3D Facial Animation

Project Report

TU DELFT – The Netherlands
February 2nd to July 16th 2004

Romain FISSETTE

Supervised by Professor Leon Rothkrantz
And Ph.D student Ania Wojdel

0. TABLE OF CONTENTS

1. Acknowledgements & Summary

I'd like to thank Leon and Ania for their sympathy and help …

2. Introduction

As many European universities, ENSEIRB allows its students to carry out their third year of studies or their final project in another country with the Erasmus – Socrates exchange program. It's an excellent opportunity to apply the knowledge gained during the engineer studies in ENSEIRB and to show our skills in a professional environment. But it's over all an interesting way to improve a foreign language, to meet other students from a lot of different corners of Europe and to discover another country with its own culture and its specific working methods.

For those reasons, I quickly decided to do my final project with the Erasmus program. I wanted to go to the Netherlands because it's a country that always interested me due to its singular culture and way of life. Moreover it's an English-speaking country, so I didn't have to learn a new language. I'm really found of 3d graphics as a computer sciences domain as much as a drawing art, and I wanted to work on a project dealing with that subject. Professor Leon Rothkrantz from the KBS laboratory in Delft offered me the most interesting project. He told me he had a quite simple software developed in C++ and Open GL by a Ph.D. student that was able to display a 3D face, and to modify it in order to create facial expressions. He suggested me to improve that software by creating an animation system and a new interface to "move" properly the 3D face. I was glad with this subject and accepted. I was also really enthusiastic to work in an important laboratory on a sharp subject.

So I worked from February 1st to July 16th 2004 in the research laboratory of Knowledge Based System in ITS faculty of TU Delft. My supervisors were Professor Leon Rothkrantz and Ph.D. student Ania Wojdel. The aims of the final project were largely reached after these 5 months. I learned a lot of new programming skills, and I feel more clever about the latest applications of computer based recognition and synthesis of human facial expressions. This experiment was technically and culturally enriching, comforted me in my desire to work in the domain of 3D graphics and brought me a lot a new ideas on the subject. It was the last and for sure the best step in my studies.

3. Purpose

3.1 Original Subject from Professor Leon ROTHKRANTZ

The aim of the project was to improve an existing software designed originally to display still facial expressions on a 3D face, by developing an animation system and its interface to animate the face. This project took part of the work of Ph.D. student Ania Wojdel, who designed the original “Facial Expression Modeler” (FEM) software, and who is still working on her thesis about Facial Expression synthesis. Because it defines the context of my project, I will introduce her work.

3.2 An intelligent system for semiautomatic facial animation

3.2.1 Introduction

Facial animation finds applications in very diverse areas of our lives. It is used in entertainment industry (cinema, video games) as well as in more “serious” industries: virtual humans can be used for medical purpose (in prediction of plastic surgery or in speech therapy) or in remote learning and teaching.

F. Parke proposed the first 3D model of the human face in 1972. In his approach, the face was represented by sets of polygons representing the surface of the face and animation was performed by linear interpolation between two wire frames previously generated and stored in the library. This approach was tedious and data intensive. Two years later, Parke introduced a first parametric model in 1982. Since this time a lot of scientists improved the original model and in the 90’s, new models based on the anatomy of the face, structure and functionality of muscles became more and more popular as the efficiency of computers increased.

Although accurately modeling of the human face is still an important challenge in facial animation, it seems that in the last few years research about human-computer interaction dominates in facial animation. Among research on man-machine interaction, the important area of research is how to built a system with a “human face” which would be able to replace a real person in a conversation. All current existing systems for automatic generation of facial expressions are rule-based. That means that a set of rules describe dependences between the text, emotions and facial expressions. This set of rules is based on the work of many psychologists, who tried to describe universal links between facial expressions and the verbal content of a message.

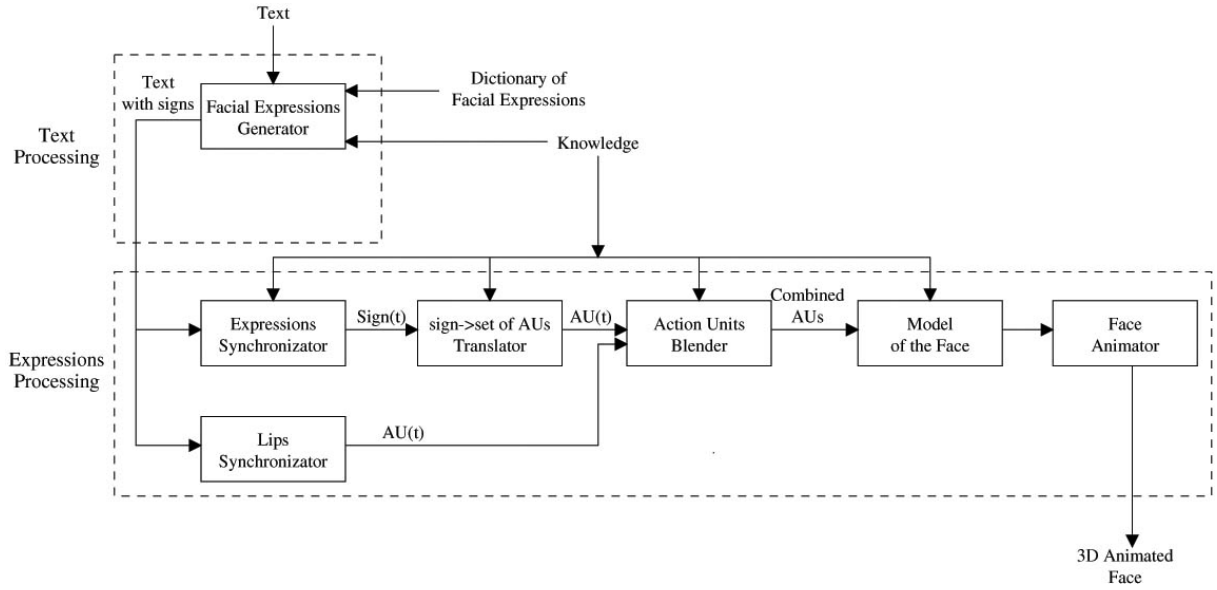
However, such systems for automatic facial animation are not flexible enough in order to generate realistic – person specific animations. They are based on universal rules for every person. But we know that everybody has his/her own way of talking and specific behavioral pattern. Moreover, depending on the kind of conversation or the outside

conditions, the same sentence can be said by the same person in completely deifferent ways.

3.2.2 Approach

Ania Wojdel and Leon Rothkrantz project is aimed at developing a system for semiautomatic facial animation. Their goal is to create an environment with a synthetic 3D face able to show some expressions where the user supported by the system can generate person-specific facial animations. The want to give a user the possibility to use a facial expressions' script language. With this script, the user can create an animation of a given face according to a text just by putting emblems representing appropriate expressions in a chosen place of the text.

3.2.3 System overview



The whole idea of the system for generating facial animations is based on a "facial expression script language".

In 1970 psychologists P. Ekman and F.W. Friesen developed an universal Facial Action Coding System (FACS) based on Action Units (Aus), where each facial expression can be described as a combination of Aus. Their work became de facto a standard in analysis of facial expressions not only in psychology but also in computer facial–related topics, that's why A. Wojdel and L. Rothkrantz decided to base facial movements on this system.

3.2.4 Facial Expressions Script Language

According to P.Ekman and F.W.Friesen, using Aus all possible facial expression can be shown. So it is possible to create a script language to define expressions using AUs as "letters" or "words" with its own syntax and semantics.

Syntax is just a set of Aus which are involved showing a given facial expression. Ekman's work only deals with the "activated or not" aspect of the action units, but it's more interesting to implement Aus with intensity ranging from 0% to 100%.

Semantics of facial expression is a verbal description of that expression. Such a description should describe what kind of information or feelings we want to communicate to the interlocutor while showing such facial expression.

All the defined facial expressions will be stored into a non-verbal dictionary of facial expressions. It will contain an symbol of all given expression, a description of the semantic and syntax as well as a picture showing this expression.

We also need to define a grammar, that means we have to describe rules about how facial expressions can be composed together. But it seems more practical to implement those rules in a way that they are not exposed to the user, and just used as an intuitive support.

3.2.5 Design of the system

The system has a modular structure, where each module is dedicated to a given task and uses its own knowledge about dependences between facial expression and/or Action Units.

It is divided in two parts: Text processing and Expressions Processing. The first part consists of only one module: Facial Expressions Generator and its based on interaction with a user. On the basis of input in form of written text and using the facial expression dictionary, the user designs facial animations. The Expression Processing part contains 6 modules. Input data is processed automatically through all of them. Task of this part is, first of all to synchronize facial expressions in time. The facial expressions have to be then transformed into a set of Action Units which are then blended in such a way that they can be showed. As the output the animation of a synthetic 3D face is displayed.

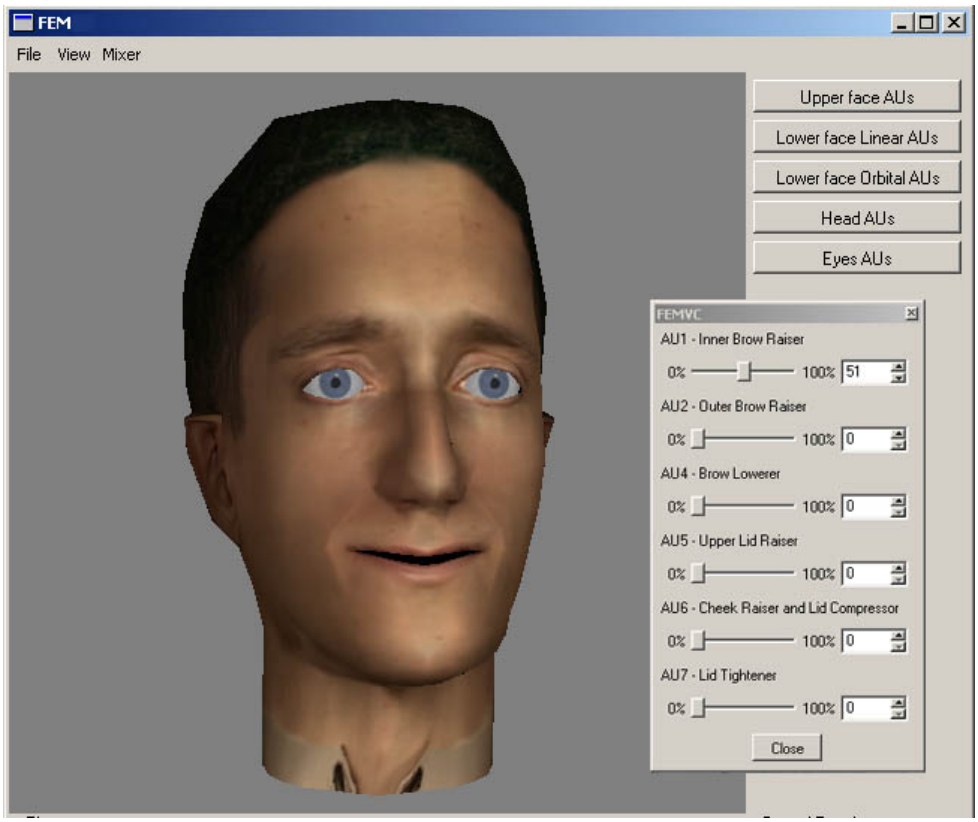
3.2.6 The Facial Expression Modeler

This program is the result of Ania Wojdel's first year of work. It fulfills the main job of the Expression Processing part in the whole system. It shows a realistic 3D face and allows user to synthesize any possible expression. It proves that it's possible to implement a Facial Expression Modeler from P. Ekman theory and single Action Units Models and to make it work in real time. Moreover it emphasizes and solves many problems about Action Units co-occurrences and Expressions Blending.

3.2.6.1 The interface

The software is one single executable file. It loads a 3D face model, which is displayed in the main window, and allows the user to activate every 44 action units with appropriate sliders from 0 % to 100 %. These action units are divided into 5 groups according to their behavior and to the part of the face they act on: upper face, lower face linear, lower face orbital, eyes and head.

The 3D face is a 3D Mesh directly based on Action Units. It is build from a triangular mesh modeled and textured with 3D Studio Max, a professional modeling software, and then exported as an ".ase" file (ASCII File Exporter). This 3D face is based on an existing person's face and is textured with a photomap so that it looks enough similar with the true face. The FEM software includes its own parser to read ".ase" files and build an internal 3D model.



3.2.6.2 Model design for a single Action Unit

Every single Action Unit is described with a few parameters, which characterize it: the area (set of vertices of the 3D face) in which an AU occurs, the type of AU (whether it squeezes skin into one point, move skin in one precise direction, etc), the direction in which it displaces vertices, and the value of AU's intensity.

In order to implement any given AU in the system, this description has to be transformed in some mathematical terms. So a possible model for Action Units can be :

- $\varphi(\mathbf{v}) : \Re^3 \rightarrow \Re+$: density function, which describe in fact both the area of AU occurence and "density" of the movement inside the area.
- $\psi(\mathbf{v}) : \Re^3 \rightarrow \Re^3$: the direction of the movement triggered by an AU.
- $\tau \in [0,1]$: the value of the activation intensity of a given AU.

Using the above components, the displacement of each point **v** on the 3D face surface can be calculated from the given formula :

$$\Delta(\mathbf{v}) = \psi(\mathbf{v})\ \varphi(\mathbf{v})\ \tau$$

Most of the Aus occurring on a human face can be described using this formula even though it assumes linear dependency between AU intensity and effective displacement. This doesn't hold however for AUs that incorporate long movements on a large area, where non-linearity becomes obvious; such Aus are e.g. head movements (AU51- AU56). In this case a more generic formula can be used:

$$\Delta(\mathbf{v}) = \psi(\mathbf{v}, \varphi(\mathbf{v}), \tau)$$

The above-proposed formalization of the AU description can be used in semi-automated implementation of the AU for a given person. The implementation proceeds in three steps.

- Ø The First step is to make 3D measurement of the real human face with a given AU 100% activated. Some selected points on the face and their movements have to be measured.

- Ø In the second step, an hypothesis on the generic form of functions ψ and φ must be stated. It should be based on the character of changes inflicted on the face (as measured in the previous step). While ψ and φ behavior heavily depend on the AU itself, their generic forms can be defined once for all subjects. In this way, an AU or at list the class that implements it, can be reused (with different parameters) to animate different persons with different wireframe models.

- Ø In the next step, we can optimize the parameters of the φ function so that it fits the length of the measured displacement, and the parameters of the ψ function so that it fits the directions of the displacements.

3.2.6.3 Example of AU implementation

In order to obtain the necessary measurements, Ania Wojdel asked a subject to show this AU and took pictures of a neutral and an AU1-showing face. She used 36control markers on one side of the subject's face and took simultaneous pictures od the frontal and lateral view of the face. In this way, it's relatively simple to manually measure facial movements.

To model a ψ function, she used the following formula:

$$\psi(\mathbf{v}) = [\cos(\alpha)\cos(\beta), \sin(\beta), \sin(\alpha)\cos(\beta)]$$

$$[\alpha, \beta] = \mathbf{A}.\mathbf{v} + \mathbf{c}$$

Where **v** is a point in the 3D space, **A** is a 2x3 matrix and **c** is a 2D vector. In this way, the image of the mapping ψ is a set of unit-length vectors with a linearly changing angle. The actual values of the parameters **A** and **c** were optimized to the measurements using MATLAB toolkit. She used Nedler-Mead method for nonlinear unconstrained minimization, and minimized the following function:

<function>

where **v**i is ith measured point and … is measured movement of this point.

As a density function φ she used a Gaussian shape:

<function>

Where … is a real number, **m** is a 3D vector and **B** is a 3x3 matrix. Again those 3 parameters were optimized to fit the measured data. This time, only the length of the movement was optimized, so the goal function was:

<function>

3.2.6.4 Mixing Aus

About the mixer …

3.2.6.5 Action Units Co-occurrences

About the blender …

3.3 Understanding the problem

The subject proposed by my Dutch and Polish supervisor was quite abstract. Consequently, I had a lot of knowledge to get before being able to think about the development. I first had to read the code and understand the main components of the existing FEM program. I had to understand the theory about FACS and Action Units. Then I had to study 3D model animation techniques in order to find a more concrete issue for the project.

There were a few requirements for the final issue. The application to be designed was supposed to :

- Be simple enough to allow me to design and implement it during my 5 months of project.

- Be complex enough to animate properly the 3D face, in terms of refresh rate and behavior credibility.
- Have an intuitive interface to create, load and save facial expressions and animation easily and quickly.

According to the preceding requirements, I decided with my supervisor to develop a simple animation system for the original FEM software, based on what I knew from 3D animation with professional 3D software. The user defines still facial expressions with the original functionality of the FEM program, and stores them in a library. He creates animations by placing graphically a sequence of expressions in a timeline. The software then animates the 3D face by interpolating the still expressions with an appropriate method. It should be possible to load and save expressions and animations. The development should integrate into the existing project.

3.4 Specifications of the project

3.4.1 Hardware requirements

- PC used for the development : Pentium IV 3.0 GHz, 1 Go Ram.

3.4.2 Software requirement

- Original FEM software developed on windows platform with Microsoft Visual C++.
- User interface developed with Trolltech QT interface toolkit.

The main requirement was to use as much as possible the functionalities of the original software, and to improve them when necessary. Mr. Rothkrantz gave me many ideas and issues to guide me for the development, all are introduced below but it was of course impossible to develop all those functionalities in the time I was given for the project.

3.4.2.1 Functional requirements

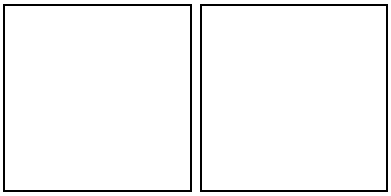
- Create, load and save still expression files in a library.
- Create, load and save animation files.
- Display the changes on the 3D face in real time.
- Adapt the software to load any 3D face model.
- Create semi-automatically the “muscles” for a given 3D face model.
- Improve the 3D rendering with Open GL.
- Create a friendly interface for Expression and Animation creation and editing.
- Create a friendly interface for the “muscle” creation and editing.
- Allow different methods of interpolation for the animations (linear, Newton, Lagrange…)

3.4.2.2 Non-Functional requirements

- Develop in ANSI C++
- Use QT interface toolkit.

3.5 Constraints

3.5.1 The QT graphic interface toolkit



Qt is a cross-platform C++ GUI application framework. It provides application developers with all the functionality needed to build state-of-the-art graphical user interfaces. Qt is fully object-oriented, easily extensible, and allows true component programming.

Since its commercial introduction in early 1996, Qt has formed the basis of many thousands of successful applications worldwide. Qt is also the basis of the popular KDE Linux desktop environment, a standard component of all major Linux distributions.

3.5.1.1 QT Object Model

The standard C++ Object Model provides very efficient runtime support of the object paradigm. On the negative side, its static nature shows inflexibility in certain problem domains. Graphical User Interface programming is one example that requires both runtime efficiency and a high level of flexibility. Qt provides this, by combining the speed of C++ with the flexibility of the Qt Object Model.

In addition to C++, Qt provides

- A very powerful mechanism for seamless object communication dubbed signals and slots.
- Query able and designable object properties.
- Powerful events and event filters.
- Scoped string translation for internationalization.

- Sophisticated interval driven timers that make it possible to elegantly integrate many tasks in an event-driven GUI.

- Hierarchical and query able object trees that organize object ownership in a natural way.

- Guarded pointers that are automatically set to null when the referenced object is destroyed, unlike normal C++ pointers, which become "dangling pointers" in that case.

Many of these Qt features are implemented with standard C++ techniques, based on inheritance from QObject. Others, like the object communication mechanism and the dynamic property system, require the Meta Object System provided by Qt's own Meta Object Compiler (moc). Think of the Meta Object System as a C++ extension that makes the language better suited for true component GUI programming.

3.5.1.2 Signals and Slots

Signals and slots are used for communication between objects. The signal/slot mechanism is a central feature of Qt and probably the part that differs most from other toolkits.

In most GUI toolkits widgets have a callback for each action they can trigger. This callback is a pointer to a function. In Qt, signals and slots have taken over from these messy function pointers.

Signals and slots can take any number of arguments of any type. They are completely type safe.

All classes that inherit from QObject or one of its subclasses (e.g. QWidget) can contain signals and slots. Signals are emitted by objects when they change their state in a way that may be interesting to the outside world. This is all the object does to communicate. It does not know if anything is receiving the signal at the other end. This is true information encapsulation, and ensures that the object can be used as a software component.

Slots can be used for receiving signals, but they are normal member functions. A slot does not know if it has any signal(s) connected to it. Again, the object does not know about the communication mechanism and can be used as a true software component.

We can connect as many signals as we want to a single slot, and a signal can be connected to as many slots as we desire. It is even possible to connect a signal directly to another signal. (This will emit the second signal immediately whenever the first is emitted.)

Together, signals and slots make up a powerful component programming mechanism.

3.5.1.3 The Meta Objects Compiler

The Meta Object Compiler (moc) is the program that handles the C++ extensions in Qt.
The moc reads a C++ source file. If it finds one or more class declarations that contain the "Q_OBJECT" macro, it produces another C++ source file that contains the meta object code for this class. Among other things, meta object code is required for the signal/slot mechanism, runtime type information and the dynamic property system.
The C++ source file generated by the moc must be compiled and linked with the implementation of the class (or it can be #included into the class' source file).

The main part of the process is fully automatic in Visual C++. The only necessary step is to create a .pro file with the environment and project configuration parameters, and path and name of all the project files. Then we run the simple “tmake” program given with QT toolkit, that generates a visual C++ project file (.dsw) from the .pro file and from all the project file. We can directly use that visual C++ project file to work on and compile the project.

3.5.1.4 OpenGL with QT

OpenGL is a standard API for rendering 3D graphics.

OpenGL only deals with 3D rendering and provides little or no support for GUI programming issues. The user interface for an OpenGL application must be created with another toolkit, such as swing on a Java platform, Microsoft Foundation Classes (MFC) under Windows - or Qt on *both* platforms.

The Qt OpenGL module makes it easy to use OpenGL in Qt applications. It provides an OpenGL widget class that can be used just like any other Qt widget, only that it opens an OpenGL display buffer where we can use the OpenGL API to render the contents.

QGLWidget provides functionality for displaying OpenGL graphics integrated in a Qt application. It is very simple to use: we inherit from it and use the subclass like any other QWidget.

QGLWidget provides three convenient virtual functions that we can reimplement in our subclass to perform the typical OpenGL tasks:

- paintGL() - Render the OpenGL scene. Gets called whenever the widget needs to be updated.
- resizeGL() - Set up OpenGL viewport, projection etc. Gets called whenever the the widget has been resized (and also when it shown for the first time, since all newly created widgets get a resize event automatically).
- initializeGL() - Set up the OpenGL rendering context, define display lists etc. Gets called once before the first time resizeGL() or paintGL() is called.

3.6 Anticipated Project Schedule

- Ø 1st Month : discovery of the environment
- Ø …

4. Conception

4.1 Start from the original Facial Expression Modeler

In order to have a complete idea of how is working the Facial Expression Modeler, and consequently how to use and improve it for animation, I will first propose an overview of its implementation.

Class MyApplication :

The FEM is compiled into a single executable file. As advised by the QT Toolkit documentation, the *main* method of the program only builds an instance of the *MyApplication* class. This class builds the different elements of the interface, loads the default 3D face Model, and its appropriate AUs, then establishes every connections between the separates modules of the program. Once all connections are done, the 3D face appears in the main window and the program is ready to be used.

4.1.1 The Interface

Its visual aspect has already been introduced in last Chapter. Let's describe its different components:

Class AppMainWindow :

This class inherits from the class *QmainWindow* defined in QT Toolkit. It is necessary for any QT software, and it is the main *widget* of the program. When building an instance of this class, we separate the main window of the program in specific parts like a grid, and then load the different parts of the graphic interface with their default parameters.

As I modified some modules, and added others, it has been necessary to modify this class.

Class OpenGLWindow :

This is the window where the 3D face is rendered. We will go deeper about 3D in the appropriate chapter; we just have to know that this class inherits from the *QGLWidget* abstract class defined in QT. This class offers an easy way to render OpenGL, we just need to implement the *initializeGL( ), resizeGL(int, int)* and *paintGL( )* methods defined in the abstract class. To draw the 3D model in the OpenGL window, we just have to obtain the *OpenGL List* of vertices and normals from the specific class where the structure of the 3D model is defined and stored (**class** FacialModel). This operation is done every time the 3D model id modified, i.e. every time the signal *ShowAUs( )* is send.

I didn't touch this class for my project as it was working well and didn't need improvement.

Class AusWindow :

This class implements the right window of the main application. It includes the five groups of sliders to set values for the Action Units and of course the five buttons to pop them up.

When a slider is moved, the signal *UpdateMask(int* values)* is send with the new intensity for the Action Units, and consequently the 3D face is modified, refreshed and displayed. The integer table "*values"* is a table of 66 unsigned integers, and represents the values of the 66 Action Units. It appears then clearly that a table of 66 integers is an excellent way to define an expression, or at least a specific position of the 3D face.

This class uses the **classes** *ComboSliderSpinBox* (to display sliders connected to spinbox) and *AusSlidersWindow* (to display the sliders and spinbox into a popup window, and get parameters for AUs intensity).

I modified this class a lot because I added lots of functionalities to the interface, and improved a lot of components and methods, especially for loading and saving expressions.

Class FileOperation :

This class deals with loading files and settings the parameters of the software. It is used by the main window for the functionalities of the top menus.

This class was just modified to accept loading on startup.

Class LightParamWindow :

Popup window used to set light parameters. It is connected with the class responsible for OpenGL display.

Class MixerOptionWindow :

Displays a board where the user can define the order of treatment of the AUs. It is connected with the **class** *AUMixer*.

4.1.2 The 3D Model

Class GeomObject :

Ania Wojdel designed its own 3D structure to display any 3D model. It is defined and implemented in this class. It contains the list of all the vertices and the list of all the normals of any given 3D model. It uses for that purpose the simple **classes** *Vector,*

Vertex, Texture, Light and *Matrix*. This class includes a method to load a 3D model from a ".ASE" file called *LoadFromASE(fstream*,QString*)* with an appropriate parser and all the methods to get and set any vertex or normal in the 3D model.

We can notice the method *SetDisplayGLList( )* that refreshes the 3D model and set it ready to be displayed in the OpenGL window.

4.1.3 Action Units

Class AU :

Uses **classes** *AUDensity, AUDirection, Vector* and *Vertex.* It's the abstract class that defines the interface for any type of AU.

…

4.1.4 Action Units Blender

Class AUBlender :

Implements co-occurrence rules. Included in the main application.

…

4.1.5 Facial Model

Class FacialModel :

Most important class of the programme. It gathers the 5 part of the 3D model of the face (face, eyes, lower teeth, upper teeth). Include every AU, and the AUMixer.

See specific method ShowAUs( ) that takes care about refreshing the 3D model relatively to the AU intensity. Needs improvements. This method is called after the signal *UpdateMask(int*)* is send.

I modified this class and created a new method ShowAUsFast( ).

4.1.6 Conclusions

- Ø The software really misses load and save functionalities to keep created expressions in files.
- Ø We need to define specific classes for expressions and animations.
- Ø The interface has to be improved for expression and animation editing.
- Ø The simplest way to animate the model seems to send the *UpdateMask(int*)* signal for every frame of the animation with appropriate interpolated values for Action

Units intensity. So we need to create an animation player to calculated the interpolated values and send the consequent signal.

Ø The ShowAUs method can be easily improved because the loops to refresh the 3D model before display are not optimized and take too many CPU cycles.

4.2 Expressions

4.2.1 Specifications

It is defined by a set of AUs intensity, a duration, an onset time and an offset time.

4.2.2 File Format

It is a standard text file with the ".exp" extension. Here is an example :

```
{
        FEM Expression File

        NAME Scared

        DURATION 1000
        ONSET 5
        OFFSET 65

        AU1 89
        AU5 88
        AU18 66
        AU27 59
}
```

4.2.3 Expression class definition

```
class Expression
{

public:
        Expression( const QString& name , int values[NUMBER_OF_ALL_AUS], int duration, int
        onset, int offset);
        Expression( const QString& filename );
        ~Expression();
```

```
        bool        load ( const QString& fileName );
        bool        save ( const QString& fileName );

        Qstring      name()   const
        void        rename(const QString& name )

        int         value(int i)
        void        setValue(int index,int val)

        int*        values()
        void        setValues(int* values)

        int         duration()
        void        setDuration(int ms)

        int         onset()
        int         onsetTime()
        void        setOnset(int onset)

        int         offset()
        int         offsetTime()
        void        setOffset(int offset)

        int*        animvalues(int ms);

        int*        interpolate(unsigned int percent);
        int*        interpolateTo(Expression* ,unsigned int percent);

        Expression   operator+ (const Expression&);

private:
        QString     n;
        int         v[NUMBER_OF_ALL_AUS];
        int         d;
        int         on;
        int         off;

        bool        read(fstream*);
        bool        write(fstream*);
};
```

4.2.4 Expression Library class definition

```
class ExpressionLib : public QObject
{
        Q_OBJECT

private:
        QList<Expression> ExpList;
```

```
public:
        ExpressionLib();
        ~ExpressionLib();

        int             AddExpression(const Expression*);
        int             RemExpression(const Expression*);

        uint            Count()

        Expression*     GetExpression(uint);
        Expression*     GetExpression(const QString&);

        static          Expression* Neutral;

public slots:
        void            OpenExpFile();
        void            LoadExpFile(const QString& filename);
        void            SaveExpFile();
        void            SelectExp(uint);
        void            SelectExp(const QString&);

        void            NewExpression(int*,int,int,int);

signals:
        void            ExpressionLoaded(Expression*,uint);
        void            ExpressionAdded(Expression*);

};
```

4.3 Animations

4.3.1 Specifications

There can be many ways to define an animation, I tried two different methods but I kept only the last because it is the simplest to use and it answers better to L.Rothkrantz and A. Wojdel wishes :

4.3.1.1 First idea: Key frames

An animation can be defined by a sequence of key frames (a set of AU intensity and a given occurring time or frame), which are interpolated by a user–chosen method (linear, Lagrange, Newton…).

In that case, animations do not use expressions with the definition that was given in previous chapter, but just abstract keyframes. Thus, all information about a given animation can be contained in a single file, without any knowledge on the expressions that appear in this animation.

Then to create an animation in the interface, we only need a simple timeline where we place keyframes that we create with the Action Units Window.

Related File Format:

```
{
        FEM Animation File

        NAME Anim01

        DURATION 3500
        INTERPOLATION_METHOD Linear

        FRAME00
        {
                AU7 0
                AU18 0
        }
        FRAME23
        {
                AU7 43
                AU18 25
        }
        FRAME48
        {
                AU7 70
                AU18 20
        }
        FRAME80
        {
                AU7 0
                AU18 0
        }
}
```

This idea is close to the standard way to create animation in a professional 3D software, but it is too abstract for the subject, and for the use we want to do with the software.

4.3.1.2 Second idea: Expressions sequence

An animation can be also defines as a sequence of known expressions, assuming that an expression is defined by a unique name, and that any given expression has the same parameters every time it is used in the software. The Expressions are taken in the library (loaded from files or created in the software) then interpolated by a chosen method.

In this case, the timing of the animation is subordinate to the timing of every expression it is made of, because it keeps the specific duration of the “fully activated” state of all expressions. But

it is still a lot customizable because the user can insert blank expressions, and we can add some parameters with the expressions. For example, we can add the ability to set for any expression included in an animation a use-quantity (in percents) for the duration and the activation.

We can also compound several sequences of expressions. For example we can make a sequence of expressions that only concerns the upper part of the face, and at the same time another sequence dealing with the eyes or with the complete head.

This idea is more compatible with A.Wojdel project, in which the final goal is to create animation from templates put next to words in a text.

Related File Format:

```
{
        FEM Animation File

        NAME Anim01

        INTERPOLATION_METHOD Linear

        TRACK01
        {
                EXP Smile
                INTENSITY 56
                LENGTH 100

                EXP Happy
                INTENSITY 56
                LENGTH 100
        }
        TRACK02
        {
                EXP HeadTurnRight
                INTENSITY 100
                LENGTH 100
        }
}
```

4.3.2 Animation class definition

```
class Animation
{
public:
        Animation(const QString& name);
        ~Animation();

        QString         Name() ;
        void            setName(const QString& N);

        int             Duration();

        int             TrackNb();
        int             addTrack();
        int             remTrack();

        void            setSeqMode(int,sequenceMode);
        void            activateTrack(int,bool);

        Expression*     getExpression(uint track, uint pos);
        int             addExpression(Expression*, uint track);
        int             insertExpression(Expression*, uint track, uint pos);
        bool            remExpression(uint index, uint track);

        int*            animvalues(int ms);

        bool            load( const QString& fileName );
        bool            save( const QString& fileName );

private:
        QString         name;
        int             trackNb;

        Track*          tracks;

        bool            read(fstream*);
        bool            write(fstream*);
};
```

4.4 The Animation player

4.5 Improving the interface

5. Implementation

5.1 Class Expression

5.1.1 Creation

5.1.2 Loading and Saving

5.1.3 Interpolation

5.1.4 The expression Library

5.2 Class Animation

5.2.1 Expected behavior

5.2.2 Class trackstage

5.2.3 Class Track

5.2.4 Creation

5.2.5 Animation Values

5.3 Class AnimPlayer

5.4 New interface Elements

5.5 Improvements in existing classes

6. Results

7. Conclusion

8. Glossary

9. Bibliography

10. Annexes

10.1 User manual

10.2 Maintenance manual

10.3 Source code