between vertices which have one vertex in between them. So they are not direct neighbors, but two rest lengths away from each other. These help to define the bending stiffness of the cloth, resulting in ether a silky or leathery fabric. Integrating these springs implies that the mesh has an even topology. Additionally to these springs, usually a set of constraints are implemented, most commonly are point and length constraints.

A point constraint pins a certain vertex to a point in space, which is useful to define areas at which the cloth is pinned to an object, and should not deformed at all. Common spaces would be the collar of a t-shirt or a side of a flag bound to a pole. Length constraints are similar to springs, but try to keep the distance between the two involved vertices at the same length avoiding any sort of stretching. They are very important for cloth simulations and are needed to get rid of the jiggly feeling that result from the use of springs.

If now a force, like gravity, is applied to the simulation and the vertices start to move, a lot of constraints and springs will have to develop forces to keep their optimal positions. This is a spreading process, so if one spring moves a vertex to keep its rest length and with it the optimal distance between its two assigned vertices, it is probably going to change the length of a neighboring springs rest length. Now the neighboring spring has to develop a force too, to maintain its rest length, and this continues through the whole simulation.

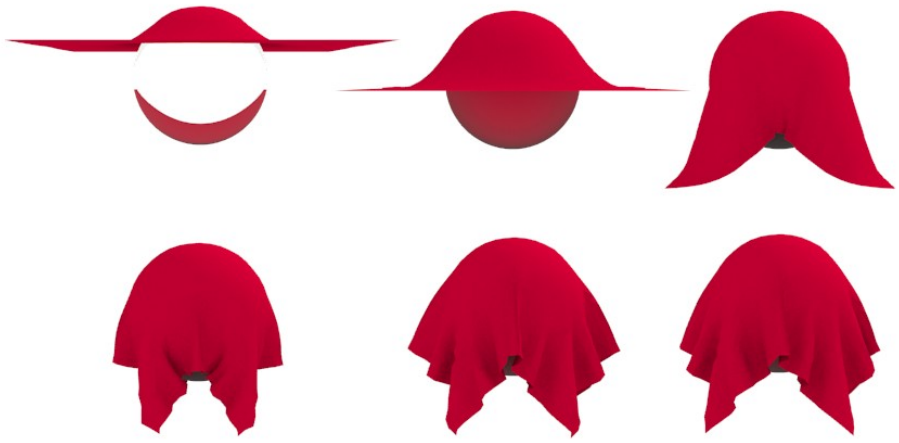

Figure 12: Example of a cloth simulation done in Blender.

The key to solve the external forces and to get the optimal positions for most of the vertices is a numerical integrator. This integrator is responsible to iterate through all the vertices and calculate their new position. It considers all the relevant forces and computes through numerical integration the resulting velocity. This velocity then can be easily applied to the vertex position, which results into an updated position. Very common integrators are the Euler' Integrator, which is a relatively fast one. More complex is the Runge Kutta integrator, which also takes longer to calculate. It also delivers better looking results however.
Unfortunately, a soft-body simulation can easily consist of thousands of objects, depending on the resolution of the mesh, and it can take a while to calculate all the necessary positions.

3.1.2 Modifications

For the purpose of simulating human skin, the soft-body simulation has to be modified. The main difference is that the human skin does not hang loosely around like clothes tend to do. External forces therefore do not have much of an influence and do not, in my opinion, have to be simulated at all for expressive wrinkles. For aging wrinkles it would be interesting to see how gravity has influence on the looser getting skin, and if it could be simulated in real time with a modified cloth simulation. For this thesis however the focus lies on expressive wrinkles. For this thesis the assumption is made that the facial skin only moves when a deformation occurs due to muscle contraction.
Skin can be stretched to a certain amount, but is elastic enough to return immediately to its original state as soon as the force is gone. In other words, every vertex of the mesh should stay in its rest position, until it is deformed due to muscle movement. Unfortunately this can not be directly simulated by using the earlier mentioned point constraints, since then the vertices are then fixed on their initialization position and would never be able to move. We on the other hand need constraints that develop a force pulling the vertex back to its original position as soon as the vertex leaves it. This simulates the skin elasticity sufficient enough for this purpose.

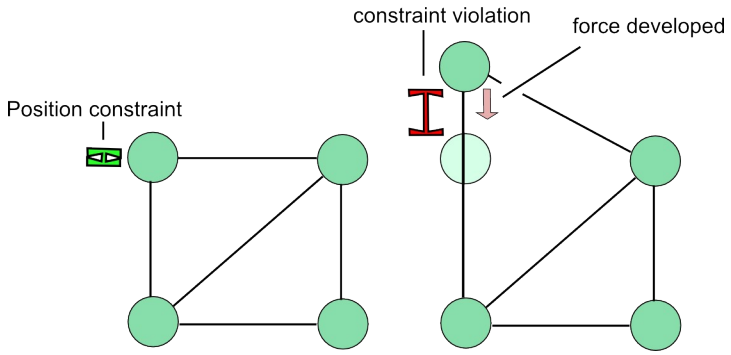

Figure 13: Force development of position constraints.

Therefore position constraints were developed, which are actually more similar to length constraints than to point constraints. However, while length constraints are implemented between two vertices with a calculated rest length, position constraints use the initialization position and the current position of a single vertex. The desired rest length always has the value zero. Therefore, as soon as a vertex moves, a force is developed that tries to pull the vertex back to its original position. The amount of these constraints that have to be integrated into the simulation depend on the resolution of the mesh. For each vertex one position constraint is added.

Position constraints simulate the elasticity of the skin, but we also need a force that handles skin compression. When the skin is compressed, wrinkles appear due to the incompressibility of skin. This results into a visual effect likewise to the one presented by [15] and can also be similarly implemented. It can be achieved by using strict length constraints, which always try to keep the vertices at their optimal rest distance. So if the two involved vertices are too close, forces are developed that try to push the vertices in a direction that allows the constraint to achieve its rest distance. When the position change spreads through the simulation, it usually only leaves the surface the option to bulge, and thus wrinkles are created. Between every vertex and its neighbor, length constraints were therefore implemented. The amount of these constraints is also depending on the mesh resolution, since each vertex has usually between three to eight neighbors.

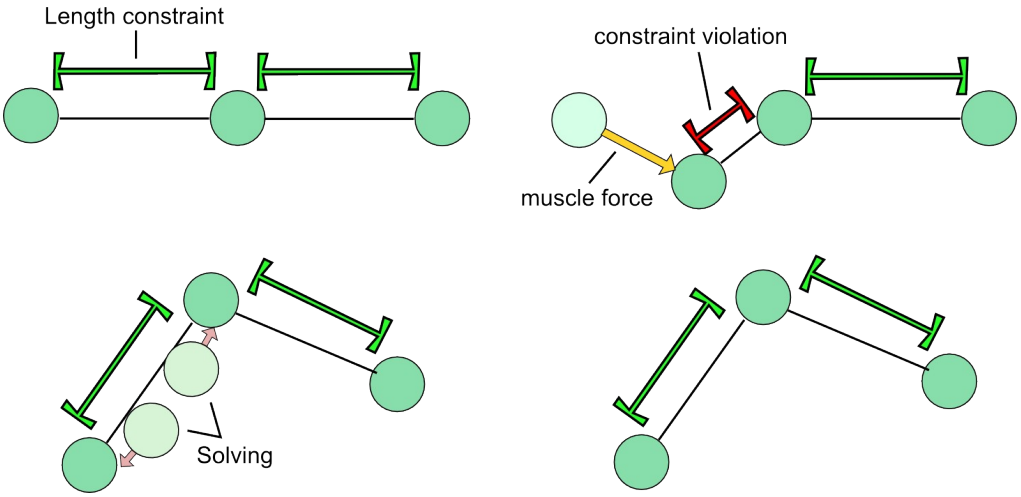

Figure 14: Solving of a length constraint violation.

Position and length constraints actually succeed already in simulating the visual properties of human skin sufficiently, integrating springs in the simulation would only make it more computation expensive. Structural springs for example are used to keep a minimal distance between neighboring vertices. In our simulation, this behavior is approximated by the length constraints, who are responsible for keeping an exact distance between vertices, and who pretty much take over the work of very stiff structural springs, resulting in a desirable incompressible surface. Shear springs keep diagonal vertices apart, which is important to prevent the faces from deforming into flat diamonds. But this danger only exists, when the mesh is hanging loosely and when it is under the influence of an external force. Since the human skin simulation used is not influenced by external forces at all, and the position constraints keep the vertices at their place, the danger of deforming into flat diamonds is not given. This makes the use of shear springs unnecessary. Also, since the mesh for a human face is very rarely of even topology, bending springs would produce very different results, depending on the part of the model that currently deforms. To produce small wrinkles a very low bending stiffness is needed too, which can be approximated by the value zero. However, they would be useful to implement different skin behavior, which would allow the comparison between the skins of elderly people, which create wrinkles more easily, and the skin of a baby, which creates fewer wrinkles. Unfortunately, this behavior would also be depending heavily on the mesh resolution, and as mentioned earlier the focus of this thesis lies on expression wrinkles.

One more class of constraints is needed, which has to simulate forces generated through muscle contraction. These constraints cannot be derived from the mesh directly, but have to be added into the simulation later by the virtual muscles. The amount of these constraints therefore depends on the number of simulated muscles. These muscle constraints are implemented between a vertex position, and a target position that is provided by the muscle simulation. Depending on the contraction strength of the muscle, a force is generated that pulls the vertex towards the target position. As soon as the contraction stops, the muscle force vanishes, allowing the vertex to return to its original position.
The final result is actually a very simple skin simulation, consisting of three types of constraints, which are used all over the face mesh and result into wrinkle generation during the deformation. These wrinkles are unfortunately depending on the mesh topology, but need no preparation or user interference at all.

3.2 The Muscle Simulation

For the implementation of facial deformations due to muscle contraction, a simple muscle simulation was chosen. Although this muscle simulation, consisting of two types of muscles, is very flexible, some modifications had to be done. In the following section I will start by giving an overview over Waters muscle simulation, and by explaining the principles behind it. Then in the end the modifications of this simulation are described, as well as the reasons for them.

3.2.1 Waters Vector Muscle Model

For the simulation of the facial muscles Waters Vector Model was chosen, due to its, in my opinion, simple design but powerful performance. A simple design because it only needs two types of muscles, with which a wide variety of facial expressions can be created. These two muscles are the linear muscle and the sphincter muscle.

3.2.1.1 The Linear Muscle

This muscle is used to approximate all the facial muscles that pull on a part of the skin surface, which is the majority of them. Linear muscles are described by one end that is attached to the bone, and another one embedded into the skin. While the end attached to the bone, in this paper called the origin of the muscle, remains static during contraction, is the other end pulled towards it. This contracting end is embedded into the skin and therefore influences a certain skin area, its zone of influence. During contraction the muscles zone of influence is pulled towards its origin. This behavior simulates the isotonic contraction of a muscle sufficiently.

In Waters model the origin of a linear muscle is defined through a single point, while the zone of influence is approximated by a circle. This muscle can therefore be described by 3 parameters. The first one is a point in space for the origin, the second parameter another point for the contracting end, and finally an angle which describes the width of the muscle, and at the same time the area for the zone of influence. From the two defined points the direction of the muscle can be calculated, which always runs from the origin towards the contracting end. The size of the muscle depends on one hand on the distance between the defined points, and on the other hand on the defined angle. If the muscle is visualized it would have the form of a sphere segment, ranging, depending on the angle, from half a sphere to a thin cone shape.

These are only the parameters we need to determine the position and the form of the muscle however. To simulate its contraction behavior, additional factors are needed. Important is the magnitude of the muscle, that tells us the contraction strength at each point on the muscle. The contraction strength is not evenly distributed on the whole muscle but fades away towards the origin point. In other words, the magnitude is zero at the bone attachment, and gradually increases to its maximum at the zone of influence. The force of the muscle contraction is simulated to dissipate because of the adjoining skin layers. This happens not only in the direction of the muscle, but also in its width. To calculate this behavior, two additional forces, the radial and the angular displacement have to be computed.

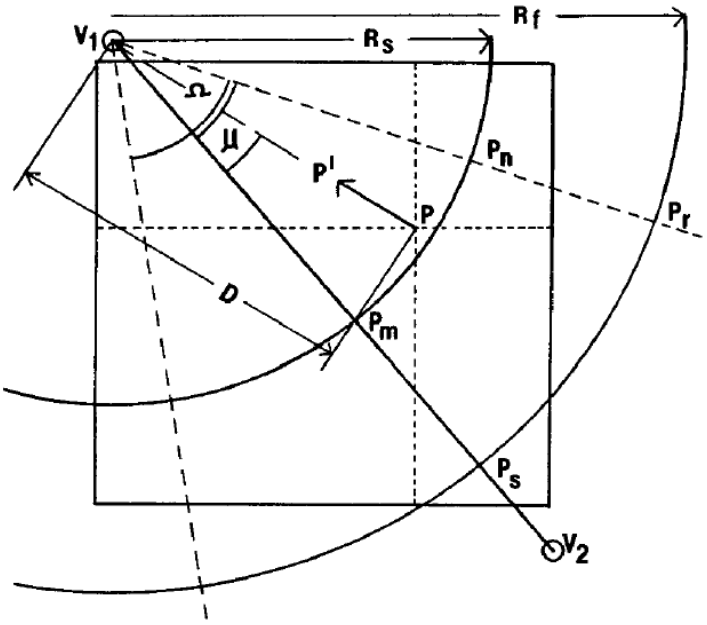

Figure 15: Waters linear muscle in a 2D space [9].

Figure 15 shows the linear muscle applied in a 2 dimensional space which is enough to understand the principles. It can easily be implemented in the same way in a 3D simulation. V_1 defines the origin of the muscle, and V_2 the center point of the influence zone. The whole muscle is defined by a circle segment, with the radius $\overline{V_1 V_2}$ and the angle Ω. For the calculation of the magnitude, two points which represent the distance from the origin, are defined. R_s represents the distance from the origin where the magnitude falloff starts, and R_f the distance the falloff ends.

For every vertex that is inside the influence zone of a linear muscle, its new position can only be computed by calculating the angular and radiant displacement value for it. In the example shown in Figure 15, a new position p' is calculated for point p by the formula

$$p' = f(p \cdot K \cdot A \cdot R)$$

K stands for the muscle spring constant, which is equal to the contraction strength, A is the angular displacement value, and R the radial displacement value.

The angular displacement value simulates the force dissipation to the side and is calculated by

$$A = \cos\left(\left(\frac{\mu}{\pi}\right) \cdot \left(\frac{\pi}{2}\right)\right)$$

where μ stands for the angle between $\overline{V_1 V_2}$ and $\overline{V_1 p}$.

The radial displacement simulates force dissipation in the inverse direction of the muscle and is approximated for nodes inside $\overline{V_1 P_m P_n}$ by

$$R = \cos\left(\left(1 - \frac{D}{R_s}\right) \cdot \left(\frac{\pi}{2}\right)\right)$$

where D stands for the distance from the point p to the origin V_1. For vertices inside $\overline{p_m p_n p_r p_s}$ the radial displacement is calculated by

$$R = \cos\left(\left(D - \frac{R_s}{R_f} - R_s\right) \cdot \left(\frac{\pi}{2}\right)\right)$$

These calculations can be adopted into a 3 dimensional simulation as they are, the only difference is that the 2D vectors change into 3D ones.

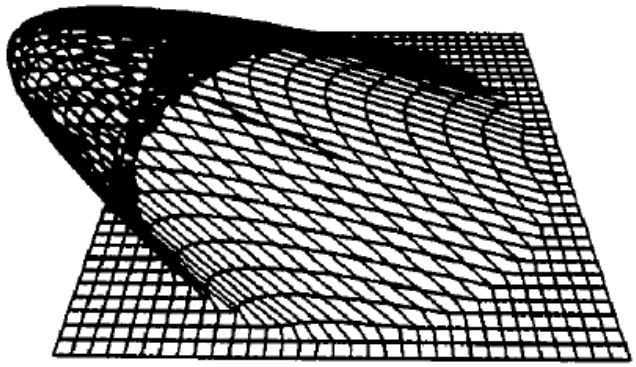

Figure 16: Waters linear muscle example deformation [9].

3.2.1.2 The Sphincter muscle

The second type of muscles Waters defined is the sphincter muscle. This muscle is completely different from the linear muscle in shape and behavior. It has a circular shape and instead of pulling on the skin surface, it pinches it towards the muscle center. These muscles cannot be found as often as linear muscles in the human face, and exist in fact only three times. Around each of the eyes and around the mouth. But they are nonetheless very

important to simulate actions like the puckering of the lips or the pinching of the eye.

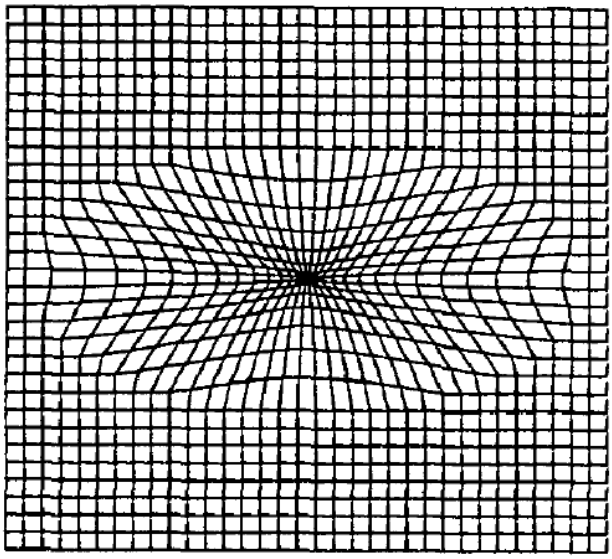

Figure 17: Sphincter muscle contraction example [9].

To define this muscle at least two parameters are needed. A point that defines the center of the muscle, which is also the origin to which the skin surface is pinched towards, and a radius. This gives us a perfect sphere. However, often it is more advantageous to use a ellipsoid instead of a sphere, to approximate the mouth and eye area more realistically, which means that this muscle needs two or three radii.

Again, for the contraction behavior a magnitude is needed that defines how strong the contraction influences the skin surface. Since the sphincter muscle is a sphere, the angular displacement is no longer needed. Sill left is the radial displacement, because the contraction force should still become less, the nearer the influenced point is towards the center. The radial displacement is calculated similar to the one for the linear muscle. For vertices inside $\overline{V_1 R_s}$ by

$$R = \cos\left(\left(1 - \frac{D}{R_s}\right) \cdot \left(\frac{\pi}{2}\right)\right)$$

and for vertices inside $\overline{R_s R_f}$ by

$$R = \cos\left(\left(D - \frac{R_s}{(R_f - R_s)}\right) \cdot \left(\frac{\pi}{2}\right)\right)$$

D stands again for the distance from the point p to the center point V_1. The formula for the contracted position p' of the point p is

$$p' = p \cdot K \cdot R$$

3.2.2 Modifications

During implementation, some difficulties were encountered for simulating the opening of the mouth, and the opening of the eyelids. The opening of the mouth is a muscle contraction that triggers the rotation of the lower jaw around a joint positioned near the ear, and is therefore hard to simulate with the before described muscles. Another problem was that approximating the behavior of the upper eye lids with linear muscles provided unrealistic results, rather looking as if the lids would be pulled into the head, instead of opening. If the lid is approximated by a quarter of a sphere, it would be more realistic to rotate it around the sphere center to simulate an eye opening, instead of pulling it upwards.
Therefore Waters muscle model was slightly modified and a third type of muscles was added, the rotation muscles.
These muscle rotate the influenced vertices around a predefined joint. Therefore upon contraction they develop a rotation force, that is equivalent to the degree of the rotation. The strength of the rotation on the individual vertex also depends on its distance to the joint position. If the vertex is further away, it is stronger influenced from the rotation.
Two types of rotation muscles were implemented, though their main difference is only the shape, with which they define their area of influence.

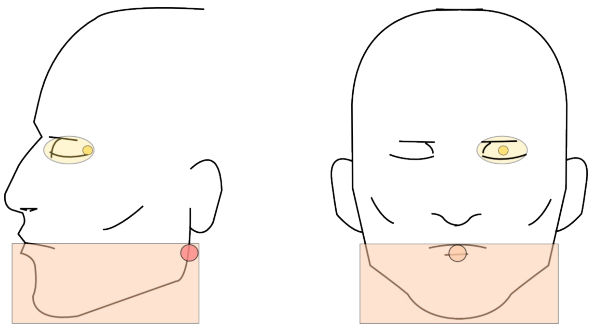

Figure 18: Rotation muscles area approximation

For the simulation of the jaw muscle, the area in which all involved vertices are positioned can be approximated by a cube. The rotation joint is at the back side of this cube. The upper face of the cube is be at the same height as the lower lip, since upon jaw rotation only the lower lip moves. Assuming that the provided face model has its eyes closed in the neutral position, it is easier to capture all important vertices for the upper lip with an ellipsoid. The rotation joint is at the back side of the ellipsoid too.

3.3 Facial Action Coding System (FACS)

The FACS, firstly published in 1972 by Paul Ekmann and Wallace Friesen, is a manual that describes the human facial expressions, and how they are achieved by their underlying muscles. For that purpose, they defined the muscle behavior in Action Units (AU), which either represent a single or a small group of muscles. This categorization depends on the visual impact the muscles have, and if the muscles have to work together to create a certain facial deformation or not. Action Units were created to parametrize the face into a set of visible muscle actions, which can be combined to create any facial expression possible. Each of these Action Units is described by words on one hand and visually presented on the other through a set of still images, and a short video clip. To differentiate the Action Units, each is assigned a number and a name describing its functionality. For example AU1 would be the 'inner brow raiser'. The numbers for the action units were assigned arbitrary and do not follow any certain logic.

All the muscles responsible for an Action Unit are additionally explained by two images, one for the muscular anatomy and a second one for the muscular action. The muscular anatomy picture shows where each muscle is located as well as the size and the form of it. In these images each muscle is drawn anatomically correct and very detailed upon a photograph of a face. A number indicates the Action Unit the muscle belongs to.

To understand the images that show the muscular action it is necessary to be familiar with the muscular anatomy, since the location of the muscle is only drawn schematically. In these images the number of each Action Unit is written in small circles, which position also approximates the area from which the muscle emerges from the bone structure. From this

position a black line is drawn to the area where the muscle is attached to the skin of the face.

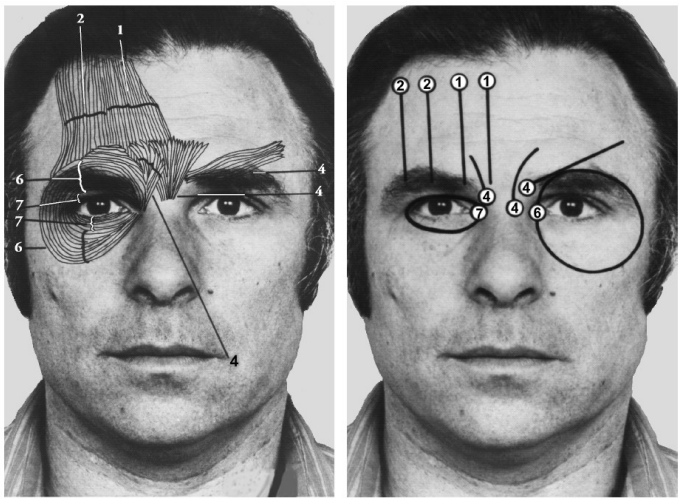

Figure 19: Muscle anatomy (left), Muscle action (right) [8].

The description of the Action Units is divided into the ones for the upper face, and the ones for the lower face. While the upper face Action Units are responsible for eyebrows, forehead and eyelids, the lower face Action Units are again subdivided into five groups. Up/Down, Horizontal, Oblique, Orbital, and Miscellaneous Actions.

The description of each Action Unit is also subdivided into three sections. The most important section for this work is the first one, called 'Appearance Changes'. In this section all the visual deformations that may occur due to this Action Unit are described, coarse deformations for example the raising of the eyebrow as well as finer scale deformations like the appearance of wrinkles. The second section, 'How to', gives a short tutorial on how to perform the action units, with a few tips if they appear difficult or impossible to do by yourself. In the third section, the 'Intensity Scoring', the different intensities of each Action Unit and its consequences are described. For this purpose a scoring mechanism consisting of five letters, from A to E was used. A being the slightest deformation and E the maximum.

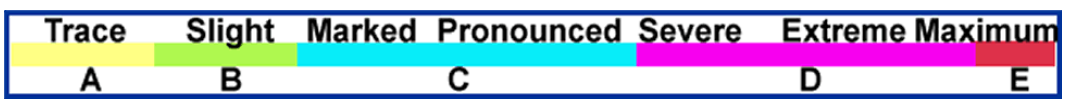

Figure 20: FACS Scoring Mechanism [8].

Depending on the intensity, different visual aspects can appear upon the muscle contraction.

Usually, each Action Unit described in the Facial Action Coding System is symmetric, and appears on each side of the face. But for the few Action Units that are unilateral and can occur on only one side, the abbreviation 'L' for left and 'R' for right are placed in front of the AU number. Altogether a total of 46 Action Units are described in the manual. For the purpose of thiss thesis not all Action Units were necessary, so only the most important ones were implemented.

4. Implementation

This chapter covers all necessary aspects, which have to be known for the implementation of the facial expression application. It starts off by giving an overview over all the tools that were necessary. These tools were chosen under the following aspects:

- **Open Source / Freeware:** Since everybody should be able to recreate the presented application, the used tools had to be freely available to everyone.
- **Multiplatform:** To be able to run on multiple operating system was another desirable aspect, since the application should be made available to as many people as possible.
- **Experience:** Most of the tools presented were already used in other projects, and I had therefore some experience in their use and knowledge about their limitations.

It then continues with the description of the skin and muscle simulation implementation, as well as with all the implemented Action Units from the Facial Action Coding System. Some additional problems that had to be solved, such as the update of the vertex normals and the visualization of the muscles are explained too.

4.1 Tools and Data

In this section an overview over all the used tools is given. Most of the tools are either freeware or open source software and therefore available for anybody. In the following pages, their general functions are described and their use for this project.

4.1.1 Ogre Software Development Kit (OgreSDK)

This graphics engine [16], is one of most widely used open source game engines and can be used on Microsoft Windows, Linux and Mac OSX. It uses the programming language C++ and supports as programming tools Microsoft Visual C++ as well as the free alternative Gtk++. The used format to represent 3d objects is the .mesh format, which is a self developed format and therefore only supported by few other programs than Ogre.

However, there exist already extensions for 3DStudioMax and Blender to support the export of models into the Ogre .mesh format.
Ogre allows the use of Direct3D as well as Open GL for the in game rendering. It uses a graph oriented structure for the rendering of a scene. There is one root node, the 'SceneManager', and each created object has to be attached to one of its child nodes to appear visible. Loading a mesh model into the application is therefore not enough. Also notice worthy is that there is no interface for the OgreSDK, it only provides libraries and content for the creation of an application. Very important for this work was the provision of a math library, with predefined 3D vectors and vector calculation operators, which made the development of the simulation easier.

4.1.2 Ogre Application Wizard

This small but very helpful tool [17] allows the automatic creation of a basic Ogre application in combination with Visual C++. The tool initializes the most important libraries, for example the OIS.lib for keyboard recognition and of course the Ogre.lib. It also generates the render window with a simple menu to choose from options for settings like the resolution or antialising, and provides a camera as well as an example model, the ogre engines mascot-like ogre head. It is nice to be able to create this basic setup up by pressing on a button, especially if you work a lot with the ogre engine.

4.1.3 Head Model

To test the simulations a face model was needed. Thankfully, during my research I stumbled across this free available male scan, in form of a bust that was delivered in .obj format and was detailed enough to allow the creation of polygonal wrinkles upon deformation. It is provided by Lee Perry-Smith under a creative common license [18].

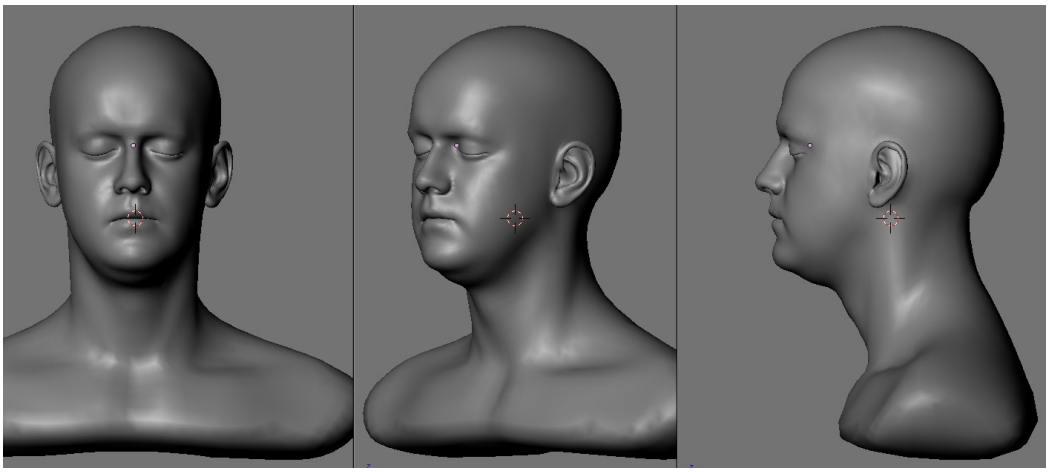

Figure 21: Unmodified Head model

4.1.4 QuickGUI

This freely available graphical user interface library [19], works well together with the ogre engine and provides a wide range of interface items. As the name states it, this tool is easily and quick to set up and and provides simple commands for the creation of its items.
It works under Windows and Linux and was chosen because of its easy integration into Ogre.

4.1.5 Blender

This is in my opinion the most powerful open source 3D graphics tool [20] out there. It supports not only modeling, but also texturing, animation and rendering of the created model. Blender is also able to work with a lot of different import and export formats. Under these is also the .obj format, which was needed to import the provided head model.
For this work however, only a small part of this tool was used. Its main purpose was the modification of the head model. Since the model was a scan of a male head with the neck and the upper part of the shoulders still attached, some parts were removed.
A different problem that the used model provided for the simulation was, that the upper and lower eyelids were connected with each other as well as the lips of the mouth. Therefore to allow the muscles to open the eyes and the mouth of the face, these connections had to be removed. The following modification were done.

- Removing the oral cavity

- Removing the eye sockets
- Removing the neck and shoulders of the model
- Removing the back of the head – to gain a better performance
- Removing the connecting vertices between the eyelids
- Removing the connecting vertices between the lips

Finally some of the vertices at the edge of the face were repositioned to give the edges a smoother look. Also, since the head scan was altered to be able to open the mouth and the eyelids, two additionally models were created to give the face more realism. A teeth model, used for the upper and lower teeth, was placed into the mouth, and an eye model into each eye socket.

4.1.6 Blender Ogre Exporter

This extension for blender is a python script that allows to export any model from blender into the ogre .mesh format [21]. Once the python script ran, the .mesh format is available under the export options of blender. This tool also needs the OgreXmlConverter [22] to work, which creates a .xml file from the models information. This .xml file is then used to transform the information into a .mesh file and a .material file for the texture. If the model is animated an additional .skeleton file is generated for the animations. The script automatically converts the blender cuboid polygons into the .mesh triangles.

4.2 Implementation Overview

The implemented application is built from a set of connected modules. There is the mesh model, which is the visible part of the simulation. Another module is the skin simulation, which purpose is the creation of wrinkles, and the muscle simulation, which animates the face model. To get the system running, a model has to be added, which has to be provided in the Ogre .mesh format,. Since this application was created to simulate human facial expressions realistically, a scanned human face model is ideal, but any model should do. The only property the face model needs to fulfill, is having a decent resolution, which is

important for the wrinkle generation. If the model is too coarse, the muscle simulation still works, however without the wrinkles it is not resulting into the desired soft looking facial expressions. From this added face model, the information needed to create the skin simulation is extracted automatically, and the simulation is then generated. For each of the vertices in the model, the information about its position and its neighboring vertices are gathered, to create the position and length constraints. How many of these constraints are added into the simulation depends on the mesh resolution.

The next step is to add all the necessary muscles to the simulation. Position, size and orientation of all the linear, sphincter and rotation muscles have to be defined and fit to the face model. Depending on the complexity of the desired muscle simulation, this can be a very time consuming process. Fortunately, all the important muscles are already implemented in the application and can be easily modified with an integrated editor, which should ease the work a bit. From these created muscles, the muscle constraints are generated automatically and added to the skin simulation. The amount of the muscle constraints depends on the number and size of the integrated muscles, since for every vertex influenced by a muscle, one muscle constraint has to be added.

Now the skin simulation is final with all the necessary constraints. The position constraints to keep the vertices at their original position, the length constraints which allow the creation of wrinkles dynamically, and the muscle constraints, that allow the muscle to deform the simulation upon contraction, and to form the facial expressions. If the simulation is started, the following steps happen upon changing the contraction value of one of the muscles.

- The muscle contracts
- Update muscle constraints
- Iterate through all constraints and solve constraint violations
- Calculate new vertex positions
- Update mesh model

After these steps are done, the deformation is visible on the face model, and all the constraints satisfied.

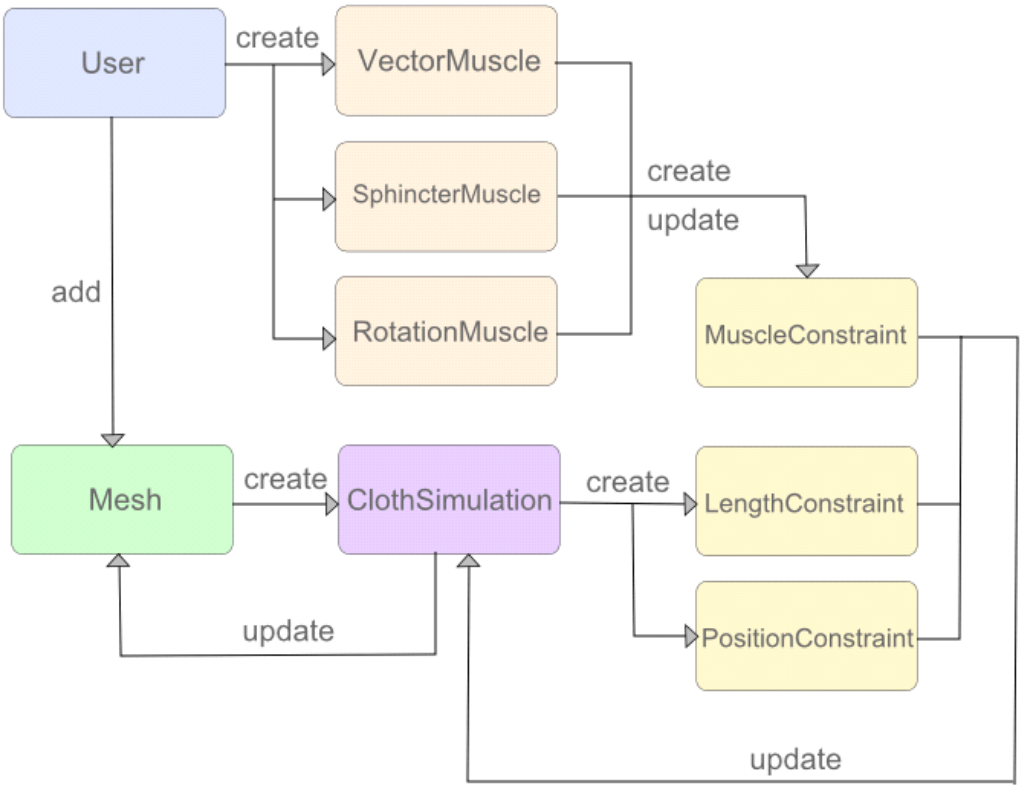

Figure 22: Application overview diagram

4.3 Implementation of the skin simulation

The skin simulation was derived from the simple cloth simulation described in [23], but changed a lot from the original during implementation. It was made less calculation expensive by eliminating the unnecessary external forces and springs as described in section 3.2.2, which also made it more suitable for higher resolution models.

4.3.1 Simulation Vertices

To build this simulation we need objects that represent the vertices of a mesh. Therefore our skin simulation consists of an array of so called simulation vertices. These have only two tasks, to get the position of their influenced 'real' mesh vertices, and to update this position later on. To do this, it is important to understand how Ogre is storing its vertex information. The first step is to define the mesh that is going to be accessed. Every mesh

can consist of a number of sub-meshes. It has to have at least one, which happens if all vertices of the mesh are connected with each other, but can have multiple. This is the case if for example a face with eyes is saved as a single mesh file. The face is then a sub mesh, and each of the eyes too, since no connections exist between each of these. The next step is to define the sub-mesh we want to access.

From this sub-mesh the vertex data is accessible, but it still contains some unwanted information. It provides the position of the vertices, but holds also information about their diffuse, specular, blending and weight values among others. To get rid of these, we define a filter that only retrieves the position of each vertex, which consists of three floating point values. These can now be retrieved and used to update the position of the mesh vertices.

Important is, that the simulation vertices are assigned a vertex id, which allows to easily access their mesh vertex partner later on. This is all the information the simulation vertices need. A vertex id and position values. The skin simulation is changing the simulation vertices positions relevant to the forces of the constraints and then the simulation vertices are updating the mesh vertices. After creating these 'virtual' vertices, the skin simulation generates the length and position constraints.

4.3.2 Position Constraints

The position constraints are quite easy to calculate and implement after generating the simulation vertices, because they deliver the initialization vertex position V_{init} already. All that is left to do is create a force F_p that tries to pull the vertex back as soon as it leaves it original position. For every simulation vertex a position constraint is created and the original position of the vertex saved in it. The next step is then during run-time of the simulation, to calculate the distance between the vertex position and its original one. This is done by comparing the length values of the distance at initialization $Dist_{init}$ and at the current moment $Dist_{curr}$. The initialization length is always zero, so it is only necessary to measure the current distance between the vertex and its original position. Depending on this distance is the magnitude M of the resulting force.

$$M = Dist_{curr} - 0$$

The next step is to calculate the direction D of the force, which can be done by

subtracting the vertex initial position V_{init} from the vertex current one V_{curr} and by normalizing the resulting vector.

$$D = n(V_{curr} - V_{init})$$

This forces tries to keep the mesh in its original state, and redoes all the deformations as soon as the muscle force vanishes. It is defined by

$$F_p = M \cdot D$$

This force has to be added to the influenced vertex position, resulting into the final formula of

$$V_{pos} = V_{pos} + F_p$$

4.3.3 Length Constraints

To implement the length constraints, it is important to know which vertices are connected with each other. This information has to be read from the mesh, and it is necessary to access the edge data. The first step is to define the mesh that hat to be accessed. Ogre saves the edges for all the sub-meshes into a simple list. Each edge in the list saves the id of the participating vertices in its vertex index, which is an integer array with the size of two.

By iterating through this edge list and extracting the mesh vertices, it is possible to match their id with the ones of the simulation vertices. For the length constraints however, it is not only necessary to know which vertices together build an edge, but also the distance between them. Upon initialization these distances are calculated by computing the length of the vector that results from subtracting the position of one of the vertices from the other.

To calculate a force F_l that keeps the vertices at the same distance from each other is the next step. The magnitude of this force is depending on the difference between the original length $Dist_{init}$ of the edge and the current one $Dist_{curr}$. If the skin is compressed, the magnitude M of the force contains a negative value, resulting in a compelling force.

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

The direction D of the force can be calculated by subtracting the positions of the simulation vertices from each other and normalizing the result.

$$D = n(V_{1\,pos} - V_{2\,pos})$$

These constraints produce the desired forces that help to keep the vertices at the same distance from each other, and built wrinkles upon compression.

$$F_l = M \cdot D \cdot 0.5$$

Both vertices of the constraint are moved in opposite directions relevant to the resulting force.

$$V_{1\,pos} = V_{1\,pos} + F_l$$
$$V_{2\,pos} = V_{2\,pos} - F_l$$

4.3.4 Muscle Constraints

The muscle constraints, which simulate the influence of muscle contraction on the skin surface, are the most important ones for the creation of facial expressions. They cannot be generated from the mesh model itself, but have to be implemented by the muscle simulation. These constraints need a target position V_{tar} for each vertex, which is assigned by the muscle during run-time. The magnitude M of the created force depends on the distance between the current position of the vertex V_{curr} and its target position.

$$M = (V_{tar} - V_{curr})^2$$

The direction D of the resulting force F_m is calculated by subtracting the vertex position from the target position.

$$D = V_{tar} - V_{curr}$$

The final force for the muscle constraints allows the face to be animated by the muscle simulation and is calculated by

$$F_m = M \cdot D$$

For each vertex that is in the influence zone of the muscle the force is added to its position.

$$V_{pos} = V_{pos} + F_m$$

For this constraint it is important that, unlike position constraints, the target position is

changed and reassigned depending on the muscles contraction during run-time of the simulation.

4.3.5 Updating vertex normals

To make the wrinkles visible on the face mesh, it is unfortunately not enough to move the vertices to the right positions. The normals for the vertices have to be recalculated too. These normals influence how much light a vertex reflects, and are therefore responsible, how dark areas on the face are. Since wrinkles are only small surface deformations, they can only be perceivable from the side as small bulges, without updating the vertex normals. To update a vertex normal, it is necessary to recalculate it, by calculating and summing up all the face normals the vertex is part of and computing the average of it.

The first step is to find out which vertex is part of what triangle. Ogre provides a triangle list for that purpose, which contains all the necessary information. After getting these vertex ids, it is necessary match them to our simulation vertices and get the current positions from them.

For each triangle the face can be calculated by normalizing the cross product of two of its edge direction vectors. These direction vectors are computed by subtracting the vertex positions from each other. The face normals are stored into an individual list for each vertex that is a part of the triangle. After iterating through the whole mesh and calculating all the face normals, the average normal for each vertex can be computed by adding all its normals in a list and dividing the result through their number. This results into a new vertex normal for each vertex.

To update the vertex normals in the mesh, it is necessary to go through a similar process like the one described in section 4.3.1 for updating the position of the mesh vertices. A mesh and a sub-mesh has to be specified, its vertex data accessed and a filter that returns us only the normal information implemented. This information consists of three floating points values. The next step is to overwrite these values with the new calculated vertex normal. After implementing the updated vertex normals, the wrinkles are clearly visible on the mesh.

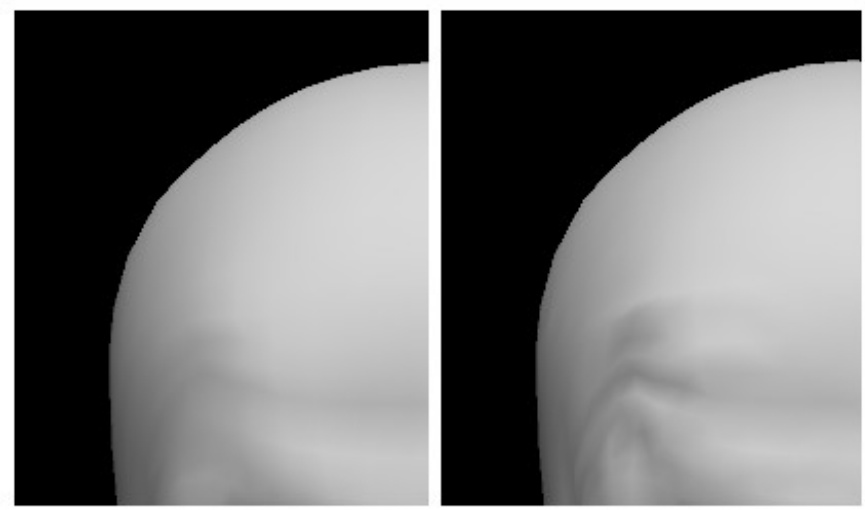

Figure 23: Forehead without (left) and with (right) the updated vertex normals

4.3.6 Skin simulation overview

To initialize the skin simulation it is necessary to define two parameters. The first is the mesh model, the second parameter is the number of iterations the simulation should work through all the constraints at each time step. After defining these parameters, the skin simulation starts by creating a simulation vertex for each 'real' mesh vertex in the model. All the simulation vertices are then stored in an array to make them easily accessible.The next step is the generation of the length constraints and the position constraints, which are stored in a list. For separation purpose there is a different list for each constraint type. Another list is created for the muscle constraints which is later filled up by the muscle simulation.

During run-time, the following steps are done for each frame in which a muscle contraction changes.

- Iterate through constraints
  - Position Constraints
  - Muscle Constraints
  - Length Constraints
- Apply changes to simulation vertices
- Update mesh vertex positions

- Calculate vertex normals
- Update mesh vertex normals

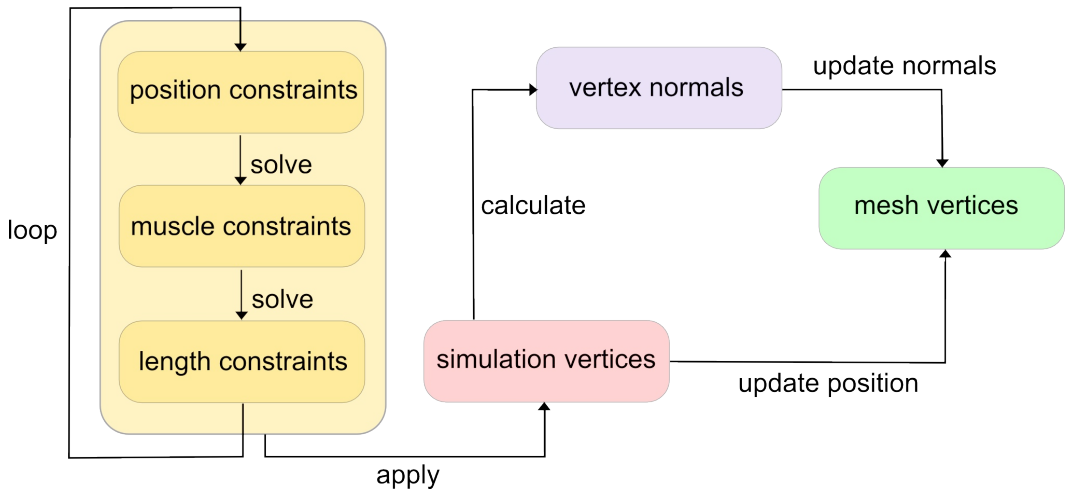

Figure 24: skin simulation process at run-time

It is to mention that the length and the position constraints have an separate value that determines their iteration number. This was implemented to allow the generation of smoother skin and more believable wrinkles. If the length and muscle constraints are computed evenly, the length constraints do not have the chance to calculate an optimal position for each vertex. This results in an unrealistic skin and wrinkles. On the other hand, the more often the length constraints are computed, the more strength need the muscle constraints to allow a satisfying deformation. More on this topic will be presented in the section 7. Test and Performance.

4.4 Implementation of the muscle simulations

The implemented muscle simulation is based on Waters muscle model and was slightly modified to allow the easier realization of problematic areas, specifically the jaw rotation and the opening of the eyelids. Three different muscle types were therefore implemented, the linear muscle, the sphincter muscle and the rotation muscle.

4.4.1 Linear muscle

The length of the linear muscle is defined by a start point, which represents the origin of the muscle, and an endpoint, the point where the muscle is attached to the skin surface. Its broad is defined by an angle alpha, and gives the muscle the form of a sphere segment. Additionally to that, the falloff start has to be defined, as a distance in relation to the origin of the muscle. These parameters take care of the visual appearance of the muscle.

Still missing though, is an identification parameter for the muscle, which was realized by implementing a name for each. This name is chosen depending on the relevant Action Unit of the Facial Action Coding System. Each muscle has to know the skin simulation which it affects too, so that it is able to write his muscle constraints into the skin simulations constraint list. And finally for initialization purpose, the mesh model has to be specified.

To assign the correct vertices to the muscle, meaning all the vertices that the muscle contains, it is necessary have to loop through all the vertex positions and use some simple comparisons. There are two statements, that have to be valid. The distance from the start point P_{start} of the muscle to the vertex position V_{pos} has to be smaller then the distance from the muscles start point to its end point P_{end}. And the angle between $\overline{P_{start}P_{end}}$ and $\overline{P_{start}V_{pos}}$ has to be smaller than the predefined muscle angle α. If this is the case for the vertex position, then a new muscle constraint is added to the skin simulations constraint list, containing the muscle name, the simulation vertex id and the initialization contraction position, which is the vertexes original position.

After all the correct muscle constraints were added, a visual form is created for the muscle. This is done by using Ogres ability to create manual objects. Manual objects are user programmed models, which can be compared to the creation of models in DirectX or OpenGL. As for these, their use should be limited to rather simple objects, since every vertex and its connections have to be individually defined. The use of manual objects makes the task a little bit easier as it is in DirectX by providing some predefined parameters.

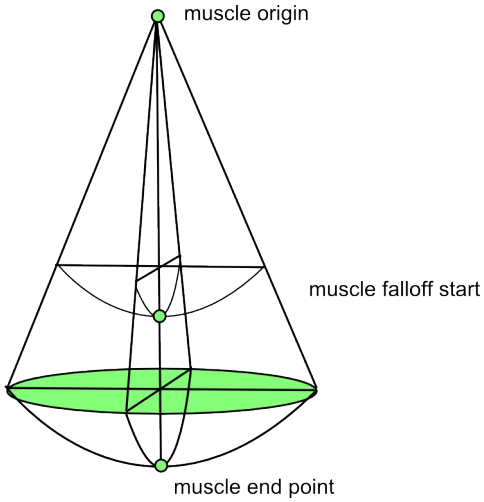

Figure 25: Linear muscle visualization

The visual appearance of the muscle is in the form of a wire model. To approximate the shape of the sphere segment, a flat circle is created with connections to the origin of the muscle. However, this form still represents a cone and to turn it into a sphere segment two curves are placed under the cone, normal to each other. Another two curves are used to show the distance for the begin of the falloff of the muscle. During the contraction of the muscle, only one parameter is needed, which simulates the strength with which the muscle contracts. As described in the theoretical section 3.2, the contracted vertex position is calculated by

$$p' = f(p \cdot K \cdot A \cdot R)$$

and all the needed values can be calculated as described in Waters paper [9]. For each influenced vertex, the relevant muscle constraint is extracted from the muscle constraint list of the skin simulation by using the muscles name as an identifier, and the contracted vertex position is then added as the constraints target position. When the skin simulations iterates through the constraints again, the muscle constraints develop a force which pulls the vertices to their target positions.

4.4.2 Sphincter muscle

This muscle, which is responsible for the circular compression around the eyes and the mouth, is defined by a center point, which represents the bone attachment of the muscle, and three radii, which are needed to simulate its ellipsoidal form. Three additional inner

radii are needed for an inner ellipsoid, which represent the falloff start of this muscle. Also a scale factor was added, since the sphincter muscle only contracts uniformly towards its center. This scale factor helps to make some modifications to the muscle behavior.

For the muscles around the eye, the horizontal contraction does not have as much effect as the vertical one, which does deform the region around the eye unrealistically otherwise. The region around the mouth on the other hand, has to contract stronger horizontally, or the circular mouth shape for the puckering of the lips can not be achieved. Other than these parameters, a muscle name for identification, the mesh and the relevant skin simulation have to be defined.

During initialization, all the vertices which belong to the muscle are extracted from the mesh. This is done by checking if the vertex position V_{pos} is inside the ellipsoid or not. The formula uses therefore each axes individually and checks if the distance from the center point P_{center} to the vertex position, divided by the muscle radius R and added up, is smaller than 1 or not.

$$\left(\frac{(\overline{P_{center}V_{pos}}x)}{Rx}+\frac{(\overline{P_{center}V_{pos}}y)}{Ry}+\frac{(\overline{P_{center}V_{pos}}z)}{Rz}\right)\leqslant 1$$

If the resulting statement is true, then the vertex is influenced by the muscle and a muscle constraint is added to the skin simulations constraint list.

For the visualization part a wire model is used and an ellipsoid is approximated by three ellipses, one for each axes. This is done for the outer ellipsoid, which simulates the boundaries of the muscle as well as for the inner ellipsoid, which represents the falloff distance.

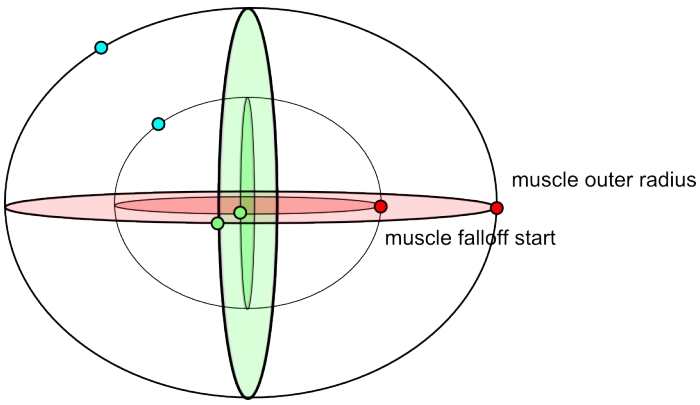

Figure 26: Sphincter muscle visualization

The contraction works similar to the one for the linear muscles. Only one parameter, a value for the contraction strength, is needed and the muscle then iterates through all the muscle constraints of the skin simulation, looking for the ones that have its name as an identification. It then updates their contraction position and the vertices are moved due to the resulting constraint force.
Another small modification that was implemented apart from the scale factor was to lessen the contracting force inside the inner ellipsoid, before the falloff starts. This was necessary since otherwise these areas did get too compressed, creating unrealistic wrinkles.

4.4.3 Rotation muscles

As mentioned in the theoretical part of this thesis, the rotational muscles are split into two subtypes, which main difference is their form. The lid muscle, that is implemented to simulate the opening of the eyelid, has the form of an ellipsoid, while the jaw muscle, which simulates the jaw rotation due to muscle contraction or in other words the opening of the mouth, has the form of a cuboid.

4.4.3.1 Lid muscle

This muscle is defined by a center point and three radii for the ellipsoidal form. Since this is similar to the sphincter muscle, the influenced vertices are also calculated by

$$\left(\frac{(\overline{P_{center}V_{pos}}x)}{Rx}+\frac{(\overline{P_{center}V_{pos}}y)}{Ry}+\frac{(\overline{P_{center}V_{pos}}z)}{Rz}\right)\leqslant 1$$

Other parameters are not needed, since the calculation of the contracted vertex positions is not based on Waters formula anymore. When this muscle contracts, the position of the joint P_{joint} is derived from the center point P_{center} minus the radius for the z axis R_z, which puts the joint in the back of the ellipsoid.

$$P_{joint}=P_{center}-R_z$$

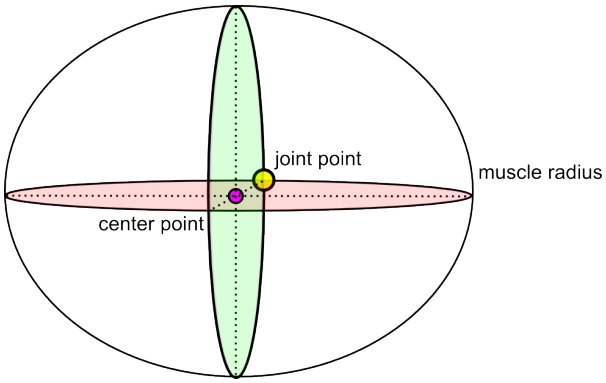

Figure 27: Lid muscle visualization

The value of the contraction strength has to be between 0 and 90 for this muscle, since a further rotation would look unrealistically, and is also not visible from the outside.
During the contraction the rotation axis of the joint, which is the x-axis in this case, is calculated and stored as an rotation matrix M_{rot}, allowing to calculate the contracted vertex positions V_{pos}' as

$$V_{pos}' = P_{joint} + \overline{P_{joint} V_{pos}} \cdot M_{rot}$$

This newly calculated position is then stored into the relevant muscle constraints as the target position.

4.4.3.2 Jaw Muscle

The parameters needed to create the jaw muscle are similar to the ones for the lid muscle. With a start point, and three length values a cuboid is formed. Though this form is simple, it approximates the jaw region sufficiently. For the calculation of the joint point, similar principles are applied too. From the start point P_{start} half the length value of the z-axis L_z is subtracted resulting into the joint point P_{joint}

$$P_{joint} = P_{start} - \frac{L_z}{2}$$

The contraction is calculated the same way as the one for the lid muscle is, with a small modification because upon opening the mouth, the lower lip tended to stay too straight. Preferable is that the middle part of the lip opens slightly further, resulting into a light

curve. Therefore for each vertex its relative position to the joint P_{joint} is calculated, and depending on the similarities on the x and y-axis, a slightly additional force calculated by

$$V_{posy}' = V_{posy} + \sqrt{(L_x - |V_{posx} - P_{jointx}|) \cdot (L_y - |V_{posy} - P_{jointy}|)} \cdot n$$

where L stands for the predefined length values of the muscle and n stands for a necessary strength multiplier.

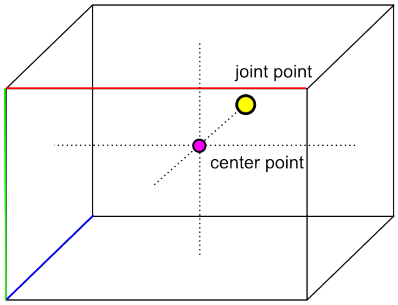

Figure 28: Jaw muscle visualization

4.5 Implemented muscles after FACS

All the muscles in the implementation are based upon the descriptions in the Facial Action Coding System, and were chosen due to their importance for the creation of different facial expressions. The size and the place for the muscles are based upon the drawings in the manual, and were then manually repositioned until the results were satisfying.Since the wrinkles that a face creates during different facial expressions depend upon age, gender and topology of the face, the photographs in the copy were only used as a loose guideline for this purpose.

Nearly all of the implemented action units, with the exception of action unit 17, 18 and 26 are created by muscles which are found on each part of the face. In total, 13 action units were used, which allow to create a large variety of expressions already.

4.5.1 AU1, inner brow raiser

This action is created by the center part of a large muscle sitting in the scalp and forehead

of the face. It raises the inner part of the eyebrows up, resulting into horizontal wrinkles on the forehead. These wrinkles do not appear across the whole forehead, but only in the middle part of it, and can be slightly raised at the center.

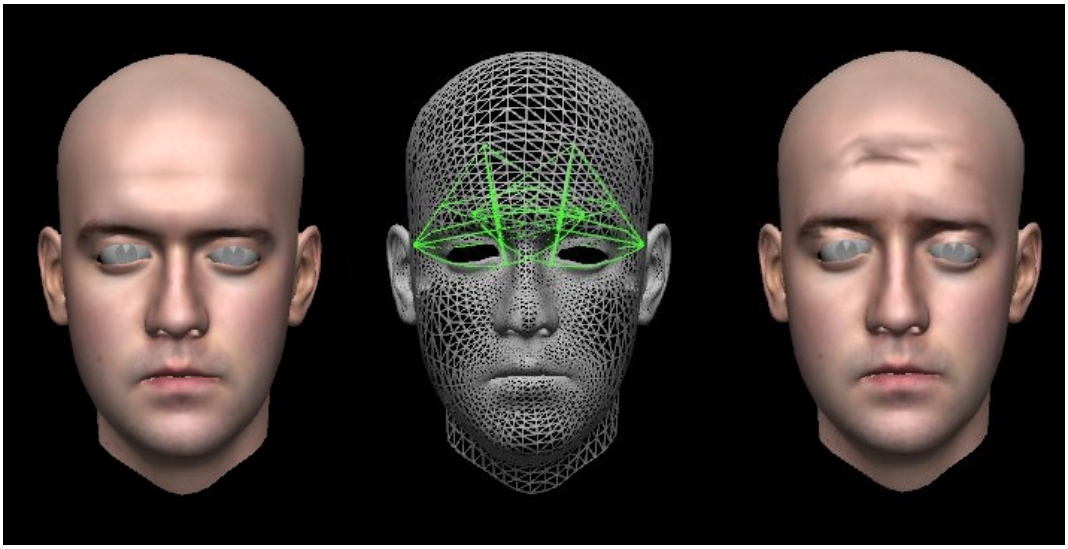

Figure 29: neutral expression (left), implemented muscles (middle), AU1 (right)

Though this action unit was simulated by two mirrored muscles, their contraction is always combined together and can not be individually changed. A previous attempt to simulate this action unit with a single muscle created wrinkles which were too arched, therefore the second muscle was added for better results.

4.5.2 AU2, outer brow raiser

Responsible for this action unit is the lateral part of the same muscle that also creates AU1. Upon its contraction the outer parts of the eyebrows are raised, resulting into a arched shape of each, and the relevant part of the eye cover fold is stretched. This causes small horizontal wrinkles to appear on the sides of the forehead, above the lateral part of the eyebrow. This action unit is mirrored, and can be produced on either side of the face separately. For each side of the face a single muscle was implemented to approximate the desired behavior.

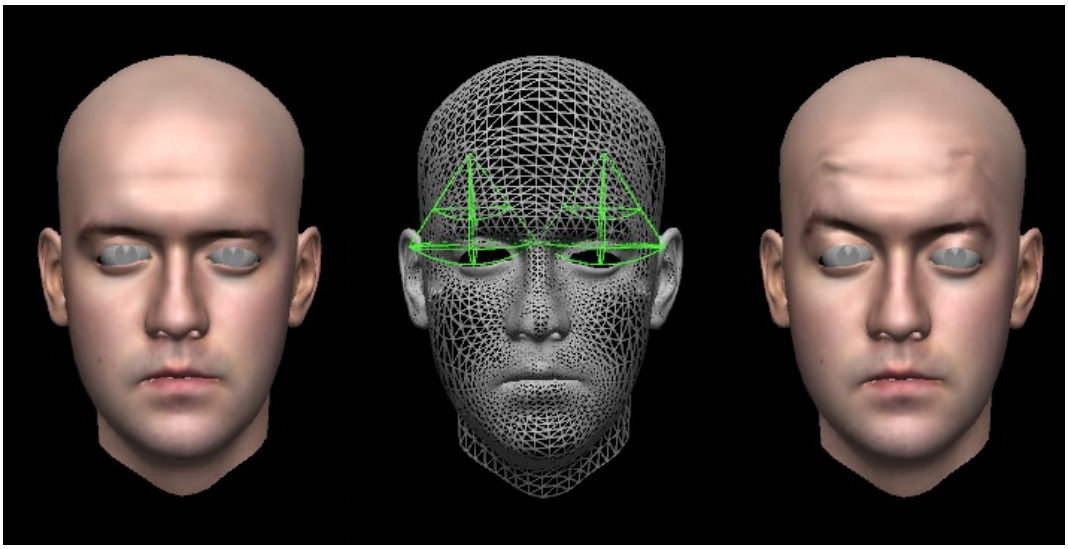

Figure 30: neutral expression (left), implemented muscles (middle), AU2 (right)

Since the acquired head model results from a scan, the sides of the face are not mirrored and have small differences in their topology, which leads to the production of different wrinkles on each side. However, this is considered as realistic since it is very likely that each side of a face produces slightly different wrinkles.

4.5.3 AU4, brow lowerer

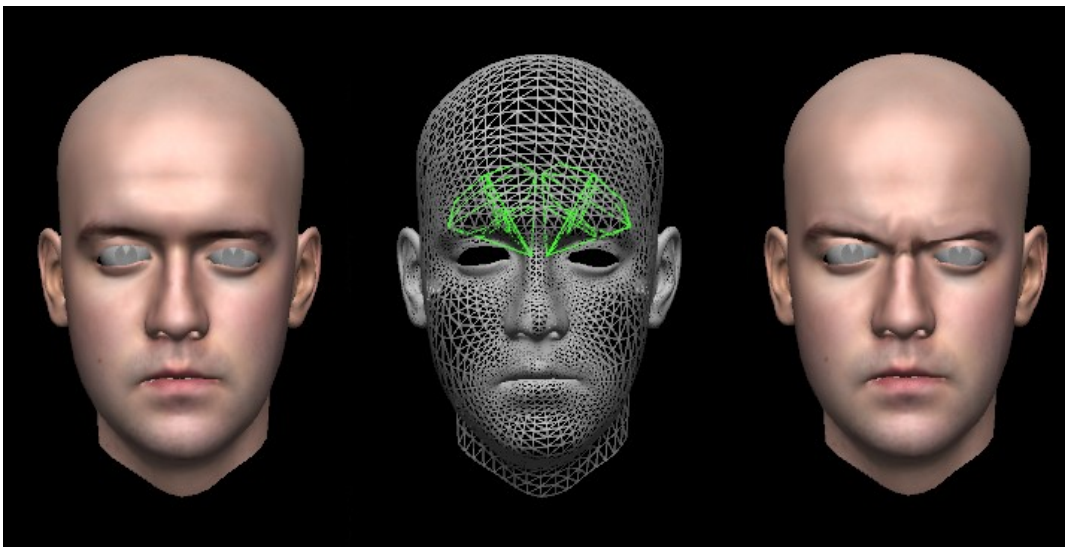

Figure 31: neutral expression (left), implemented muscles (middle), AU4 (right)

Three different muscle strands are involved to produce this action unit,. The most powerful of these origins a little to the side of the root of the nose and spreads upwards to a point above the eyebrows on the lower part of the forehead. It is responsible for pulling the eyebrows together and lowering them. The second strand origins at the center of the root of the nose and spreads a little more vertically to the center of the forehead. Even more vertically runs the last strand, originating next to the inner part of the eyebrows and running to their corners.

These three strands nearly always act together, and were therefore chosen to be represented as a single action unit. Upon their contraction the inner and sometimes also the center part of the eyebrows is lowered and the eyebrows are pulled towards each other. This produces vertical wrinkles between the eyebrows which may also appear at a 45 degree angle.

Although three different muscle strands produce this action unit on each side of the face, in the simulation they are approximated through a single linear muscle, which already produces decent results. For the whole action unit, two muscles were needed which also always work together and can not be used separately.

4.5.4 AU5, upper lid raiser

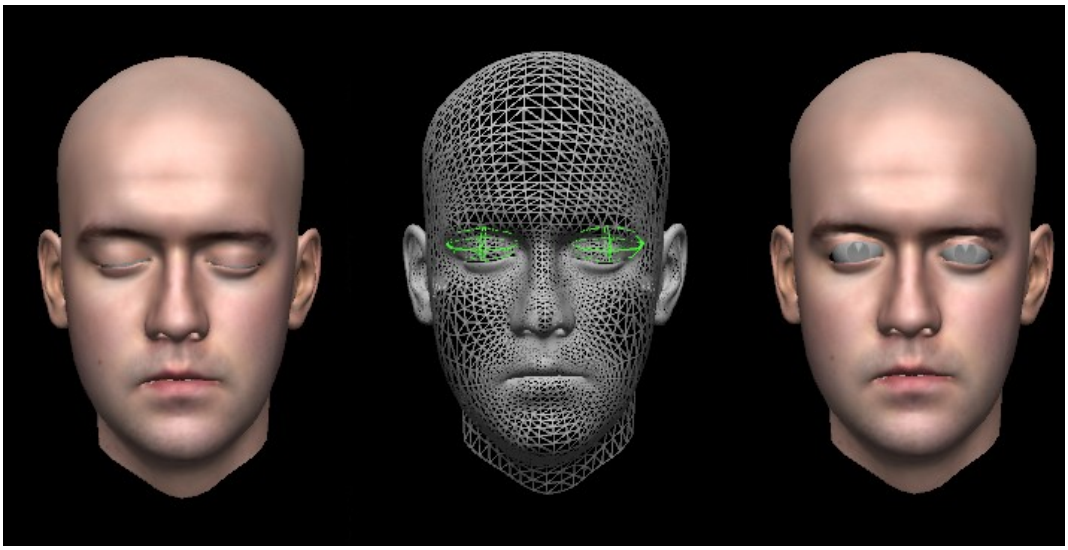

Figure 32: neutral expression (left), implemented muscles (middle), AU5 (right)

This action unit represents the pulling back of the upper eye lid into the eye socket. The

muscles responsible for this action were not visually presented in the Facial Action Coding System, but their position and size was guessed from their purpose. When the relevant muscle relaxes, the eye lid falls over the eyeball and closes the eye. During the normal opening of the eye this muscle is only slightly contracted. Upon strong contraction the upper eyelid is pulled further back, often until there is nothing left visible.

For the approximation of this action unit the lid muscle, a subtype of the rotation muscle, was created and implemented on each side of the face. Normally these two muscles work together, for the purpose of blinking or closing the eyes, or to stare with wide open eyelids, but sometimes they have to be used individually. Therefore this action unit can be used for either side of the face separately to produce animations like a wink. Especially important is this action unit for letting the eyelids follow the movement of the eyes of a character, which was integrated into the eye simulation of the implementation.

4.5.5 AU6, cheek raiser and lid compressor

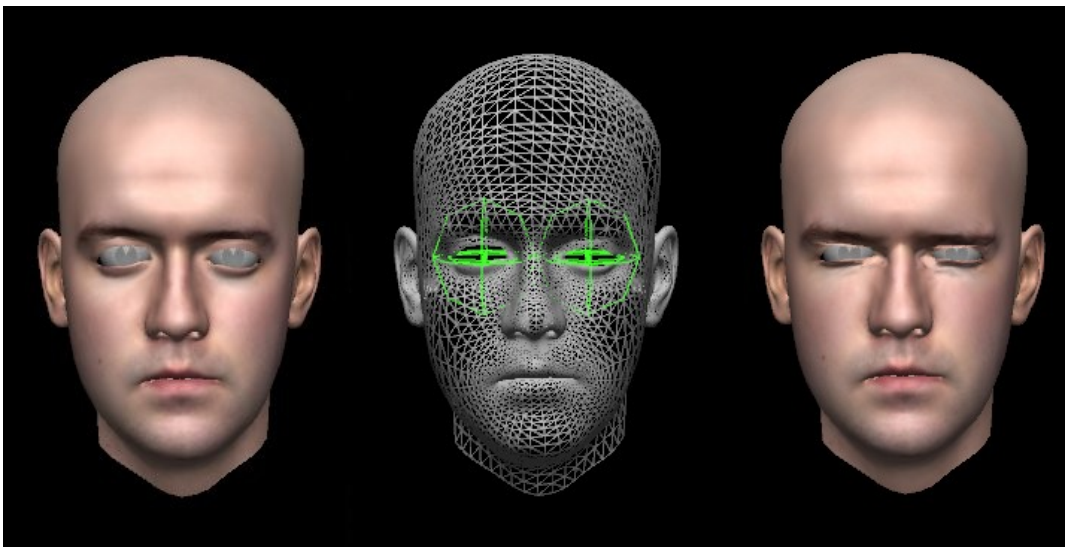

Figure 33: neutral expression (left), implemented muscles (middle), AU6 (right)

The muscle for this action unit is one of the implemented sphincter muscles, and runs circular around the eye socket. Its circumference reaches from the eyebrows to under the lower eye furrow. Upon contraction this muscle pulls the skin surrounding the eye towards it. In detail, it lowers the eye brows a little bit, pulls the skin from the temple towards the

eye and lifts the cheek upwards. This process can, depending on the contraction strength, narrow the eye aperture and push the eye cover fold down.
During this action unit wrinkles can be generated below the eye and fine crow lines appear that origin at the outer part of the eye. It may also deepen the lower eye lid furrow. This action unit is approximated by a single sphincter muscle for each side of the face, that simulates the circular muscle around the eye sufficiently. Since this facial action can occur on each side individually, the action unit was divided and can be controlled separately.
Unfortunately the resulting wrinkles are only visible below and at the inner part of the eye, and nearly no crowfeet or wrinkles are created on the outer side. The reason for this may be the mesh resolution since the wrinkles around the eyes have to be very fine.

4.5.6 AU9, nose wrinkler

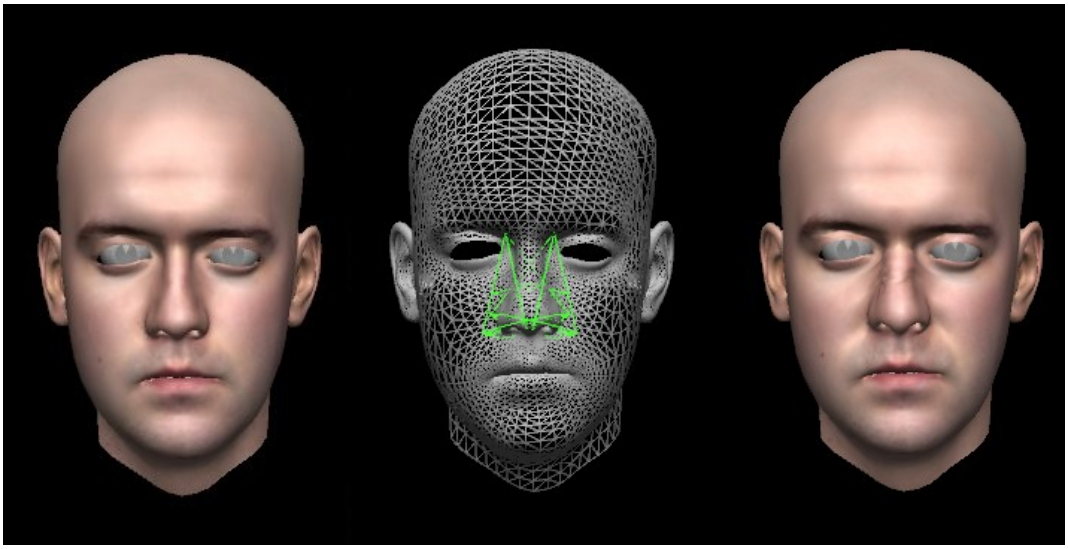

Figure 34: neutral expression (left), implemented muscles (middle), AU9 (right)

This action unit simulates a muscle that origins at the root of the nose and reaches downwards along its side to a point below the nostril wings. During the contraction of this muscle the skin at the side of the nose is pulled towards the origin of it, which results into a skin compression creating wrinkles on the upper part of each side and the root of the nose. This process can also widen and raise the nostril wings. In the process it may lower the brows a little bit and raises the upper lips slightly. These deformations were unfortunately

not implemented with this action unit, and the focus lied on the creation of wrinkles upon contraction.

However, the lowering of the eyebrows can be easily added by combining this action unit with AU4. Although this action is the result of a single muscle, the simulation uses two linear muscles, one on each side of the nose, to create the desired wrinkly result. These two muscle can only be contracted as a single unit, and produce for the same reason as AU2, different wrinkles on each side of the face. This action unit may not seem important, but it helps to get more realistic results for various facial expressions, like anger or confusion.

4.5.7 AU10, upper lip raiser

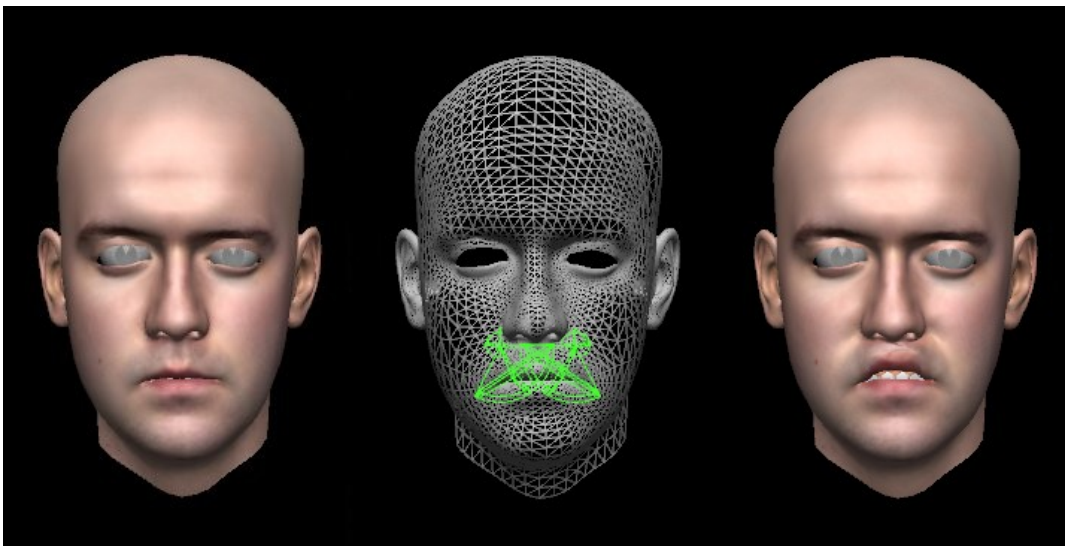

Figure 35: neutral expression (left), implemented muscles (middle), AU10 (right)

Responsible for this action unit are two muscles, one on each side of the face, that origin relatively high at a point over the nose and attach themselves at the skin next to the nasolabial furrow. Upon contraction they pull on their skin attachment and raise it up and to the side towards the cheeks, resulting in the lifting of the upper lip. If the contraction is strong, it also raises and widens the nostril wings a bit and deepens the nasolabial furrow beneath them.

Since this action unit is based on a single muscle on each side of the face, the realization in the implementation was done with a single muscle too. However there were some

complications with the muscles that did not allow to implement the finer details. The raising of the upper lip with two muscles could be done, but to pull its upper part back at the same time, to create the nasolabial furrow, proved to be not possible. To do so, the muscle would either distort parts of the nose too much, or pull the upper part of the lip too much inwards, which both lead to unrealistic deformations.
The action unit was therefore divided into two tasks, raising the lips and creating the furrow. Each of these tasks has a muscle assigned to it, which results in an overall of four muscles for this action unit. Since the actions created with these action unit are always mirrored, all muscles contract at the same time and are supposed to only be used as if it was a single unit.

4.5.8 AU12, lip corner puller

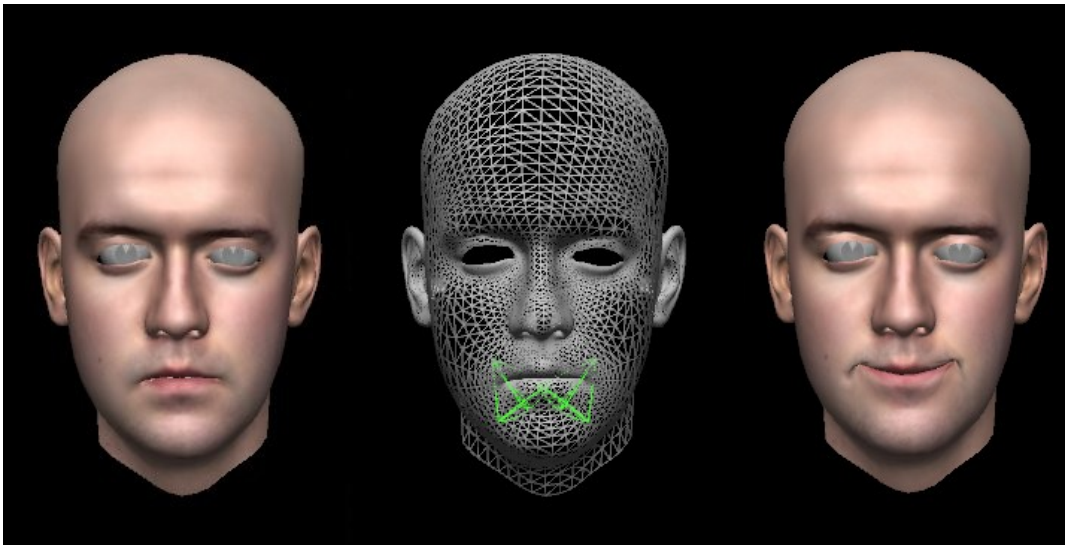

Figure 36: neutral expression (left), implemented muscles (middle), AU12 (right)

The muscle for this action unit origins in the lower part of the cheek bones and its other end is attached to the lip corners. When this muscle contracts it pulls the corner of the lips up and to the side, resulting in a smiling mouth shape. This behavior was approximated with a single linear muscle on each side of the face. There is no connection between these muscles, allowing the contraction on only one side of the face for the raising of only one lip corner.

This action unit was implemented with the knowledge that AU20 was also going to be used, which is responsible for the stretching of the lips. Therefore AU12 should allow a good combination AU20. The focus of AU12 lies in pulling the lip corner upwards, and to produce a realistic looking smile, AU20 has always to influence the face at the same time. Without it, AU12 results into unrealistic looking furrows, which can be seen in Figure 36.

4.5.9 AU15, lip corner depressor

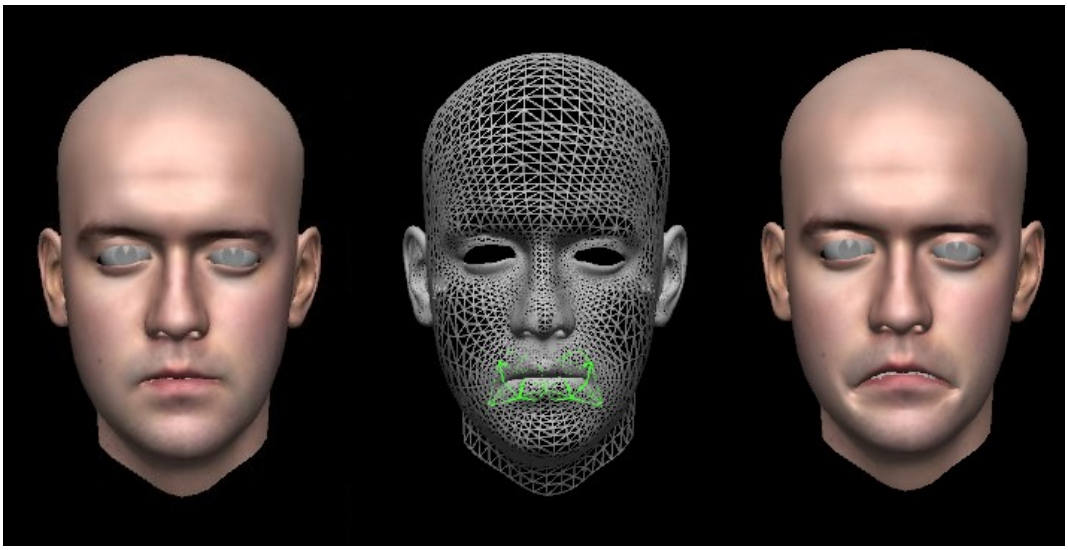

Figure 37: neutral expression (left), implemented muscles (middle), AU5 (right)

This action unit is the opposite of AU12 and pulls the corner of the lips down, which results in a mouth shape similar to an arch. The responsible muscle origins at a point in the region of the lower chin and runs along the side of the chin up to the corner of the lips. Upon its contraction, the lip corners are not only lowered but also stretched horizontally, which can result in some pouching or bagging of the skin near the lip corners.
The implementation of this action unit was uncomplicated, a single muscle on each side of the face was created to approximate the behavior. Its origin may not be as low as the real muscle, but it pulls the lip corners low enough to be sufficient for any facial expression. Also, when this muscle is contracted it creates a furrow at the lip corners, which looks pleasantly enough to give the impression of soft skin in this area.
The two muscles for this action were chosen to be controllable individually, since the

lowering of the lip corners is needed on only one side of the face to create asymmetric facial expressions.

4.5.10 AU17, chin raiser

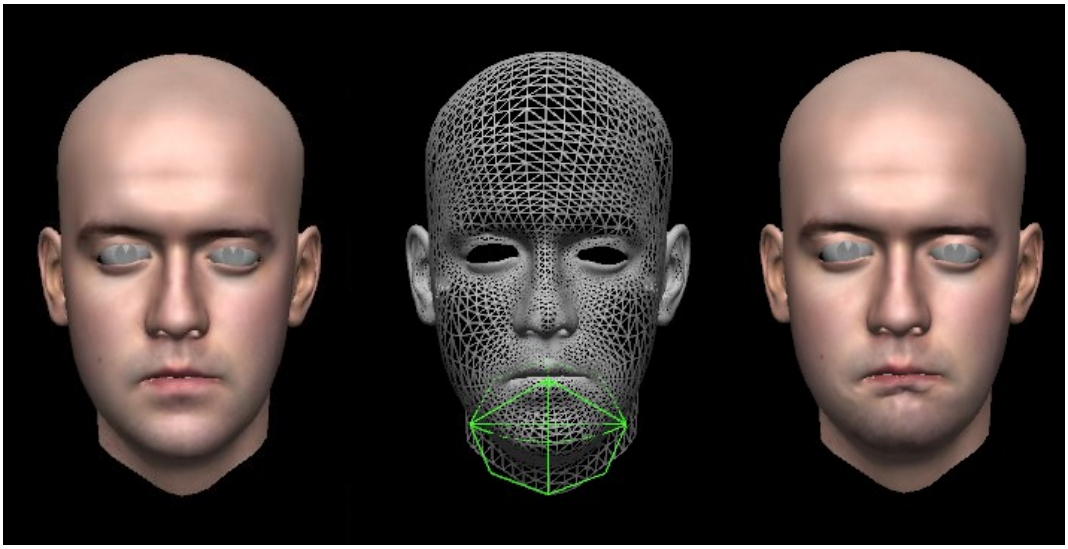

Figure 38: neutral expression (left), implemented muscles (middle), AU17 (right)

The muscle on which this action unit is based on is originating in the area just below the lips and reaches far down to the lower part of the chin. When the muscle contracts it pushes the skin of the chin upwards, which also results in a small raise of the center of the lower lip, creating a slight arch. During this process the skin of the chin surface gets compressed, which can result into a depression under the lower lip and small wrinkles on the chin boss.
This action unit is one of the few that only need a single muscle, because it happens at the center of the face and is therefore not a mirrored action. The implementation used therefore a single linear muscle for the approximation of this action unit.
During its contraction the depression under the lower lip becomes clearly visible, and a number of small deformations are visible on the skin boss, although none of them can really be called a wrinkle. A small wrinkle also appears at the corner of the lips, and the lower lip gets a little bit compressed. The raising of the chin may also not seem to be an important action unit, but again it provides small details that raise the realism of facial

expressions such as sadness.

4.5.11 AU18, lip pucker

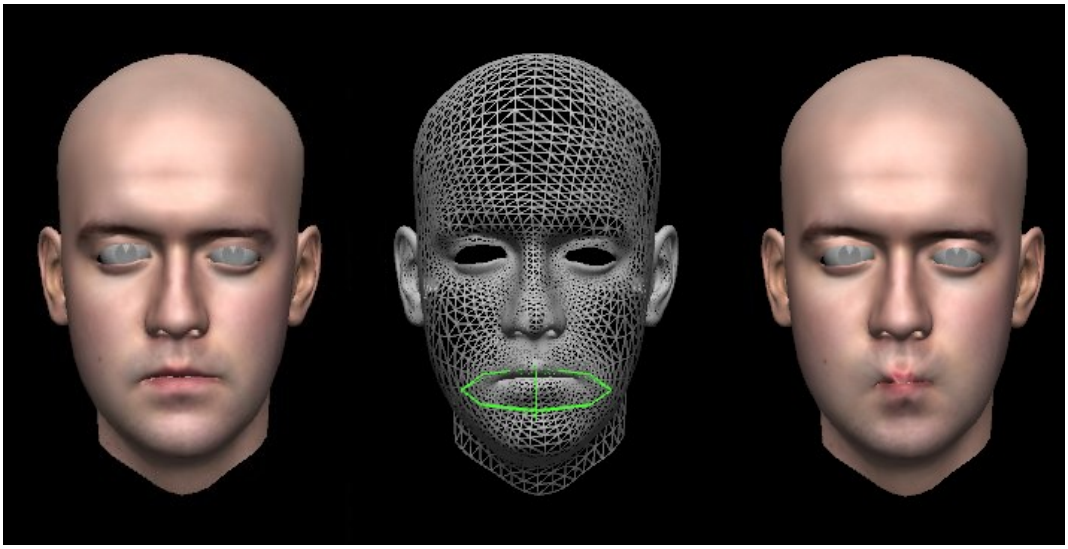

Figure 39: neutral expression (left), implemented muscles (middle), AU18 (right)

This is the second action unit that uses a sphincter muscle for its approximation. The real muscle it is based on, runs over the upper and below the lower lip and is not really circular, but its behavior can be sufficiently approximated assuming that it is. Upon its contraction it pulls the lip corner towards the center of the mouth, resulting in a round mouth shape, which may be open to form the letter 'o' or it presses the lips against each other. At the same time the center of the mouth is pulled forward. During this deformation, small wrinkles can occur on the skin of the upper lip, and also, though rarer, on the lower lip.
As already mentioned this action unit was implemented through a sphincter muscle, which allowed us to contract the surrounding skin of the mouth towards its center. Due to the relatively strong skin compression necessary to create a nearly round mouth shape, light wrinkles appear on the lower lip, especially to the side of it, were the compression force is the greatest. The lips, which are slightly pressed against each other during the contraction, show also very light wrinkles due to the compression.

4.5.12 AU20, lip stretcher

This action unit simulates the horizontal stretching of the lips. The underlying muscle origins far to the side and back of the lower face, near to the jaw and attaches its other end to the lip corners. It does not only pull the mouth corners to the side upon contraction, but also to the back in the direction of the ears. Depending on the face topology the lip corners are slightly raised or lowered during this process, but the main focus of this contraction lies on the horizontal stretching of them.The lips become stretched and a little bit flatten during the contraction and also the nostril wings can become elongated. Small wrinkles may appear on the side of the cheek and the skin is pulled into the lower part of the nasolabial furrow.

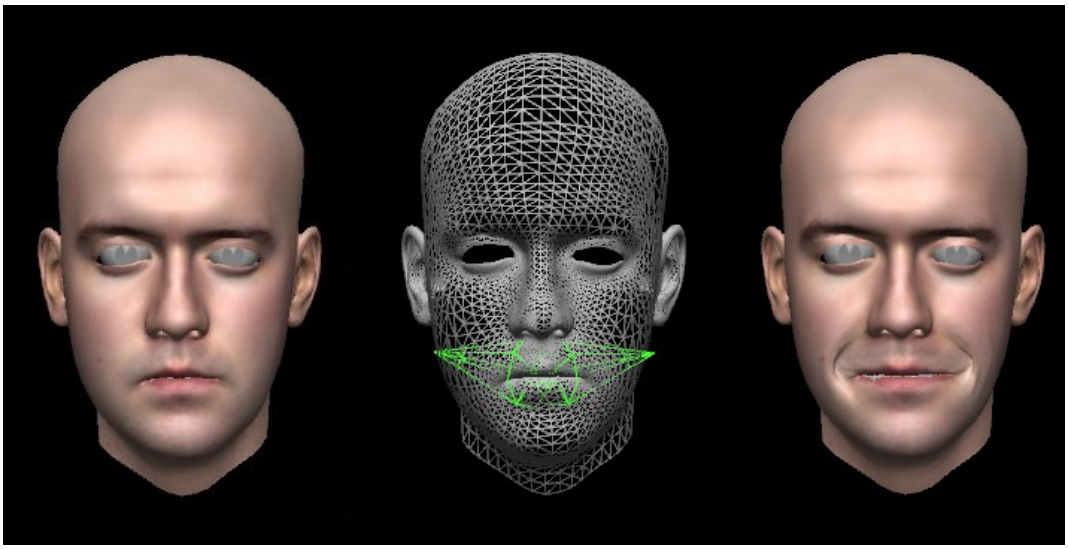

Figure 40: neutral expression (left), implemented muscles (middle), AU20 (right)

This action unit was implemented using a single linear muscle on each side of the face. Upon their contraction the big furrow near the lip corners becomes clearly visible and also a small wrinkle is visible to the side of it. As mentioned earlier, it is necessary to use this action unit in combination with AU12, to create a realistic looking smile.

4.5.13 AU26, jaw drop

The last implemented action unit is responsible for opening the mouth. It is a little bit

different than the other facial actions, since in this case, also a joint is involved. The responsible muscle sits far to the back and side of the head, and is attached to the jaw bone. He is responsible for keeping the jaw up and the mouth closed. When this muscle relaxes, the jaw drops and the lips part resulting in a mouth opening motion. This is in contrast to AU27, which pulls the jaw open, and results in a bigger mouth opening. But the implemented muscle can not be strictly assigned to either of these action units, and may be seen as a mix of these two.

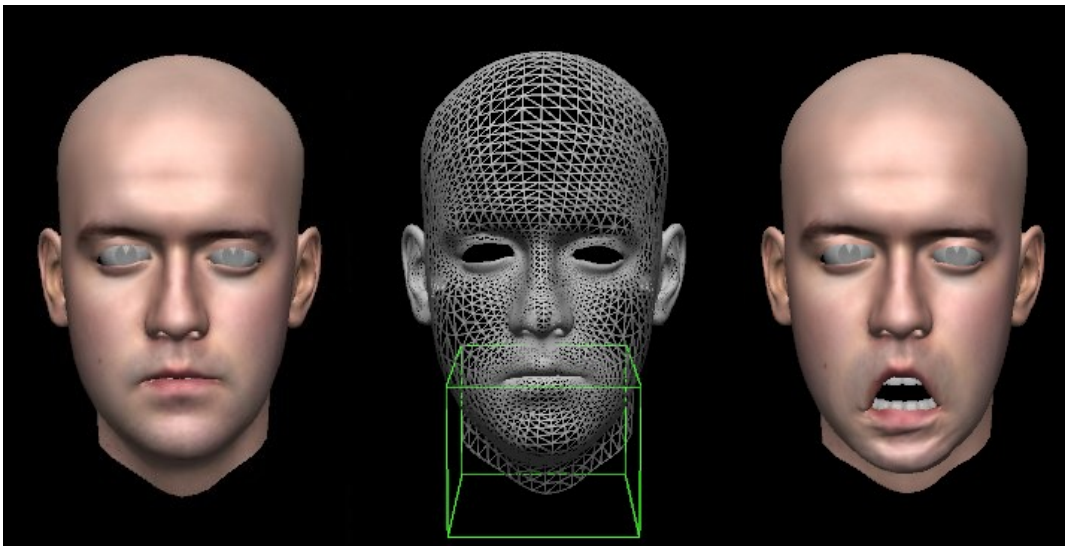

Figure 41: neutral expression (left), implemented muscles (middle), AU26 (right)

This behavior was approximated with the jaw muscle, which is again a subtype of the rotation muscle. Due to the only small space between the lips, the corners of the lips are pulled too far down during contraction, but this is easily correctable, by altering the mouth region a little bit and creating a bigger gap between the lips. During contraction, small wrinkles are created at the chin boss too, which was not mentioned in the FACS, but still gives a soft impression.

5. Application GUI

The GUI for the application is divided into a few sub menus, which all handle different tasks. All together there are five of them, which are responsible for

1. individual action units
2. predefined emotions
3. rendering options
4. outputs
5. real-time editor

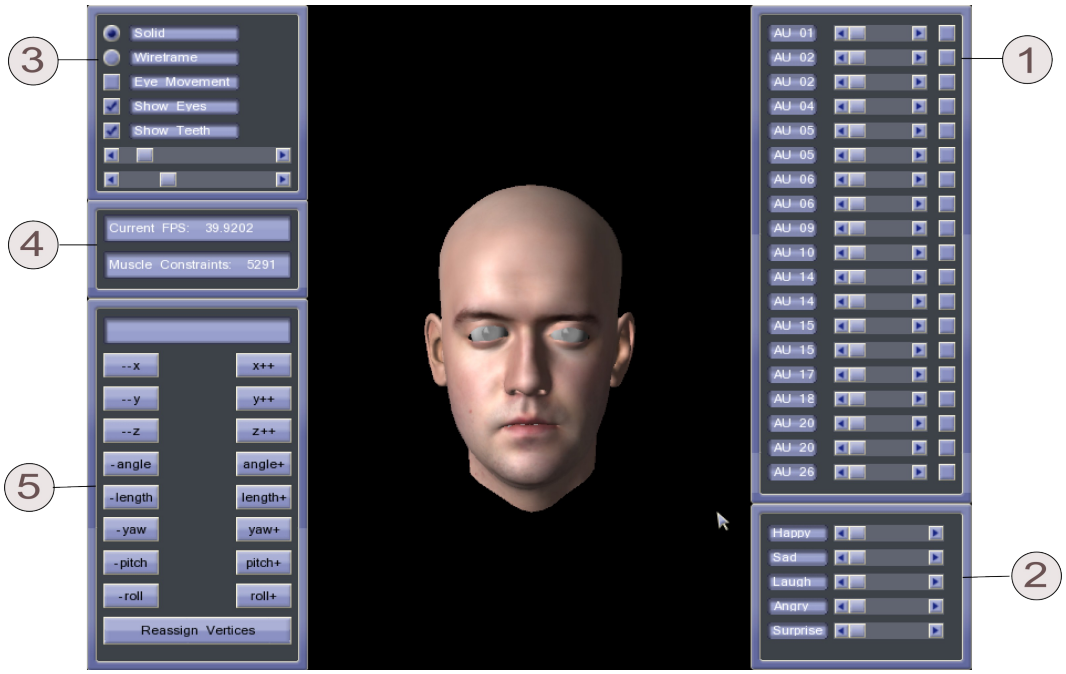

Figure 42: Application GUI overview

5.1 Individual action units

This sub menu provides a set of sliders that allows to manipulate the contraction strength of all the individual action units presented in the previous chapter. If the names exist double, then the first slider manipulates the left side of the face, the second one the right. Each row consists of a text field representing the action unit number, a slider for the

contraction strength, and a check box, that allows to make the muscle visible. This option is important to see what area the muscle covers and especially important for the editor, since only visible muscles can be selected. The sliders can also be used together to combine the different muscles and allow the creation of a variety of facial expressions.

5.2 Predefined emotions

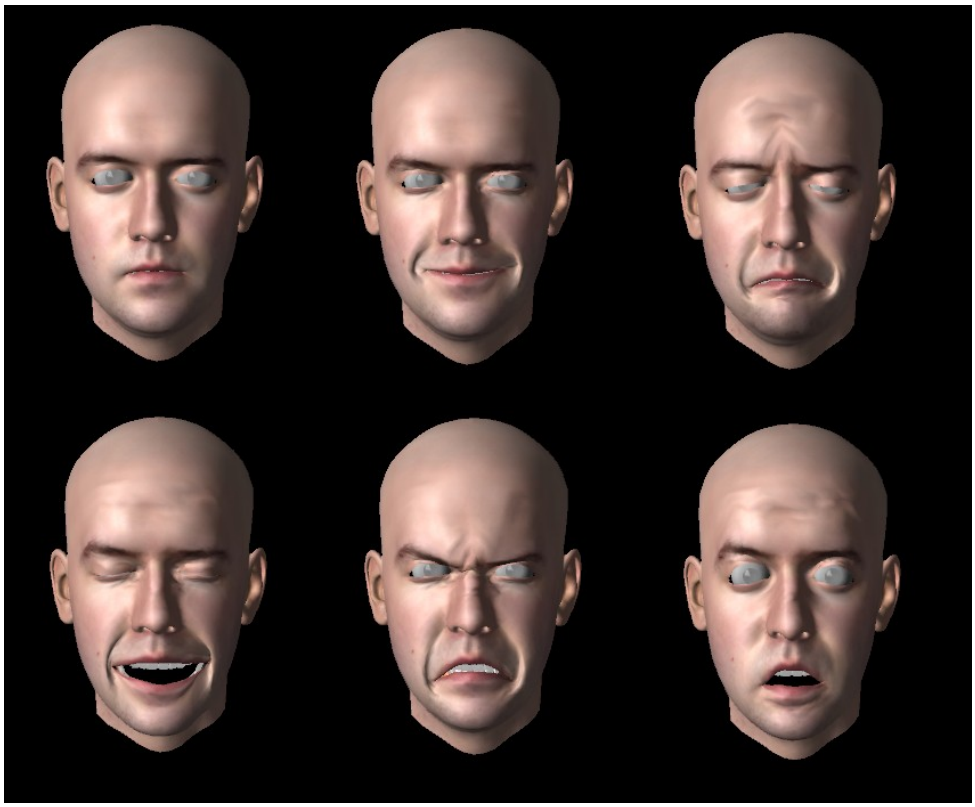

Figure 43: (in order) neutral, happy, sad, laughing, angry, surprised expression

For testing purpose and to show some of the results that can be achieved using this simulation, a few facial expressions were predefined. These facial expressions are created by defining a combination of the implemented individual action units. The predefined expressions are, happiness, sadness, laughing, anger and surprise. Although these expressions can be combined, to create for example an angry or a happy surprise expression, they were designed to be used alone and result in unrealistic deformations if the value of the sliders is too high.

Each of the defined expressions results out of the combination of the following action units

- 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

5.3 rendering options

Under this sub menu, some miscellaneous actions are combined, which all have in common that they alter the rendering of the model in some way.

5.3.1 Solid and wireframe mode

These radio buttons allow to switch the head rendering between the solid mode, which is the normal textured model view, and a wireframe mode, which renders the head as a wireframe model and allows to view its triangles. This can be useful to determine under which resolution, what kind of wrinkles are generated and allows to view the deformations clearer.

5.3.2 Show eyes and teeth

As mentioned earlier, to increase the realism of the simulation some additional models were created, which are the eyes and the teeth of the face. The visibility of these models can be switched on and of using these check boxes. This allows for example to hide the models when the head is viewed in the wireframe mode, since they would show through the mesh, and may act distracting.

5.3.3 Eye simulations

To make the head look a little bit more livelier, a simple eye simulation was implemented,

which can be switched on and off. For the simulation of a realistic eye behavior, two tasks have to be implemented. The first one is to let the eyes blink after a certain time period, which keeps the eyes moistened. After [24], blinking happens depending on the emotional state and the thoughts of the character, so it is not easy to assign a time after which a blinking has to occur. A random value is therefore chosen between 3 and 8 seconds, to determine when the next blinking happens.
The second task is to rotate the eye around in an area in which the iris would normally stay. This movement field was determined by simply testing the rotation range of the eyes that still looks naturally. The resulting values were between -30 and 30 for the yaw rotation, which is the left and right movement of the eye, and -20 to 25 for the pitch rotation, which is the up and down movement. For every new eye position, a value in the range of these is chosen,
To simulate the eye movement realistically, the eye should stay at its position for a while, and then move rather quick to the new position upon a change. So a random value with very little valid range is used to determine when the eye should change its position. A minor problem had to be solved before completion, which was that no blinking may happen during the movement of the eye. Therefore, the eye movement value is only updated if the eye currently does not blink.

5.4 Light

The application uses two kinds of light, an area lighting, which allows to define the hardness of the shadows and is responsible for the overall darkness of the model, and a directional light. This directional light can be controlled using these two sliders. Using the first slider will change the direction of the light on the horizontal layer, allowing to smoothly turn the light one time around the model. The second slider alters the vertical direction, and moves the light from above the model to its bottom. This can be useful to get a feeling about how different facial expressions look under changing light conditions, and also to highlight wrinkles, which may not be as visible as the can when the light comes directly from the front.

5.5 Outputs

In this section some values of interest are shown. Currently there are only two implemented output fields in this sub menu. The first one is the frame rate, which is a very important factor and especially interesting for measuring the performance of the application. The other one is the amount of muscle constraints currently used. This was implemented to get an overview of the implemented constraints, and how they change when the muscles are altered with the editor.

5.6 Real-time editor

This editor allows to arrange and redefine all the muscles in real-time so that they fit the face model better. It was implemented, since the coding of muscles is not as intuitive and the results not as easily viewable as they would be with an editor. Since the form of the muscles are quite different, the editor differs between the types of the muscles, and automatically provides the relevant interface items. The top text field however always stays the same independent from the muscle type, since it contains the name of the currently selected muscle.

5.6.1 Ray casting

To be able to select the muscle that has to be changed, Ogres ray casting abilities were used. It is necessary to define an empty ray, which is used whenever the left mouse button is pressed and to assign it to a camera. With the information from the camera the ray searches for any objects touched by the mouse pointer. The ray then calculates all the objects that are on his path and sorts them by distance. After this process it is possible to iterate through the returned results.

The ray only differs between two type of objects, world fragments and movables. World fragments are static objects and not important for the editor. The interesting ones are the movables, which are all sort of dynamic objects, like our muscles or the face model. The

first thing that has to be done is to get rid of all the mesh models the face consists of. This is accomplished by comparing the mesh names, the ray returns, with the ones we assigned to the face models. With this comparison it is possible to remove the face model, the eyes and teeth from the result list. After that procedure the selected muscle is the next on the list. One more filter has to be applied to identify the type of the selected muscle, since their different forms require the changes of different parameters.

5.6.2 Linear muscles editor

For the linear muscles, the editor provides options to change their position, their rotation and the scale of the muscles. Six buttons are responsible for changing the position of the muscle, which is done individually in each of the three axises. Since the changes can be either in the positive or negative direction, six buttons are the required minimum.

To scale the muscle, two parameters can be changed. The length and the angle of the muscle. By changing the length parameter, the overall size of the muscle is changed and it can be compared to a scale in all three axises. The angle on the other hand changes not only the broad of the muscle, but alters also the form quite a bit. As mentioned in the linear muscle section 4.4.1, the form of the muscle is a sphere segment, which can range from nearly a line shaped cone to a half sphere. For this reason the manual object of the muscle has to be redrawn after changes to the angle are made.

There is also the option, although for this muscle only, to rotate the muscle. The rotations are done individually too, allowing to make changes on each of the axis separately. Two buttons are provided to yaw, pitch or roll the muscle in the negative or positive direction.

Important is that, after all the changes for a muscle are done, the 'assign vertices' button has to be pressed. Only after pressing it, are the vertices for the muscle recalculated. This button was integrated to achieve a better performance, since calculating the vertices all the time while the editor is in use would be costly.

5.6.3 Sphincter muscle editor

For the sphincter muscles, the editor allows the manipulation of their position and size. The position changes work the same way as they do for the linear muscles. Six buttons are

provided which allow to move the muscle in the positive or negative direction of one of the axis. To change its size however, three parameters can be altered for this muscle. Each parameter represents the radius of the muscle in one of the axes, and they can be altered separately too. Six buttons are provided for this purpose. To finalize the changes on the sphincter muscle, the 'apply vertices' button has to be pressed again, so that the vertices are reassigned.

5.6.4 Rotation muscle editor

The editors interface for the rotational muscles was implemented generic enough to allow the use of the same buttons for the lid and jaw muscles. To move the muscle around on the axis six buttons were implemented. For the changes to the size of the muscles, the same principles as for the sphincter muscles are applied. Six buttons, where two represent each axis. This is possible, since the lid muscle, which form is an ellipsoid, and the jaw muscle, which is a cuboid, both need three parameter for their definition, which represent radii in one case and length values in the other. It is necessary for this muscle to press the 'assign vertices' button to allow the correct representations of the changes.

5.6.5 Saving and Loading

If no muscle is selected, the editor changes to the saving and loading interface. In this menu it is possible to load different head models, and previously saved muscle systems. All the head models that are going to be used, have to be in a specified directory. The editor reads all the files from this directory, and shows the ones with a .mesh ending in a drop-down box. After a file was chosen, the load model button has to be pressed. When a new head is loaded, the application automatically deletes all constraints in the skin simulation. They are recreated after a new muscle system is loaded. To load the muscles from a file, the file has to be in a predefined directory. A drop-down box allows to choose from these, and the load muscles button loads the muscles, and updates the skin simulation. After changes were made to the muscle system, it can be saved by typing a name and pressing the save muscles button. The muscle system is then written into a .txt file and stored in the predefined directory.

6. Tests and Performance

This simulation was developed and tested on a personal laptop, using an Intel Duo Core processor with 2.52GHz each, 4Gb Ram and a ATI Mobility Radeon HD 4600 Series graphics card. The two most important parameters for the visual performance of the simulation are the resolution of the used face model and the number of iterations. Some tests were therefore conducted with these two parameters.

6.1 Iteration testing

To test the minimum required number of iterations for the length constraints, that would still provide acceptable results, tests were done using the previously mentioned standard head. The easiest possible way was used for this, by trying out the different numbers and observing the results. This was done by comparing one of the predefined facial expressions, laughing, since it provided changes to the mouth region, the eyes and the forehead.

Figure 44: 1 iteration (left), 2 iterations (middle), 3 iterations (right)

The tests started with the number 1, which only provided a single calculation for each length constraint per frame. As we can see in Figure 44 above, the results are not really

satisfying. The forehead for starters, is supposed to wrinkle slightly, but instead only one hard wrinkle can be seen which looks rather like two separate scars. It is not perceived as very realistically because of the thickness of the wrinkle. Probably the most passable is the eye region, because only very slight changes exist to the neutral expression. The circular muscle around the eyes provide a slight compression and the nose wrinkles a bit, but the result looks alright. However, the region around the mouth is the worst, because of its strong deformation. The furrow around the mouth is too hard, and the mouth itself has a very unnatural form. Especially the mouth corners and the lower lips deform unnaturally as they seem to loose their original form quite a bit.

A single deformation is obviously not enough for a model of this resolution. With two iterations per frame the result is perceivable better. The region on the forehead is a little bit more wrinkly, though the wrinkles themselves still look too hard. Not much changed in the eye region, however the wrinkles there are not as hard anymore as they were with only one iteration. The mouth region is clearly better now, as it can be seen in Figure 44. The lower lip does not deform that much and keeps its form and the furrow around the mouth is not as hard anymore. Also the corners of the mouth are much smoother now. However the overall look is still not as realistically as desired, and the generated wrinkles not soft enough. Therefore another test with three iterations was done.

The wrinkles in the forehead region look very smooth now, and are also not as hard anymore. In my opinion they achieve the goal to make the region look soft and appear realistically enough. Now the eye region looks nearly the same as with two iterations, only the shadows appear to be slightly less hard again. Most changes appear in the mouth region. The lips now keep their form completely as do the mouth corners, and the furrows around them are again smoother and less hard. Three iterations are in my opinion therefore the minimum amount of iterations for a satisfying result.

6.2 Resolutions

The second series of tests used different resolution head models to observe if lower resolution models would still provide soft looking expressions. To allow the comparison of

the mesh models, the lower resolution ones were derived from the standard head, by using the poly reducer script that blender provides. Notable is that the polygon reduction by hand would probably allow to keep a better mesh topology and may provide better results with the implementation, than this script does.

For the tests the following three head models were used. The first one is the standard model, which is a slightly modified head scan, consisting of 5596 vertices and 10974 triangles. The second one is a copy of the first one, that was reduced by 25 percent and consisted of 4199 vertices and 8207 triangles. The third one is another copy of the standard head, but this one was reduced by 50 percent and consisted of 2808 vertices and 5472 triangles. For this comparison one of the predefined facial expressions was used again, but this time it was the expression anger.

Figure 45: Low resolution (left), medium resolution (middle), high resolution (right)

If we have a look at Figure 45 above, we can see that there are only slight differences between the different resolution models.The most remarkable difference is probably in the region of the mouth. The mouth corners of the low resolution model are not so clearly defined anymore and appear somewhat round. This can also be seen on the furrow surrounding it, which appears sharper on the high resolution model.The area between the eyebrows only has one deep furrow instead of the small wrinkles the high resolution model generates. Finally some of the wrinkles the nose produces are missing or not as strongly visible in the low resolution version.

However, the overall soft feeling of the model and of the facial expression is still visible in

my opinion and produces decently realistic results.

6.3 Frame-rate

A third series of tests was conducted for the frame-rate that the application could achieve. Therefore, the distance of the face model to the camera, and the head models resolution were changed. To test the influence of the distance on the framerate, the default head model consisting of 5596 vertices was placed 20 (near), 40 (medium) and 60 (far) units away from the camera. The frame-rate was captured once without any simulated muscle contraction, and the second time with. During contraction, the frame-rate drops quite a bit, due to the amount of constraints and necessary calculations, as it can be seen in Figure 46 below. The distance however has only little influence on the frame-rate.

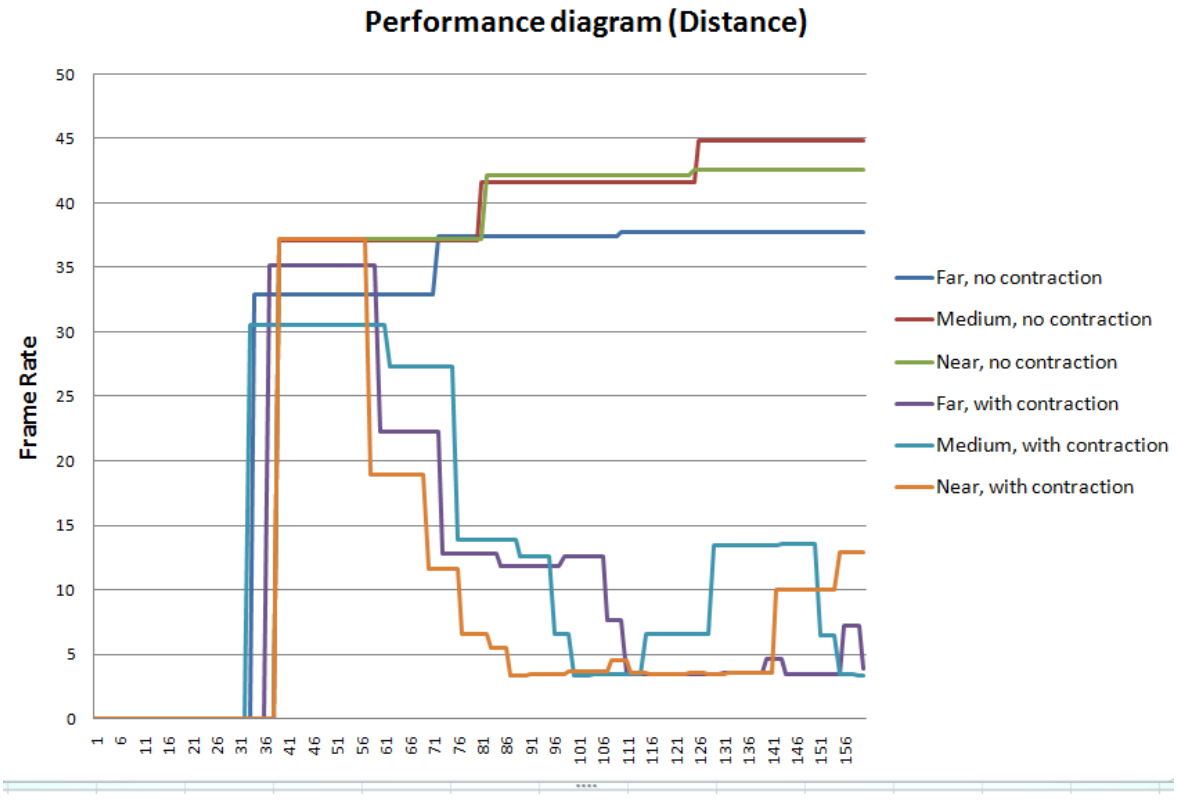

Figure 46: Showing the frame-rate for different distances

To test the influence of the models resolution on the frame-rate, the same head models were used as described in section 6.2. The frame-rate was captured without and with simulated muscle contraction. There is nearly no difference in in the frame-rate while no muscle contraction occurs, which can be seen in Figure 47 below. During contraction however, it is clearly visible that the low resolution model achieves a higher frame-rate, due to the fewer constraints it possesses. The medium and high resolution models on the other hand achieve nearly the same results.

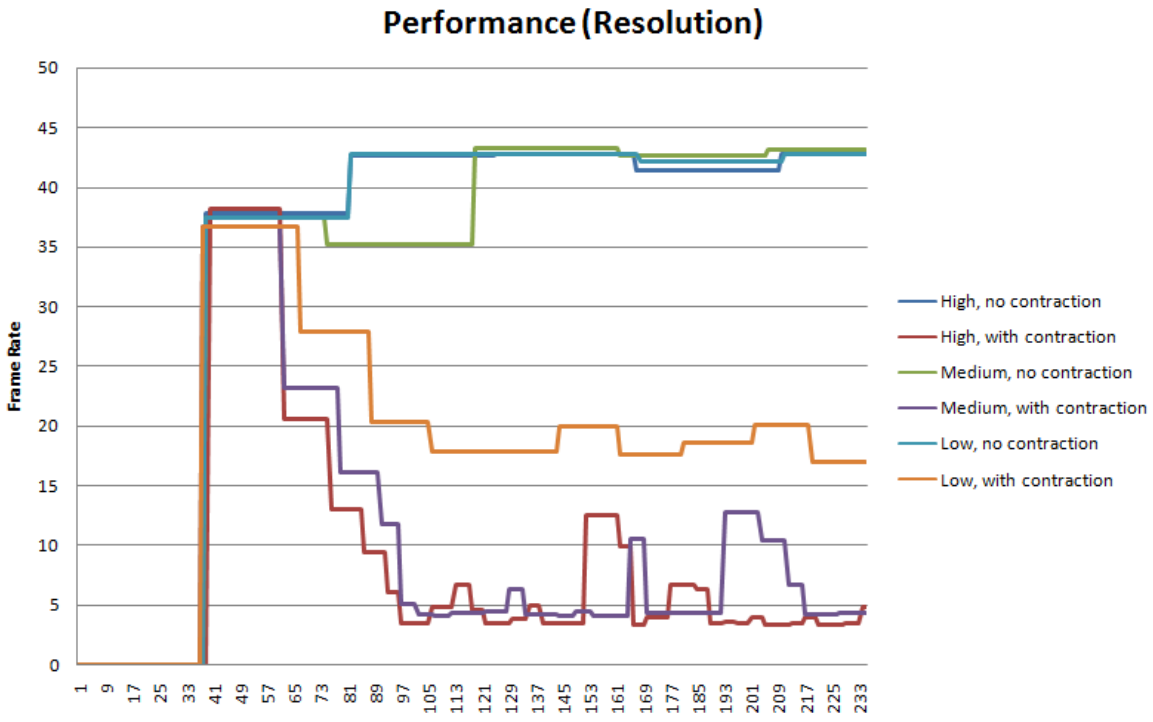

Figure 47: Shows the influence of different resolution head models on the frame-rate

7. Further Improvements

To really incorporate the developed tool into a production pipeline, some further improvements might be useful. The following features are probably the most necessary

Further Testing: To really get a grip of the applicability of the simulation, it has to be applied on more face models with different topologies. The implemented editor allows the easy fitting of the muscles to the correct regions, but to avoid extensive corrections the face models have to be provided in the same size. For this work a scanned head data was used, which has a relatively even topology, however it would be interesting to see how good of a performance could be achieved with a model especially created for this simulation. A models that has a higher resolution in areas were small wrinkles should be generated and a lower in other regions.

Wrinkle shader: Currently the presented system does all its calculation in the CPU and therefore the resulting frame-rate may be suboptimal. The implementation allows the sufficiently smooth manipulation of muscles and deformation of skin, depending on the mesh resolution, but at 10000 triangles the frame-rate is only passable. A probably better way would be to outsource some of the workload to the GPU. It might be possible to transform the deformation of the vertices into a vertex shader, which would ease the work on the CPU quite a bit, since the calculations of the wrinkles and the vertex normals are rather expensive.

Realistic skin: Currently only the provided texture for the head scan is used on the model, however it does not make the skin look very realistic. The visual appearance of the face would probably gain a lot from a normal map that simulates small pores. It might be also interesting to look into shader programming, to create a sub surface scattering effect, which emulates the light dispersion under the material surface, and allows to achieve very realistic looking skin.

8. Conclusion

The goal of this thesis was to develop a way to automatically create soft looking realistic facial expressions from a polygon model and to provide a simple way to animate it. To achieve this, two main problems were solved. A way was developed that allowed the creation of wrinkles upon skin compression, that gathered all the needed information from the provided mesh model. The solution derived from the idea of using a soft-body simulation to achieve realistically behaving skin. Through the creation of the wrinkles, the facial expression gets an overall wrinkly look, that simulates the impression of soft skin.
The second problem was to combine the skin simulation with a simple way to animate the face. After extensive research, a simple muscle simulation was considered the best way, and a slightly modified Waters vector muscle simulation was implemented.
With the combination of these two techniques, a tool was developed that allowed the deformation of a face model with a set of sliders, and allowed easy adjustments of the provided muscle rig with a real time editor. The tests that followed the implementation proved in my opinion that the developed simulation worked satisfyingly for the used head model with different resolutions. And only slightly lesser results were achieved with the lower resolution model.
However some improvements are still necessary, especially to make the performance increase, since the frame-rate is rather low due to the expensive CPU calculations that are necessary for the deformation of the vertices.

From the idea, research, theoretical development and implementation to the final writing of the thesis the project took about 7 month until its completion, and was done under the supervision of the computer entertainment research department of the university of Vienna.

9. References

1: Hodgins, J., Jörg, S., O'Sullivan, C., Il Park, S. and Mahler, M.; The Saliency of Anamolies in Animated Human Characters, 2010

2: Zhigang, D. and Junyong, N.; Computer Facial Animation: A Survey, 2007

3: Parke, I.; Computer Generated Animation of Faces, 1972

4: Escher, M., Pandzic, I. and Thalmann, N.; Facial Deformations for MPEG-4, 1998

5: Bickel, B., Botsch, M., Angst, R., Matusik, W., Otaduy, M., Pfister, H. and Gross, M.; Multi-Scale Capture of Facial Geometry and Motion, 2007

6: Bickel, B., Lang, M., Botsch, M., Otaduy, M. and Gross, M.; Pose-Space Animation and Transfer of Facial Details, 2008

7: Alexander, O., Rogers, M., Lambeth, W., Chiang, M. and Debevec, P.; Creating a Photoreal Digital Actor: The Digital Emily Project, 2009

8: Ekman, P., Friesen, W. and Hager, J.; Facial Action Coding System, 2009

9: Waters, K.; A Muscle Model for Animating Three-Dimensional Facial Expression, 1987

10: Duy Buy, B., Heylen, D. and Nijholt, A.; Improvements on a simple muscle based 3D face for realistic facial expressions, 2003

11: Waters, K. and Terzopoulos, D.; The Computer synthesis of expressive faces, 1992

12: Kähler, K., Haber, J., Yamauchi, H. and Seidel, H.; Head shop: Generating animated head models with anatomical structure, 2002

13: Dutreve, L., Meyer, A. and Bouakaz, S.; Easy Acquisition and Real-Time Animation of Facial Wrinkles, 2011

14: Bando, Y., Kuratate, T. and Tomoyuki, N.; A Simple Method for Modeling Wrinkles on Human Skin, 2002

15: Larboulette, C. and Cani, M.; Real-Time Dynamic Wrinkles, 2004

23: Skeel Keng-Siang, L.; Introduction to Soft Body Physics, 2008

16: Ogre - Open Source Graphics 3D Engine , http://www.ogre3d.org/

17: Ogre Application Wizard , http://www.ogre3d.org/forums/viewtopic.php?t=10543

18: Infinite, 3D Head Scan , http://www.ir-ltd.net/infinite-3d-head-scan-released

19: QuickGUI , http://www.ogre3d.org/tikiwiki/QuickGUI

20: Blender , http://www.blender.org/

21: Blender to Ogre Exporter , http://www.ogre3d.org/tikiwiki/Blender+Exporter

22: Ogre XML Converter , http://www.ogre3d.org/tikiwiki/OgreXmlConverter

24: Kelly, S.; Animation Tips & Tricks Volume 1, 2008

10. Appendix

10.1 Zusammenfassung

Abstract: In dieser Arbeit wird ein Weg zur Erstellung von realistischen, weich wirkenden, virtuellen Gesichtsausdrücken vorgestellt. Um dies zu ermöglichen wurde eine Simulation der Gesichtsmuskeln mit einer Simulation der menschlichen Haut kombiniert. Für die Gesamtsimulation wurde daraufhin ein Tool erstellt, welches das Austesten und die Manipulation der Muskeln ermöglicht.

Die realistische Animation der Gesichter virtueller Menschen, ist mit zwei Problemen verbunden. Man muss die komplexen Muskeln des Gesichtes simulieren, sowie die Eigenschaften der menschlichen Haut visuell nachbilden. Wenn dies nicht geschieht, dann erscheinen die virtuellen Charaktere seltsam hölzern, und ihre Gesichter maskenhaft. In dieser Arbeit wird eine Simulation vorgestellt, welche versucht diese Probleme zu lösen, um die Generierung von weich wirkenden, realistischen Gesichtsausdrücken zu ermöglichen.
Dabei wurde zur Annäherung der Muskelkontraktionen, Waters Vector Muscle Model verwendet in Kombination mit dem Facial Action Coding System. Dieses Muskelmodell besitzt ein relativ einfaches Design, bestehend aus nur zwei Muskelarten, ermöglicht aber die Generierung einer weiten Palette von Gesichtsausdrücken. Lineare Muskeln dienen zur Simulation aller isotonischen Muskelkontraktionen, welches den Großteil unserer Gesichtsmuskeln entspricht und Ringmuskeln zur Annäherung von kreisförmigen Schließbewegungen, wie sie um die Augen und Mund vorkommen.
Das Facial Action Coding System dient in dieser Arbeit als Grundlage zur Auswahl und Positionierung der Muskeln. Es war zudem hilfreich da es die Folgen der Kontraktion einzelner Muskeln oder Muskelgruppen im menschlichen Gesicht beschreibt. Um die erstellten Gesichtsausdrücke weich wirken zu lassen, wurde eine einfache Hautsimulation entwickelt, welche ohne jegliche Vorarbeit auskommt. Diese Simulation beruht auf einer kleinen Gruppe von Bedingungen welche die Eigenschaften der Haut zufriedenstellend annähert. Insgesamt gibt es drei Arten, Positions-, Längen- und Muskelbedingungen.

Positionsbedingungen simulieren die Elastizität der Haut und stellen den Ursprungszustand des Modells wieder her. Längenbedingungen dienen der Annäherung des Faltenwurfes der Haut und Muskelbedingungen übertragen die Muskelkraft und dessen resultierende Deformation auf die Hautsimulation.
Zum Austesten der Simulation wurde ein Programm entwickelt, welches mit Hilfe der Ogre Grafikengine erstellt wurde. Zur Programmierung des selbigen wurde Microsofts Visual Studio C++ verwendet. Das erstellte Tool ermöglicht das Laden von Gesichtsmodellen und Muskelsimulationen, und stellt eine Anzahl von Reglern bereit, um den Einfluss der einzelnen Muskeln auszutesten. Es enthält zudem einen Editor, welcher die Manipulation der einzelnen Muskeln ermöglicht.

10.2 Summary

Abstract: In this thesis, a way is presented for the creation of realistic, soft-looking, virtual facial expressions. A simulation of the facial muscles and a simulation of the human skin was combined for this task. In succession a tool was developed which allowed the testing of the simulation and the manipulation of the virtual muscles.

The realistic animation of virtual faces poses two problems. On one hand the complex muscles of the human face, and on the other hand the properties of the human skin have to be approximated. If this is not the case, the virtual characters seem strangely 'dead' and their faces mask-like. In this thesis a simulation is presented, which tries to overcome these problems to achieve realistic, soft-looking facial expressions.

To approximate the muscle contractions, Waters vector muscle model is used in combination with Ekmans facial action coding system. This muscle model has a relatively simple design, using only two muscle types, but allows the creation of a wide range of facial expressions. Linear muscles are used to approximate the isotonic contraction behavior, which fits the majority of our facial muscles, and sphincter muscles simulate the circular contraction that occurs around the eyes and the mouth. The facial action coding system is used as a guideline for the choice and the positioning of the muscles. It was also helpful since it describes the visual impact of single muscles and muscle groups on the surface of the face.

To achieve a soft-look on the facial expressions, a simple skin simulation was developed which can be applied without any preperation work. This simulation relies on a small set of constraints, which simulate the properties of the human skin sufficiently. Altogether there are three types, position, length and muscle constraints. The position constraints are used to simulate the elasticity of the skin and they undo all the changes which occur due to the muscle contraction. Lenght constraints are used to approximate the wrinkles which the skin generates upon compression and muscle constraints transfer the muscle force and the resulting deformation onto the skin simulation.

To test the simulation an application was developed, using the Ogre graphics engine and Microsofts Visual Studio C++ for the programming part. The developed tool allows the

loading of different head models and muskle simulations, and provides a set of sliders which simulate the contraction strength of the indicvidual muscles. It also provides an editor to allow manipulation of the muscles in real-time.

Leon Beutl - Lebenslauf

PERSÖNLICHE INFORMATION

- Geburtsdatum: 03.02.1987 in Wien
- Staatsangehörigkeit: Österreich
- Familienstand: ledig

AUSBILDUNGSWEG

- 2010 - *, Masterstudium Medieninformatik - Universität Wien
- 2008 - *, Bachelorstudium Japanologie - Universität Wien
- 2006 – 2010, Bachelorstudium Informatik - Universität Wien
  Abschluss: Bachelor of Science
- 2006, Ausbildung zum Multimedia Assistant - Bit Schulungscenter
- 2005 – 2006, Ableistung des Präsenzdienstes
- 1997 – 2006, BG 10 Ettenreichgasse
  Abschluss: AHS Matura
- 1993 – 1997, Volksschule Tesarekplatz

BERUFSTÄTIGKEITEN

- SS 2011, Tutor in Advanced Media Technology - Universität Wien
- SS 2010, Tutor im Praktikum der Medieninformatik - Universität Wien
- 2010, Blender3D Tutor - P.O.S.T
- 2008 - *, Standmanager - österreichische Lotterien

Wien, 2011