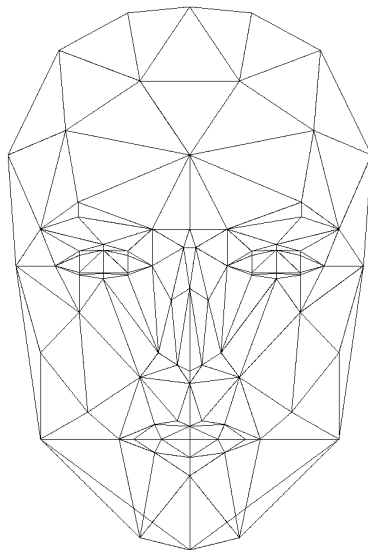

Kinect driven facial animation

Bachelor's thesis
June 29, 2016



Author: Guillermo Ojeda Noda
Director: Antonio Susín Sánchez
Bachelor degree: Informatics Engineering
Specialization: Computing



UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Abstract

Nowadays, facial animation is a core part of the character animation industry. From movies to video games, facial animation is done by most companies with the help of expensive equipment that capture real people's facial expressions as data and use it to animate their characters.

The Kinect device presents itself as an inexpensive alternative to this kind of equipment. With it, this project develops a facial animation application that makes use of the user's facial expressions to animate a 3D model. There will be a full specification of the project, track of the development in its different phases and an analysis on different aspects of the project, such as the limitations of the device, the development and possible extensions.

Resumen

Actualmente, la animación facial es una parte clave en la industria de la animación. Desde películas hasta videojuegos, la animación facial es realizada por grandes compañías a base de usar cámaras y herramientas de alto coste con el fin de capturar las expresiones faciales de personas de verdad y usarlas como datos para animar a sus personajes.

Kinect es un dispositivo que se presenta en este ámbito como una alternativa económica. Haciendo uso de él, este proyecto desarrolla una aplicación de animación facial que aplique las expresiones faciales del usuario a un modelo 3D. Habrá una especificación completa del proyecto, así como un seguimiento del desarrollo en sus distintas fases y un análisis de los distintos aspectos del proyecto, como las limitaciones del dispositivo, el proceso de desarrollo y las posibles ampliaciones.

Contents

	Page
List of Tables	vi
List of Figures	vii
List of Listings	viii
1 Introduction	1
2 Scope of the project	2
2.1 Objectives	2
2.2 Scope	2
2.3 Methodology and rigour	4
2.3.1 Development tools	5
2.3.2 Progress monitoring	5
2.3.3 Validation of results	5
2.4 Obstacles and risks of the project	6
2.4.1 Main program	6
2.4.2 Kinect	6
2.4.3 3D model	6
2.4.4 Schedule	7
3 Contextualization and references	8
3.1 Context	8
3.1.1 Areas of interest	8
3.1.2 Actors	8
3.2 State of the art	9
3.2.1 Facial motion capture	9
3.2.2 Facial recognition	10
3.2.3 3D viewers	11
3.2.4 Avatar video conference	12
3.3 Use of previous works	12

3.3.1	Facial motion capture	12
3.3.2	Facial recognition	12
3.3.3	3D viewers	12
3.3.4	Avatar video conference	13
4	Temporal planning	14
4.1	Description of tasks	14
4.1.1	Reading and understanding the problem	14
4.1.2	Project planning	15
4.1.3	Initial system set up	15
4.1.4	Main development	15
4.1.5	Final task	16
4.2	Time table	17
4.3	Resources used	17
4.3.1	Hardware resources	18
4.3.2	Software resources	18
4.4	Gantt chart	18
4.5	Action plan and alternatives	19
5	Budget and sustainability	20
5.1	Budget estimation	20
5.1.1	Hardware resources	20
5.1.2	Software resources	21
5.1.3	Human resources	21
5.1.4	General resources	22
5.1.5	Total budget	23
5.2	Budget control	23
5.3	Sustainability	24
5.3.1	Economic sustainability	24
5.3.2	Social sustainability	25
5.3.3	Environmental sustainability	25
6	Development of the project	26
6.1	First steps	26
6.2	Kinect's interface	27
6.2.1	Kinect for Windows SDK	27
6.2.2	Creating the interface	31
6.2.3	Upgrading the 3D viewer	35
6.3	Model loading and rendering	35
6.4	Adding Face Tracking into Kinect's interface	36
6.4.1	Face Tracking SDK	36
6.4.2	Extending the Kinect's interface	44

6.5	Meeting the first obstacle	48
6.5.1	FBX SDK	49
6.5.2	Main ideas of the FBX SDK	53
6.6	FBX helper	54
6.6.1	Creation of the helper	54
6.6.2	Modifying the model's interface	56
6.6.3	Testing the rendering and some deformations	60
6.7	Connecting both interfaces and the second obstacle	60
6.8	Workaround	64
6.9	Tweaking the animations	65
7	Analysis and conclusion	66
7.1	Changes to the initial project	66
7.2	Development analysis	67
7.3	Technical competences	67
7.4	Conclusion	68
	Bibliography	69

List of Tables

Table	Page
4.1 Summary of the time spent on each task.	17
5.1 Hardware resources budget.	21
5.2 Software resources budget.	21
5.3 Human resources budget.	22
5.4 General resources budget.	22
5.5 Total budget.	23
6.1 Head pose angles analysis.	41
6.2 Animation Units analysis.	43

List of Figures

Figure	Page
2.1 3D head used on this project.	3
2.2 Description of Kinect's components.	4
3.1 Andy Serkis performing the role of Golum in The Lord of the Rings.	10
4.1 3D rotations of a 3D head.	17
4.2 Gantt chart of the project.	18
6.1 Basic 3D viewer built at the beginning of the project.	26
6.2 3D reconstruction of a static scene with the use of Kinect Fusion. . .	28
6.3 <i>Face Tracking SDK</i> in action.	29
6.4 Kinect's captured image finally showing on our 3D viewer.	34
6.5 In case Kinect is not detected, an error is shown in the window. . . .	34
6.6 Rearranged 3D viewer.	35
6.7 Model rendered in our window.	36
6.8 Coordinate system of the <i>Face Tracking SDK</i>	37
6.9 Tracked points on a successful face tracking call.	40
6.10 Head pose angles.	41
6.11 Mask rendered on top of Kinect's image when a face is detected. . .	48
6.12 Improved rendering with the <i>FBX SDK</i>	59
6.13 Applying full weight to the happy <i>BlendShape</i> channel.	60
6.14 Applying full weight to both the happy and shocked <i>BlendShape</i> channels.	60
6.15 Jaw not moving properly with the head.	63
6.16 Correct head movement and animations.	65
6.17 Implemented more AUs.	65

List of Listings

Listings	Page
6.1 Initialization of the Kinect device.	31
6.2 Getting the next frame in the image streams into our buffers.	32
6.3 Initialization of the <i>Face Tracking SDK</i>	44
6.4 Initialization of the <i>Face Tracking SDK</i>	46
6.5 Header of the <i>FBX helper</i>	55
6.6 FBX loader of the model's interface.	57
6.7 Update function of the model's interface.	59
6.8 Function that saves the original state of the matrices in the pose. . .	61
6.9 Function that modifies the pose's matrices with the new head's pose.	61
6.10 Extended <i>update</i> function of the model's interface.	62
6.11 Extended <i>SetDefaultPose</i> function of the model's interface.	64

1. Introduction

Character animation has evolved throughout the years to the point that there is no need anymore to do it manually, but with the help of a computer it is possible to achieve a level of realism that 50 years ago would have been thought impossible. Movies, video games, TV series and many more have been directly affected by the breakthroughs on computer animation.

Yet there are things that a computer cannot do alone. One of the obstacles computer animation faces is when trying to animate the facial expressions of a human-like character with the highest realism possible. This is solved by taking the real facial expressions' data of a person through a set of cameras and giving it to the computer for treatment and finally animating the character. But here we find another obstacle: the kind of equipment that is usually used to do this task is quite expensive and directly decides who can achieve this kind of realism, casting aside the small producers and developers that can't afford them.

In this project we aim to develop a facial animation application that reduces the said expenses by using a cheap device called Kinect that has an overall SDK¹ and a *Face Tracking SDK*, thus making it easier to extract the real facial expressions data but adding a new obstacle: the complexity of translating this data into the character animation in real time. We will also study the limitations of this device and how we will try to overcome them.

This document has the objective to give the reader a detailed description of the project. It is divided in several sections, giving details about the scope, the context, the temporal planning, the budget and sustainability, the actual development and an overall analysis on the project with a conclusion, respectively.

¹Software Development Kit.

2. Scope of the project

This sections aims to make a description of our project. We are going to start with the formulation of the problem that we want to solve. Then, the scope of the project is going to be defined, describing all the possible obstacles that we may face during its development. Finally, we are going to describe how are we going to work in the 2.3 *Methodology and rigour* section.

2.1 Objectives

The main objective of our project is to develop an application that given the data from Kinect animates a 3D model's face in real time. With that in mind, we look for the highest realism possible using this device.

Aside from Kinect's data, we will need a 3D model to work with. Then, another objective would be finding a proper 3D model and making it suitable to our needs.

The last objective is to extend this application into something that a user can apply to his daily life: a video conference application that instead of using the video feed of a normal camera attached to the computer, uses Kinect's feed of data to animate 3D models for each user as an *avatar*.

2.2 Scope

We will focus on the main objective of creating the initial application, which is composed by a set of sub-objectives: create a 3D viewer, import and render our 3D model, collect Kinect's data and render its feed of video, treat the *Face Tracking* data and translate it into our model and, finally, improve the facial expressions until we are satisfied with the results. All of this will imply a huge amount of

work, therefore the last objective of extending the initial application into a video conference application is treated as an extra.

In order to achieve all of our objectives, we need to find a suitable 3D model that adjusts properly to what we want to do. As we focus on facial animation, we will use a 3D head. For an optimum realism, it would be necessary to do a 3D scan of the user's head, but since that would increase the expenses of our project and it would bring complexity as having a different model for every user would bring the need of some sort of constraints on how the modeling is done, we have decided to go for a 3D model found on the Internet (See figure 2.1). This model had to be as humanly as possible, thus making it possible to achieve said optimum realism.

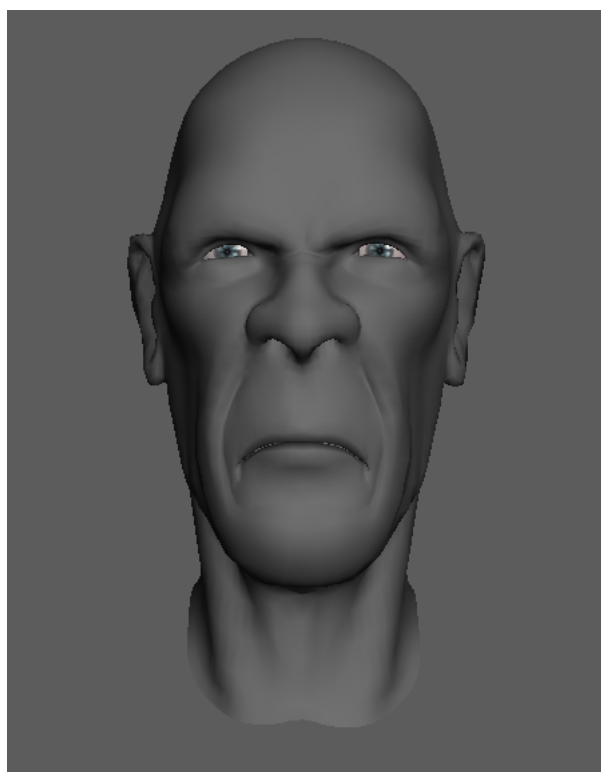


Figure 2.1: 3D head used on this project.

We also need a way to get a feed of data of a person's face in real time. This decision is the one that brings down the expenses of our project considerably: we decide to go for Kinect (See figure 2.2). We will talk more about Kinect in the 3.2 *State of the art* section.

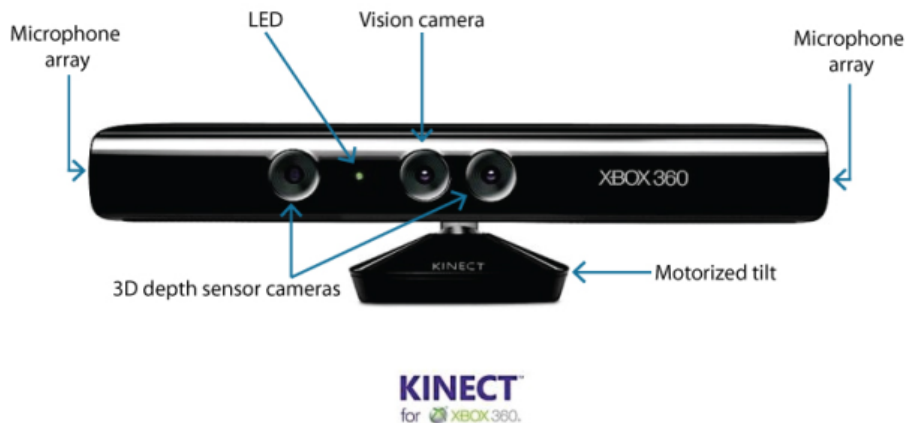


Figure 2.2: Description of Kinect's components.

2.3 Methodology and rigour

We have planned a rough development schedule that will help us carry out this project until its completion. That is what is explained in the following lines.

At first, we need to make a mock-up application that on one side shows the Kinect's feed of image with the *Face Tracking* data so we can assure ourselves that Kinect is working and tracking the face correctly. On the other side, it will show our 3D head facing the camera. This will allow us test in the following steps of development the correct translation of facial expression and head movement.

Once we have this initial application, the next step is to treat the data of the Kinect and translate it onto our 3D head. We will begin by just moving certain parts of the face and rotating the head. Then trying to smooth the movements with deformers, tweaking everything and so on. Therefore, it is going to be an iterative process in which we will work on trying to improve it until we reach the level of realism we want or are able to achieve.

Lastly, we may extend the application and design the desired video conference application, exploring ways of implementing it in a user-friendly way.

2.3.1 Development tools

The project will be developed entirely in C++ in a Windows environment, as the set of drivers for Kinect is made for this operating system. We will also use Visual Studio to work on the project. Moreover, we will implement everything graphics related in OpenGL 4.5 with the help of the libraries *SFML*, *GLEW* and *GLM*.

We will make use of the *Kinect for Windows SDK* that will allow us to read the data of the Kinect and treat it accordingly in our application. Also we will use the *Face Tracking SDK*, that will give us the information about the face of the user.

Lastly, we may use other programs like *Git Bash* to maintain version control, *Autodesk Maya 2016* or *Blender* to work with the mesh of the 3D head and *Adobe Photoshop CC 2015* for anything related to graphics or textures needed for the application.

2.3.2 Progress monitoring

We will use GitHub to keep version control and track the work done alongside the weekly meetings with the director. This will be initially done in a private repository that once the project is finished will be made public. We also may do weekly tests to assess the results achieved, so we can track if we are on the right path towards our objectives.

2.3.3 Validation of results

The method that we will use to validate the results is pretty straightforward: we will test the application with different people and a set of different facial expressions. The people chosen to test could be almost anyone, as to get a wider variety of results, and the set of facial expressions will be random, thus letting us know if the application works correctly with any combination of facial expressions and that it doesn't restrain itself to the physiognomy of some people.

And, if it comes to that, we also may test the final video conference application in a local server to see if it works properly with two or more users.

2.4 Obstacles and risks of the project

The development of the project has a set of objectives that are well defined and achievable, but we may face different problems. The main ones have been listed and explained in the following sections.

2.4.1 Main program

At first, we will develop a simple application where the user will see himself and the 3D head being animated as he moves his face or changes expressions, which will be quite the milestone. But as our last objective we want to create a video conference application, which is going to make us face the challenge of extending our initial application into a much larger system that is user-friendly.

2.4.2 Kinect

The Kinect for Xbox 360 has a limited frame rate of around 30 FPS¹, changing along the resolution, which is a great handicap. When the user moves the head around quickly or changes facial expressions constantly, this frame rate may cause a loss of data which will turn into a poorly animation.

This could be solved by using the Kinect for Xbox One, which has two times its frame rate. But as we are trying to keep this project almost expenseless, we will stick with the one we have.

2.4.3 3D model

Trying to imitate the facial expressions with an optimum realism is what could be the biggest obstacle of the entire project. As the model is a set vertices and faces, the movements that it can achieve by simply moving them are quite far from being real expressions. That's why we will try to implement deformers that will smooth the movements and bring realism to our project, which is not an easy task.

We are also working on a real time environment, so we fight against the clock when gathering Kinect's data and animating in order not to lose frames in our application. This leads to an essential need of efficiency in our code.

¹Frames Per Second

2.4.4 Schedule

This project is to be developed in a matter of 4 months, making it quite difficult not to meet problems regarding time. Nevertheless, we will define a temporal planning that, alongside the weekly meetings with the director, hopefully will let us achieve all of our objectives.

3. Contextualization and references

This section is divided in two: the first one describes the context of the project, that is, a brief description of the area of interest that the project is about and the actors affected, while the second one gives information about the state of the art of the areas we are going to explore in the project.

3.1 Context

In this section we are going to do a description of the areas of interest that this project is about and we are going to explain which are the actors that are going to be affected with its development.

3.1.1 Areas of interest

The main area of interest of this project is technical, which involves aspects like the creation of the application, which is basically a 3D viewer, the treatment of 3D models and the extension of an initial application onto a functioning video conference application.

There also is a *realism* area of interest, as to how good enough is a facial animation. This is mainly a subjective opinion, as it is almost impossible to imitate every facial expression of a human being to perfection.

3.1.2 Actors

The development of our project involves several actors, which are listed and described in the following lines:

Project developer

This project has only one developer, myself. I will be doing every task that is needed, which means I will be working on the planning of the project, the documentation, the information research, the coding, the design and the testing.

Project director

The director of this project is Antonio Susín Sánchez, associate professor from the *Applied Mathematics* department of the *Polytechnic University of Catalonia*. His role is to supervise the project developer with weekly meetings to assess that the project is being carried out properly according to the temporal planning and the objectives are being achieved. He also has to guide him when difficulties are met so that the project comes to a successful end.

Users

With our project's first objective we don't have that much of a user base, as the user will only be able to watch his expressions on a 3D head. But with our final objective, we aim at everyone with the video conference application as it could become a part of their daily life.

Other benefactors

Apart from the users, we can consider the developers as benefactors because they will have at hand the tools to develop their own facial animation applications. Moreover, this project will let them develop in a less expensive manner and without losing that much of realism in comparison to the expensive equipment used by professional producers or developers.

3.2 State of the art

This section's goal is to give information about the state of the art of the different elements that form our project. In the following lines we have listed the different areas that our project is going to explore, detailing each of them.

3.2.1 Facial motion capture

Facial animation is an essential part of big companies that focus their work on animation. Nowadays, big animation studios apply a massive set of cameras and equipment to record the actors' movement and, more specifically, their facial expressions [12] [4] (See figure 3.1).



Figure 3.1: Andy Serkis performing the role of Golum in The Lord of the Rings.

This kind of equipment and cameras turn to be highly expensive and sometimes exclusive to each studio. Here is where Kinect comes into place, reducing the expenses substantially.

Kinect [9] is a cheap device that most people think that it only works with its gaming console: the Xbox 360. Actually, it can be connected to a computer and used to develop a wide variety of applications. It has a set of cameras that let us obtain feeds of depth and image through the *Kinect for Windows SDK* [10], which we will detail in the 6.2 *Kinect's interface* section.

3.2.2 Facial recognition

Facial recognition is the core part of this project, as a method or algorithm to obtain data regarding a person's face from some kind of image or video source. There are algorithms in the computer vision field to detect faces [15] that work really well and could be implemented into our project, as well as multiple SDKs that are explicitly made for this purpose, such as *Nexa | FaceTM* [3] and *FaceSDK* [7].

However, in our particular case, we are already set to make use of Kinect, which also offers its own powerful tool called *Face Tracking SDK* [8] that, connected with the feeds of data from *Kinect for Windows SDK*, offers plenty of information about faces detected by Kinect, their facial expressions, the head's movement and more. We will also detail its features in the 6.4 *Adding Face Tracking into Kinect's interface* section.

3.2.3 3D viewers

A 3D viewer can be defined as a program capable of loading 3D models, in order to show them to the user and ultimately letting the user interact with them. With that definition, every program with a 3D window includes a 3D viewer, so, all the programs used in 3D modeling and all the 3D-based video games are examples of 3D viewers. Knowing the huge amount of modelers and video games that exist, we can say that this field has been deeply explored.

One example of a 3D viewer is *libQGLViewer* [6], which is a C++ library based on *Qt* that eases the creation of OpenGL 3D viewers. And another example is a video game developed by this project's developer, myself, in addition to one of my colleagues: *keep running N nobody gets hurt* [13]. It is a simple 3D runner game that includes most of the features of a proper 3D viewer.

Regarding the loading of the 3D model, it depends on its format. We are able to export our 3D model from *Autodesk Maya 2016* (which is the environment where it was designed) into a wide variety of formats, but the more distinguished and approachable for this project are the following:

- **Wavefront OBJ** [17]: It is a geometry definition file format (.obj) first developed by *Wavefront Technologies* for its *Advanced Visualizer* animation package and is widely adopted by other 3D graphics application vendors. For the most part it is a universally accepted format. Its main characteristic is that it has a simple data-format and represents 3D geometry alone — namely, the position of each vertex, the UV position of each texture coordinate vertex, vertex normals, and the faces that make each polygon defined as a list of vertices and texture vertices. This makes this format the easiest to implement in our project.
- **Autodesk FBX** [16]: It is a proprietary file format (.fbx) developed by *Kaydara* and owned by *Autodesk* since 2006. It is used to provide interoperability between digital content creation applications. This file format is proprietary, however, the format description is exposed in the *FBX SDK* [2] which provides header files for the FBX readers and writers. Although it doesn't seem like an improvement from the OBJ, after taking a deep look into their documentation and apart from a geometry definition, the FBX format holds a lot more information: cameras, lights, animations, poses, constraints, etc. This may be the optimal file format for our project, but it is the hardest to implement.

3.2.4 Avatar video conference

There exists studies on avatar video conferencing on work scenarios [5], but there is no actual technology yet developed. As stated on the paper:

“Lower ratings for the avatar condition were partly due to users’ frustrations when the avatar system did not track them perfectly.”

Low budget systems like the one we are trying to build tend to meet problems, partially caused by the limitations of the hardware or by the fact that the video conference systems that already exist are too popular and people don’t see the need to change them.

3.3 Use of previous works

In this section we are going to discuss whether we should use the solutions that we have presented in the *State of the art* section or not in each of the areas.

3.3.1 Facial motion capture

As we have stated before, we decide to use a Kinect for Xbox 360 to develop this project. Clearly, we will be using every aspect of its SDK.

3.3.2 Facial recognition

We obviously are in the need of a way to detect the user’s face, as it will be key in our project being our main way of animating the 3D head correctly. And as we decided to use Kinect, the natural choice is to use the *Face Tracking SDK* tool.

3.3.3 3D viewers

Since I am used to work with the environment I developed for the video game, as a 3D viewer we are going to use its structure of managing windows, input and loading and rendering 3D models. This 3D viewer works with the OBJ format, but if the chance of swapping to the FBX format presents itself, we shall make use of everything that the *FBX SDK* offers us.

3.3.4 Avatar video conference

Being one of our objectives and the fact that there is no actual work we can take advantage of, building the video conference application will be taken as a challenge and built from scratch.

4. Temporal planning

This is the temporal planning section, which aims to describe the tasks that are going to be carried out in order to finish the project. There will also be detailed an action plan that summarizes the actions that need to be taken so the project is carried out in the desired time frame. However, there may be adjustments to the planning depending on the development of the project.

The project was started on February 1st, 2016 and its deadline is June 4th, 2016. Therefore, the schedule has to be designed given these time constraints.

4.1 Description of tasks

In this section we are going to describe the tasks that we have planned to do in order to carry out our project. On that note, the PC hardware resource is used in every task.

4.1.1 Reading and understanding the problem

The project starts with the project's developer reading the documentation related to Kinect in order to understand what are we going to do and what do we have available in terms of software when we decide start planning the project and implementing. This task is done in a matter of almost a month as the project is started on holidays, but it should not take more than 15 hours.

4.1.2 Project planning

This is the initial documentation, also done by the project's developer with the advise of the project's director. It basically covers everything in the GEP module, focusing on seven stages: scope of the project, contextualization and references (state of the art), temporal planning, economic management and sustainability, initial presentation, specifications on the specialty and oral presentation and final document.

This task defines what is going to be done in the following tasks and how. However, as we have a tight schedule and it takes around 125 hours, some of the development tasks are done at the same time. For this task we make use of the *Overleaf* \LaTeX software resource.

4.1.3 Initial system set up

In order to begin the development of the project, it is needed to set up the tools that we will require afterwards. As simple as it seems, it takes time to set up everything correctly and that is why we list it as an important task in our schedule. This task is done by the project's developer and it should take 5 hours.

For this project, we are working on a *Windows* environment. Therefore, we need the *Visual Studio IDE*¹ to create the main program, *Autodesk Maya 2016* and *Blender* to modify 3D models, *Adobe Photoshop CC* in the case we need to modify textures or images and *Git Bash* to keep version control of our project with GitHub. We also need *Kinect for Windows SDK v1.8* as it will install all the desired drivers and tools that will let us work with Kinect.

After checking that everything works as expected, we can start working on the most important task of our project.

4.1.4 Main development

As the main task of the project, it covers all the tasks related with implementation and testing of the program that achieves the objectives stated in the 2 *Scope of the project* section. It is carried out by the project's developer, it takes approximately 400 hours in total, makes use of almost all of the software resources and it is the task that starts using Kinect. It can be divided in the following phases or sub-tasks:

¹Integrated Development Environment

- **Viewer set up:** In the beginning, we will focus in the essential implementation of a 3D viewer that works with Kinect. The viewer will take Kinect's feed of image and print it on a window. Then, if Kinect detects a face we will print on top of it a mask with Kinect's *Face Tracking* data. This will allow us to track the face animation in the following phases of development and test that Kinect is working properly. Time it takes: 50 hours.
- **Model representation:** Once we have our initial viewer, we want to represent our 3D head. After modifying our model in a way that is easier to work with, we will extend the viewer by implementing a 3D model importer and renderer. Afterwards, we will change the design of the application and represent the model on the left side of the window and Kinect's feed of image with the mask on the right. Time it takes: 35 hours.
- **Basic interactions:** With our program finally showing on screen everything that we need to test results, we will start by translating some basic interactions of the *Face Tracking* data into our model. These basic interactions would be the movement of the head, in other words, the 3D rotations: pitch, roll and yaw (See figure 4.1). This will be done on a real time basis, thus making us implement this translation as efficient as possible. Time it takes: 35 hours.
- **Facial expressions:** Now we will dig deeper on the *Face Tracking* data. We will try to translate everything that we can onto our model, moving its whole mesh the same way as the face that Kinect is detecting. This will be an iterative process: we will keep working on it until a certain degree of realism is achieved. Time it takes: 80 hours.
- **Improvements:** Having our 3D head expressing itself correctly, we will implement deformers and other ways of making the facial expressions smooth and more realist. This is the phase where, if done right, we can certainly get promising results. Time it takes: 120 hours.
- **Extension:** In the case that we achieve the level of realism that we want and we have time left, we will extend the program and build a video conference application. Time it takes: 80 hours.

4.1.5 Final task

As the final task, we will test that everything works correctly, finish the documentation and prepare the delivery of the project and the final presentation. It should not take more than 35 hours, makes use of *Overleaf L^AT_EX* and it is done by the project's developer.

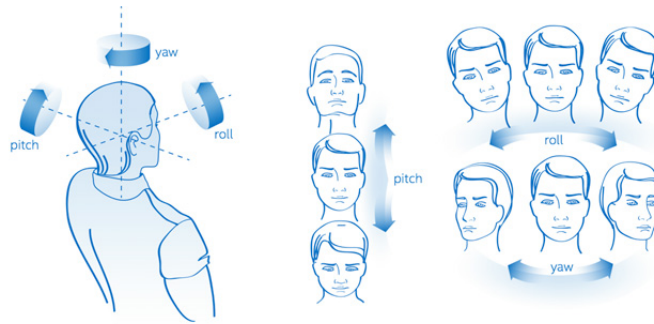


Figure 4.1: 3D rotations of a 3D head.

4.2 Time table

The following table 4.1 summarizes the time spent in each of the tasks described in the previous section.

Task	Time spent (hours)
Reading and understanding the problem	15
Project planning	125
Initial system set up	5
Main development:	400
· Viewer set up	50
· Model representation	35
· Basic interactions	35
· Facial expressions	80
· Improvements	120
· Extension	80
Final task	35
Total	580

Table 4.1: Summary of the time spent on each task.

4.3 Resources used

We can divide the resources that we are going to use in hardware and software resources. Here are both lists, including in which tasks each resource is used.

4.3.1 Hardware resources

- PC (with the following specifications: Intel Core i5-4590 @ 3.30GHz, 12 GB RAM, Gigabyte GeForce GTX 760 OC with 2GB GDDR5 and Samsung SSD 850 PRO 256GB): Used in all tasks of the project.
- Kinect: Used in all the development tasks of the project.

4.3.2 Software resources

- *Windows 10*: Used in all tasks of the project.
- *Visual Studio IDE*: Used in all the development and testing tasks of the project.
- *Autodesk Maya 2016*: Used in the main development task when needed.
- *Blender*: Used in the main development task when needed.
- *Adobe Photoshop CC*: Used in some tasks when needed.
- *Overleaf L^AT_EX* [14]: Used in the documentation related tasks.
- *Git Bash*: Used in all the development tasks of the project.

4.4 Gantt chart

Our schedule is represented by the Gantt chart shown in the figure 4.2. We have taken into account the holidays at the beginning of the project as a part of it.

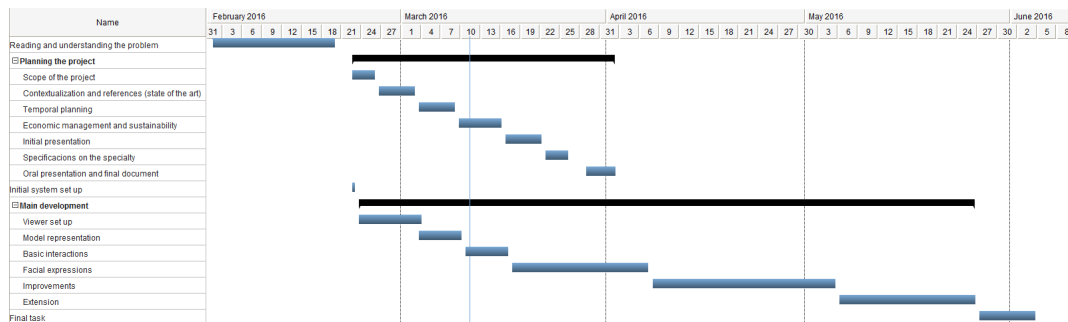


Figure 4.2: Gantt chart of the project.

4.5 Action plan and alternatives

The action plan is basically composed by the tasks described in the *4.1 Description of tasks* section and it will be carried out following the schedule detailed in the *4.4 Gantt chart* section. There will also be weekly meetings with the project director to monitor the project's development and give guidance when having difficulties.

On the other side, we may face problems that can affect the said schedule, such as hardware problems, difficulties when implementing or even extraordinary delays. If any of these happen, we shall act as we describe on the following lines as to bring these delays to a minimum:

Hardware delays

There is the possibility of malfunctioning hardware, as the PC or Kinect may break down. In that case, we shall find replacements or try and repair them, which implies delays on our schedule. As these hardware resources are an essential part of this project's development, we are tied to extend the duration of the project and keep our action plan as it is.

Implementation delays

This project's development has the same vulnerabilities as any other. We may find ourselves stuck in some parts of the implementation of the 3D viewer, model representation and facial expressions or any of the other tasks regarding code implementation. So we will be on the need of exploring ways to overcome these obstacles using the guidance of the project's director.

When facing these delays, we are to prioritize the completion of the main objective: creating an initial application that reads Kinect's data and animates the facial expressions of the user onto a 3D head. Therefore, we will adjust our schedule and use the time that we assigned to extend the initial application to manage these delays, even with the outcome of not being able to carry out this extension. And if this alternative is not enough, we will have to extend the duration of the project.

Extraordinary delays

There may be delays that escape our control, such as any of the project's human resources falling ill, bad weather conditions or personal matters. There is no way of predicting these delays and, if any of these do happen, we will have to extend the duration of the project as much time as needed.

5. Budget and sustainability

This section is about the budget and the sustainability of the project. It contains a detailed description of the costs of the project, describing both material and human costs, an analysis on how the different obstacles could affect our budget and an evaluation of the sustainability of the project. Just like before, the budget described is subject to modifications depending on the development of the project.

5.1 Budget estimation

We are going to do an estimation of the budget needed in order to develop our project. The budget is going to be divided in four sections, depending on the kind of resources that we are taking into account. These are hardware, software, human and general resources. We will also detail how all of these resources are related the tasks we described in the *4 Temporal planning* section. At the end of the section, we are going to show the total budget obtained by combining the four sections together.

To calculate the amortization, we will take into account two factors: the useful life of the resource and the fact that our project is going to last approximately four months.

5.1.1 Hardware resources

The table 5.1 contains the costs of the hardware that we are going to use in the development of the project.

Out of the two hardware resources, the PC is used in every task of this project, as we need it from the very beginning until the end. On the other side, Kinect starts to be used at the main development task until the completion of the project.

Product	Price	Useful life	Amortization
PC (including all the needed devices)	1000.00 €	4 years	184.95 €
Kinect	50.00 €	4 years	9.25 €
Total	1050.00 €	-	194.20 €

Table 5.1: Hardware resources budget.

5.1.2 Software resources

The table 5.2 summarizes the costs of the software that we are going to need to develop our project.

Product	Price	Useful life	Amortization
Windows 10 Pro	279.00 €	3 years	38.90 €
Visual Studio IDE	646.00 €	3 years	89.60 €
Autodesk Maya 2016	4800.00 €	1 year	665.74 €
Blender	0.00 €	-	0.00 €
Adobe Photoshop CC	870.84 €	3 years	120.78 €
Overleaf \LaTeX	0.00 €	-	0.00 €
Git Bash	0.00 €	-	0.00 €
Total	1,795.84 €	-	915.02 €

Table 5.2: Software resources budget.

As we stated in the previous section, we use the PC in every task of the project, so the operating system *Windows 10 Pro* is used in every task as well. That aside, all of the other software (except *Overleaf \LaTeX*) is used in all of the main development sub-tasks and in the final task, as we use all of them to implement and test our application. Finally, *Overleaf \LaTeX* is used in the project planning and in the final task, as we use it to do the GEP related documentation and then we finish it up by creating the final project's documentation, respectively.

5.1.3 Human resources

The table 5.3 shows the costs of the human resources involved in the development of our project. There are four roles in it, which are actually done by this project's developer, myself.

Product	Price per hour	Time	Cost
Project manager	50.00 €	125 h	6,250.00 €
Software designer	35.00 €	120 h	4,200.00 €
Software programmer	35.00 €	275 h	9,625.00 €
Software tester	30.00 €	60 h	1,800.00 €
Total	-	580 h	21,875.00 €

Table 5.3: Human resources budget.

To understand how the hours are given to each role in table 5.3, we will detail how the tasks in 4.1 *Description of tasks* are divided:

- The project manager will only be in charge of the project planning.
- The software designer and software programmer are going to work on everything that involves programming, in other words, all of the sub-tasks of the main development task. They also are assigned the task of reading and understanding the problem.
- The software tester will participate in the main development sub-tasks in a lighter way, testing the progress of the project, and the final task, which involves extensive testing to finally check that everything works.

Knowing the number of hours related to each task and making a rough calculation of every role's participation, we end up with table 5.3.

5.1.4 General resources

The table 5.4 summarizes the indirect costs of our project.

Concept	Price	Quantity	Cost
Electricity	0.13 €/kWh	174 kWh	22.62 €
Paper	29.03 €/pack	1 pack	29.03 €
Total	75.68€/year	-	51.65 €

Table 5.4: General resources budget.

The electricity reflected on table 5.4 is an approximation of the energy consumed by the hardware used throughout the whole project, which relates to all of its tasks. There also is the paper used during the project planning and at the final task when printing the project's documentation, which is also taken into account.

5.1.5 Total budget

Using the data shown in tables 5.1, 5.2, 5.3 and 5.4, we can detail the total cost of the project in the following table 5.5.

Concept	Cost
Hardware resources	194.20 €
Software resources	915.02 €
Human resources	21,875.00 €
General resources	51.65 €
Total	23,035.87 €

Table 5.5: Total budget.

5.2 Budget control

Our budget could need modifications if the development of the project makes it impossible to follow the original planning, which is likely to happen. This is directly related to the possible delays observed in the *4.5 Action plan and alternatives* section.

If difficulties are met, it is unlikely that we will need more software resources, as there are plenty of free applications that can act as a substitute of almost any commercial application. So, the budget sections we have to control are those related to hardware and human resources.

In case of malfunctioning hardware, we may need to add to our budget the costs of repairing them or even replacing them. But in an ideal situation, we won't need any more hardware resources apart from the resources that we have already listed in the previous section.

As stated in the *4 Temporal planning* section, the task that may take longer is the implementation and improving of facial expressions. This task involves software designers, programmers and testers, so we have to take into account that the costs of these human resources could grow if the problem becomes more difficult than we expected when setting our planning or if there is any of the delays observed in the *4.5 Action plan and alternatives* section.

In these cases, we are to use the Gantt chart to readjust our schedule in order to complete our project in time and with the project's original budget.

5.3 Sustainability

In this section we are going to evaluate the sustainability of our project in three different areas: economic, social and environmental.

5.3.1 Economic sustainability

In this document we already can find an assessment of the costs of our project, taking into account both material and human resources.

The cost stated in the *5.1 Budget estimation* section could be the only cost spent in the project, as we aim to create a working facial animation application and then extend it into a video conference application. Therefore, it will not need maintenance in terms of updates. However, it may need further development if we want to improve the quality of realism of the facial expressions or extend the video conference application into a perfectly working commercial application. This will add more cost to its development.

On the other side, it would be difficult to do a similar project with a significant lower cost. In terms of human resources, which are the main expenses of this project, we do not know if the project's planned time of completion is realistic enough, so we cannot say that it would be possible to do it in less time and reduce the costs. In fact, it is possible that they may increase.

Almost the same happens with the hardware and software resources. We need a computer and a Kinect device, so trying to avoid any of their cost is out of the question. And we work on a *Windows* environment, so we need to buy it. However, we could reduce the cost on software resources by not using *Visual Studio IDE* and *Adobe Photoshop CC*, as there are open-source applications or inexpensive ways of doing what this project needs from these, and restricting our model modification to *Blender*, in other words, not using *Autodesk Maya 2016*. So, ultimately, we could reduce our software resources budget in 876.12 € at most, which is almost a minimum impact on the total budget.

This project is going to be awarded a 9 in the economical viability area, since its price makes it affordable, but there is some possibility it could change.

5.3.2 Social sustainability

We focus on facial animation, which is a part of the 3D animation industry that is not available to everyone, since the technology that it requires comes with high expenses. Therefore, as we stated previously, our project tries to make it possible for small developers to endeavour in this field and, ultimately, deploy a 3D avatar video conference application that almost everybody can use.

Clearly, it has a positive impact on society, as we give a chance to people with low income. This promotes the growth of the mentioned sector, innovation on animation techniques and improvement of the competitiveness of small animation companies. On the other side, the only negative affected by it would be the big animation companies, as they would have an increase in competitors.

This project is going to be awarded an 8 in the improvement of the quality of life area, since it can help small developers widen their possibilities but is not that much of an improvement on video conference applications as people tend to prefer those that already exist.

5.3.3 Environmental sustainability

Throughout the development of our project we are going to have a computer running alongside the Kinect device. This means that these devices, the energy spent and the paper used to print the documentation are going to be the only resources used that affect the environment.

In fact, we can estimate the energy spent developing the project. We can suppose that our computer and Kinect consume an average of 300 W per hour when we are working on the project. As we need to use the computer in all the development of the project, which is 580 hours long, the energy we need is 174 kWh, which is equivalent to 66.98 kg of CO₂. It is a high amount of energy, but as we need to make extensive use of these devices there is no way to reduce it.

This project is going to be awarded a 9 in the resources analysis area, since the only part of the project that is not environmental friendly is the energy that we need to use, and it is not an important amount if we compare it to the amount that an average person uses nowadays.

6. Development of the project

This section follows the actual development of the project along the 4 months that it's taken. We aim to describe how the temporal planning in the 4 *Temporal planning* section was approached, the obstacles that we met and the alternatives that we decided upon, how this affected the project, the results of each development phase and, ultimately, the final results.

6.1 First steps

The project started exactly as it was devised in the temporal planning. Research was done on the different SDKs available to work with Kinect by means of their documentation on *Microsoft's* different websites. And the documentation for the GEP course was completed and now serves as the foundation of this documentation.

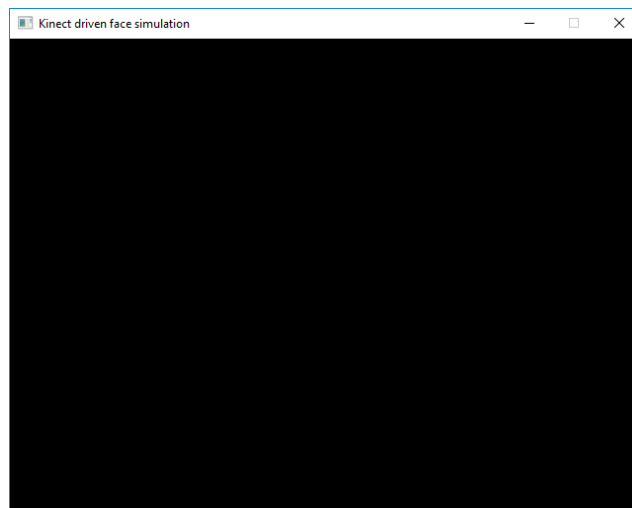


Figure 6.1: Basic 3D viewer built at the beginning of the project.

We also took off the development of the project by creating a blank application, in other words, a 3D viewer with nothing more than a window showing a black background. As it was mentioned in the *3.3 Use of previous works* section, we made use of an old project called *keep running N nobody gets hurt*. We removed the code that we didn't need and built our own basic 3D viewer with the same code structure, using C++ and OpenGL 4.5 with the help of the libraries *SFML*, *GLEW* and *GLM*. (See figure 6.1).

6.2 Kinect's interface

After having our 3D viewer implemented, we set to try out Kinect and its SDK so we familiarize ourselves with its features and start working with it. We decide to create its own interface that will manage everything related to it: powering up Kinect, obtaining and analyzing its feeds of data, rendering the image captured onto our window and, lastly, powering down Kinect in a proper manner. In order to explain this interface, we will detail first the *Kinect for Windows SDK* and then point which features we make use of and how in its implementation. Afterwards, we will detail how the 3D viewer was upgraded.

6.2.1 Kinect for Windows SDK

The *Kinect for Windows SDK* enables developers to create applications that support gesture and voice recognition, using Kinect sensor technology on computers running *Windows 7*, *Windows 8*, *Windows 8.1* and *Windows Embedded Standard 7*. In the following lines we will describe its main tool, the *Natural User Interface*, other underlying SDKs, APIs¹ and tools that empower the use of Kinect as a development tool and the data structures and functions of importance.

Natural User Interface

The *Natural User Interface* (NUI) is the core of this SDK. Through it we can access audio data streamed out by the audio stream and color image data and depth image data streamed out by the color and depth streams. In addition to the hardware capabilities, the *Kinect software runtime*, which is a part of this SDK, implements:

- A software pipeline that can recognize and track a human body. The runtime converts depth information into the skeleton joints in the human body. This makes it possible to track up to two people in front of the camera.

¹Application programming interface.

- Integration with the *Microsoft Speech APIs* so that developers can implement a speech recognition engine in Kinect-enabled applications. This makes it possible to add voice commands, such as "Start Tracking" or "Stop Tracking", to their applications.
- A tight integration with the *Face Tracking SDK*, which makes it possible to track human faces.

Kinect Fusion

Kinect Fusion provides 3D object scanning and model creation using a Kinect sensor. The user can paint a scene with the Kinect camera and simultaneously see, and interact with, a detailed 3D model of the scene (See figure 6.2).

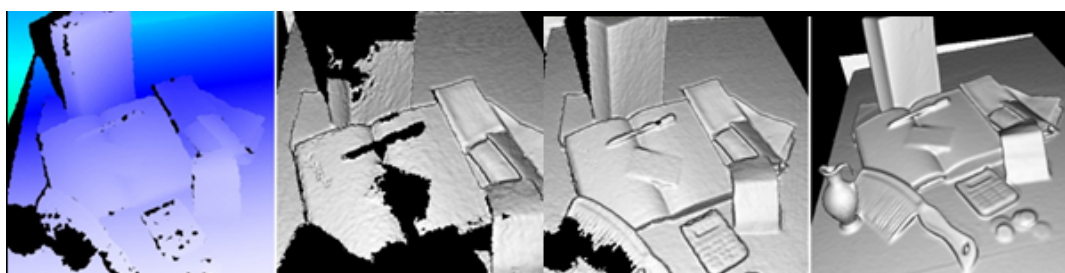


Figure 6.2: 3D reconstruction of a static scene with the use of Kinect Fusion.

Kinect Interaction

Kinect Interaction is an API that allows Kinect-enabled applications to incorporate gesture-based interactivity. It provides the following high-level features:

- Identification of up to two users and identification and tracking of their primary interaction hand.
- Detection services for user's hand location and state.
- Grip and grip release detection.
- Press detection.
- Information on the control targeted by the user.

Kinect Studio

Kinect Studio is a tool that helps record and play back depth and color streams from Kinect. This tool may be used to read and write data streams to help debug functionality, create repeatable scenarios for testing, and analyze performance.

Background Removal API

The *Background Removal API* provides green screen capabilities for a single person. This API uses various image processing techniques to improve the stability and accuracy of the player mask originally contained in each depth frame. The stream can be configured to select any single player as the foreground, and remove the rest of the color pixels from the scene.

Face Tracking SDK

The *Face Tracking SDK*'s face tracking engine analyzes input from a Kinect camera, deduces the head pose and facial expressions, and makes that information available to an application in real time (See figure 6.3). For example, this information can be used to render a tracked person's head position and facial expression on an avatar in a game or a communication application or to drive a natural user interface (NUI). This SDK will be explained in depth in the 6.4.1 *Face Tracking SDK* section.

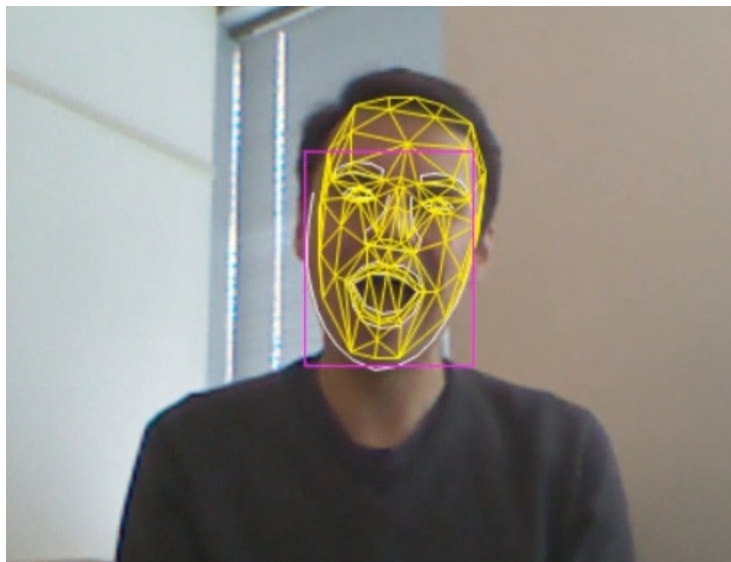


Figure 6.3: *Face Tracking SDK* in action.

Data structures and functions

This SDK, like any other, has its own data structures and functions that are used throughout its whole implementation. Therefore, we need to explain them as to understand what are they used for and, more importantly, how can we use them. But, as there are a great number of them, we will limit ourselves to the most relevant for our project at this point:

- **INuiSensor:** This is an interface that holds a reference to a Kinect sensor and let us work with it. It has relevant functions such as *NuiInitialize*, that allows

the initialization of the Kinect sensor with specific flags; *NuiImageStreamOpen*, which opens an image stream with a handler that we specify and specific settings, such as the resolution and the type of image stream; *NuiImageStreamGetNextFrame*, which gets the next frame of data of an image stream specifying its handler and a *NUI_IMAGE_FRAME* structure; *NuiImageStreamReleaseFrame*, which releases the frame of data from an image stream specifying its handler and a *NUI_IMAGE_FRAME* structure; and *NuiShutdown*, which turns the sensor off, although no action is taken if the sensor is already off.

- **IFTImage:** It is a helper interface that can wrap various image buffers. Its relevant functions are *Allocate*, which allocates memory for the image of passed width, height and format; *CopyTo*, which is a non-allocating copy method; *GetBuffer*, which gets the buffer of the image; *GetBufferSize*, which gets the size of the buffer; *GetHeight*, which gets the height of the image; *GetWidth*, which gets the width of the image; and *Release*, which releases the interface.
- **INuiFrameTexture:** Represents an object containing image frame data that is similar to a Direct3D texture, but has only one level (does not support mipmapping). Its relevant functions are *LockRect*, which locks the buffer for read and write access and has a parameter called *pLockedRect*, that is a pointer to a *NUI_LOCKED_RECT* structure that receives information about the locked region; *BufferLen*, which ; and *UnlockRect*, which
- **NUI_LOCKED_RECT:** Structure that defines the surface for a locked rectangle. Its relevant members are *pBits*, which is a pointer to the upper-left corner of the rectangle; and *Pitch*, which is the number of bytes of data in a row.
- **NUI_IMAGE_FRAME:** Structure that contains information about a depth or color image frame. Its relevant member is *pFrameTexture*, which is a pointer to an *INuiFrameTexture* object that can be used to access or manipulate frame data as a texture resource.
- **NuiGetSensorCount:** Function that gets the number of Kinect sensors on the machine, including disconnected ones.
- **NuiCreateSensorByIndex:** Function that creates an instance of the Kinect sensor with the specified index so that an application can open and use it.
- **FTCreateImage:** Function that creates an image object instance and returns its *IFTImage* interface.

6.2.2 Creating the interface

Using our 3D viewer's interface code structure, we will explain in the following lines how we, making use of the *Natural User Interface*, initialize the Kinect device, obtain its data each frame and render it.

Initializing Kinect

As seen in the listing 6.1, we power up the device in a safe manner, in other words, we test that it is connected to the PC, create its *INuiSensor* and initialize it, deciding which hardware and runtime features we wish to use. In our case, we select the color camera (*NUI_INITIALIZE_FLAG_USES_COLOR*), the depth camera (*NUI_INITIALIZE_FLAG_USES_DEPTH*) and the skeletons (*NUI_INITIALIZE_FLAG_USES_SKELETON*), even though we don't use the depth camera and the skeletons at this point, but will later on. After making sure that there weren't any errors, we open both data streams specifying their settings (resolution of 640 x 480 for the color camera and 620 x 240 for the depth camera) and then we create and allocate the *IFTImage* interfaces for each stream specifying their resolutions and image formats (*FTIMAGEFORMAT_UINT8_B8G8R8X8* for the video buffer and *FTIMAGEFORMAT_UINT16_D13P3* for the depth buffer), as to where we will store their data at each frame. And at each step, we check that there were no errors given by Kinect.

```
1 bool Kinect::initKinect() {
2     // Get a working kinect sensor
3     int numSensors;
4     if (NuiGetSensorCount(&numSensors) < 0 || numSensors < 1) return
false;
5     if (NuiCreateSensorByIndex(0, &sensor) < 0) return false;
6
7     // Initialize sensor
8     HRESULT hr = sensor->NuiInitialize(NUI_INITIALIZE_FLAG_USES_SKELETON
| NUI_INITIALIZE_FLAG_USES_DEPTH | NUI_INITIALIZE_FLAG_USES_COLOR);
9     if (FAILED(hr)) return false;
10
11     m_bNuiInitialized = true;
12
13     // Open the image streams
14     sensor->NuiImageStreamOpen(
15         NUI_IMAGE_TYPE_COLOR,           // RGB camera?
16         NUI_IMAGE_RESOLUTION_640x480,  // Image resolution
17         0,                               // Image stream flags
18         2,                               // Number of frames to buffer
19         NULL,                            // Event handle
20         &rgbStream);
21 }
```



```

22  sensor->NuiImageStreamOpen(
23      NUI_IMAGE_TYPE_DEPTH,           // Depth camera
24      NUI_IMAGE_RESOLUTION_320x240,  // Image resolution
25      0,                               // Image stream flags
26      2,                               // Number of frames to buffer
27      NULL,                             // Event handle
28      &depthStream);
29
30  // Create the IFTImage interfaces that will hold the image stream's
31  // buffers
32  m_VideoBuffer = FTCreateImage();
33  if (!m_VideoBuffer) return false;
34
35  hr = m_VideoBuffer->Allocate(640, 480, FTIMAGEFORMAT_UINT8_B8G8R8X8);
36  if (FAILED(hr)) return false;
37
38  m_DepthBuffer = FTCreateImage();
39  if (!m_DepthBuffer) return false;
40
41  hr = m_DepthBuffer->Allocate(320, 240, FTIMAGEFORMAT_UINT16_D13P3);
42  if (FAILED(hr)) return false;
43
44  return true;

```

Listing 6.1: Initialization of the Kinect device.

Obtaining the data

In terms of obtaining the data from the image streams, as seen in the listing 6.2, we do the same process on both streams. We obtain the next frame from the stream and store it into a *NUI_IMAGE_FRAME* structure, checking that if there is no next frame it means we don't have to update the buffer. Then, we obtain the *INuiFrameTexture* object from the previous *NUI_IMAGE_FRAME* and from it we get the data that we need, which we store into a *NUI_LOCKED_RECT*. Checking that data obtained is not bogus, we copy it into our buffers. This way we have available Kinect's data every frame.

```

1  void Kinect::getKinectVideo() {
2      NUI_IMAGE_FRAME imageFrame;
3      NUI_LOCKED_RECT LockedRect;
4      if (sensor->NuiImageStreamGetNextFrame(rgbStream, 0, &imageFrame) <
5          0) return;
6      INuiFrameTexture* texture = imageFrame.pFrameTexture;
7      texture->LockRect(0, &LockedRect, NULL, 0);
8      if (LockedRect.Pitch != 0)
9      { // Copy image frame
10         memcpy(m_VideoBuffer->GetBuffer(), PBYTE(LockedRect.pBits),
11             std::min(m_VideoBuffer->GetBufferSize(), UINT(texture->BufferLen())));

```

```

10     }
11     else
12     {
13         std::cout << "Buffer length of received image texture is bogus"
14         << std::endl;
15     }
16     texture->UnlockRect(0);
17     sensor->NuiImageStreamReleaseFrame(rgbStream, &imageFrame);
18 }
19
20 void Kinect::getKinectDepth() {
21     NUI_IMAGE_FRAME pImageFrame;
22     NUI_LOCKED_RECT LockedRect;
23     if (sensor->NuiImageStreamGetNextFrame(depthStream, 0, &pImageFrame)
24     < 0) return;
25     INuiFrameTexture* pTexture = pImageFrame.pFrameTexture;
26     pTexture->LockRect(0, &LockedRect, NULL, 0);
27     if (LockedRect.Pitch != 0)
28     { // Copy depth frame
29         memcpy(m_DepthBuffer->GetBuffer(), PBYTE(LockedRect.pBits),
30         std::min(m_DepthBuffer->GetBufferSize(),
31         UINT(pTexture->BufferLen())));
32     }
33     else
34     {
35         std::cout << "Buffer length of received depth texture is bogus"
36         << std::endl;
37     }
38     pTexture->UnlockRect(0);
39     sensor->NuiImageStreamReleaseFrame(depthStream, &pImageFrame);
40 }

```

Listing 6.2: Getting the next frame in the image streams into our buffers.

Rendering

The rendering of this interface differs from the other interfaces in our 3D viewer, as we don't render a 3D model or anything of the like. Instead, we have a buffer that holds the color camera's data with the resolution that we specified when opening its image stream, so in order to properly view it in our 3D viewer, we have to render it as a 2D billboard. We do this by using our buffer as a texture and then in the rendering our specific fragment shader does the work (See figure 6.4. We should also say that, if the Kinect device is not connected to the computer, the interface uses an image with the message "Kinect device is not detected" as the 2D billboard (See figure 6.5).

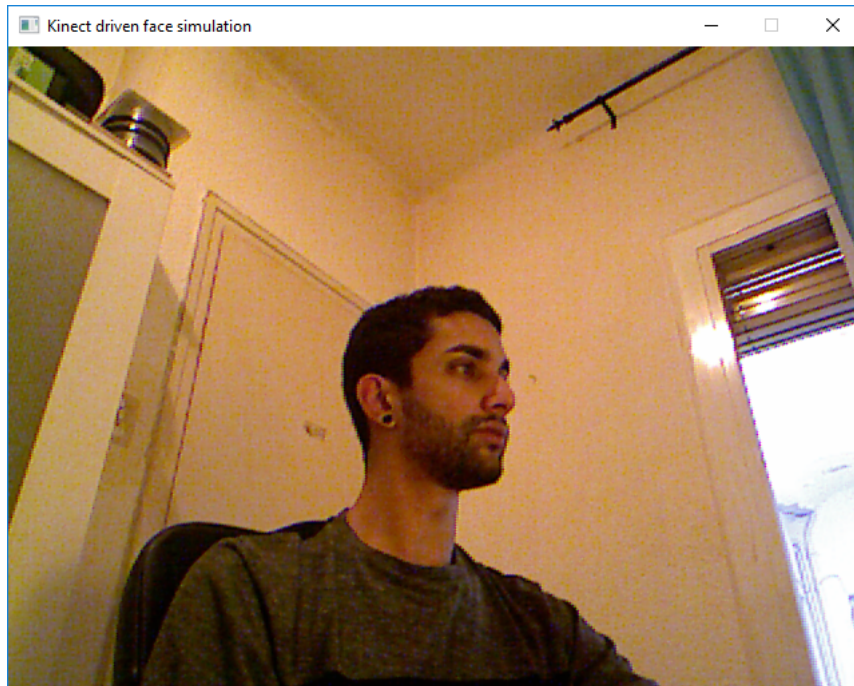


Figure 6.4: Kinect's captured image finally showing on our 3D viewer.

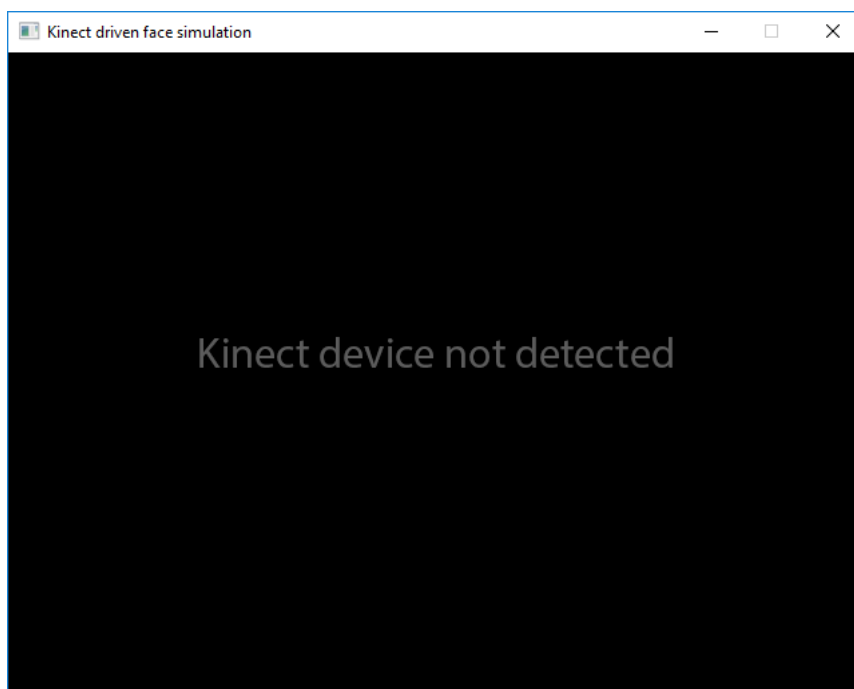


Figure 6.5: In case Kinect is not detected, an error is shown in the window.

6.2.3 Upgrading the 3D viewer

Once the Kinect's interface was implemented, we decided to rearrange the window of the 3D viewer as to show on the right side Kinect's captured images and on the left side a black background where our 3D model will be placed. This meant modifying the rendering of the Kinect's interface to place the 2D billboard on the right and doubling the width of our 3D viewer's window, specifically, changing its resolution from 640 x 480 to 1280 x 480 (See figure 6.6).

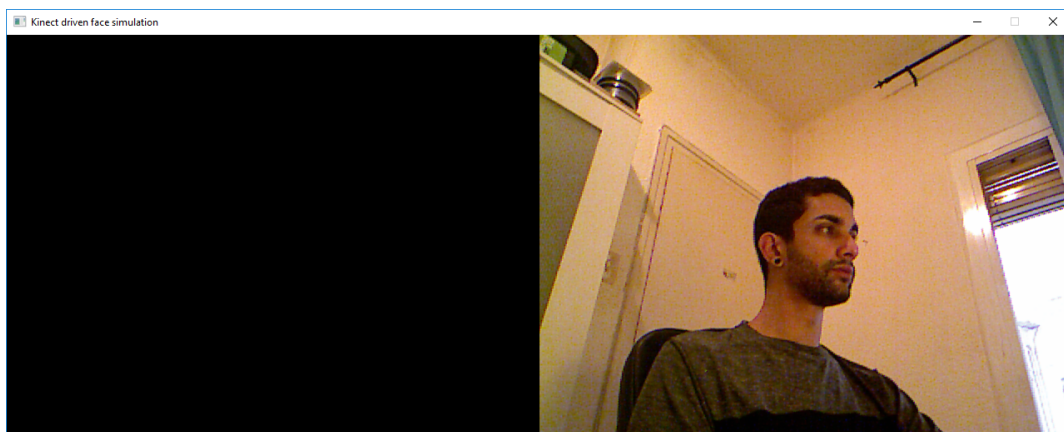


Figure 6.6: Rearranged 3D viewer.

6.3 Model loading and rendering

As we had Kinect's images showing on our window, at this point we wanted to render our model on the left side. Taking into account that our 3D viewer worked with the OBJ format and we decided that we would go with it, we exported the model triangulated in this format from *Autodesk Maya 2016*.

In order to load the 3D model, we only had to slightly adapt the OBJ loader that we had. However, in the rendering we had to change the camera's position and apply a translation and scaling to our model as to have it well placed in our window (See figure 6.7). This was done in the model's own interface, which was part of the original 3D viewer.

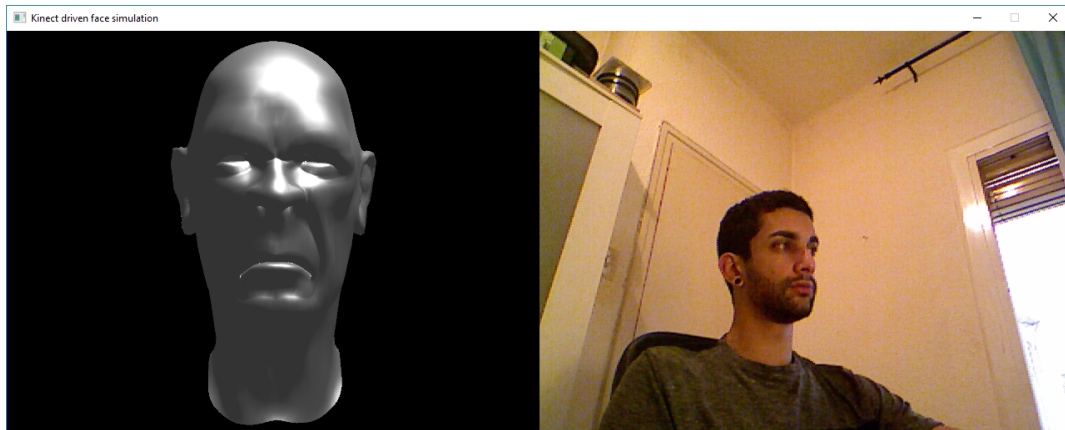


Figure 6.7: Model rendered in our window.

6.4 Adding Face Tracking into Kinect's interface

Having our window with its final design, it was time to try and detect faces and also print a mask on top of Kinect's images as a way of knowing if a face is being detected and how, so we are able to test the accuracy of the animations and the quality of the facial recognition in later stages of the project. Therefore, we will detail first the *Face Tracking SDK*, then explain how we extended the Kinect's interface with it and, finally, how the mask was rendered.

6.4.1 Face Tracking SDK

The *Face Tracking SDK*, together with the *Kinect for Windows SDK*, enables developers to create applications that can track human faces in real time. In the following lines we will describe how the SDK works, which are the new interfaces that it introduces, the outputs that it emits and some data structures of importance.

Coordinate system

The *Face Tracking SDK* uses the Kinect coordinate system to output its 3D tracking results. The origin is located at the camera's optical center (sensor), Z axis is pointing towards a user, Y axis is pointing up. The measurement units are meters for translation and degrees for rotation angles.

The computed 3D mask has coordinates that place it over the user's face (in the camera's coordinate frame) as shown in figure 6.8.

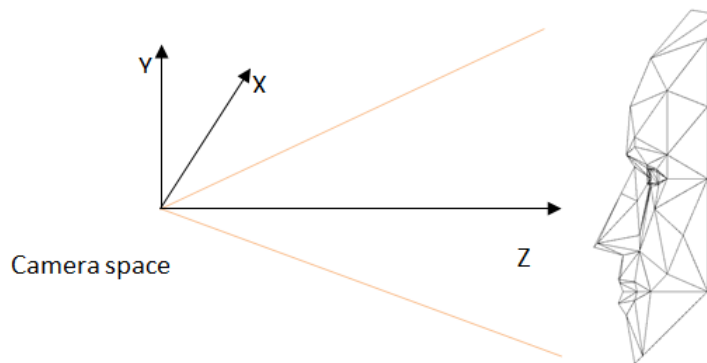


Figure 6.8: Coordinate system of the *Face Tracking SDK*.

Input images

The *Face Tracking SDK* accepts Kinect color and depth images as input. The tracking quality may be affected by the image quality of these input frames (that is, darker or fuzzier frames track worse than brighter or sharp frames). Also, larger or closer faces are tracked better than smaller faces.

IFTFaceTracker interface

The main interface is *IFTFaceTracker*, and its instance may be created by calling *FTCreateFaceTracker*. After initialization, it allows tracking a face synchronously by passing color and depth images as input (see *IFTImage* as part of *FT_SENSOR_DATA*); the results are returned via an *IFTResult* instance. It is assumed that both color and depth input images are coming from a Kinect sensor. There also is an initialization method that allows configuring the *Face Tracking SDK* for the system it is used in.

IFTFaceTracker provides a method *CreateFTResult* to create an instance of *IFTResult* for holding face tracking results specific for the model the instance of *IFTFaceTracker* is using. An application will need to create an instance of *IFTResult* before it can start tracking faces. An application can use the *IFTFaceTracker* method to get an array of potential head areas for the image data (*FT_SENSOR_DATA*) provided by the application. It is up to the developer to interpret the results and decide which faces to track.

To start tracking a face, the developer needs to call *StartTracking*. *StartTracking* is an expensive method that searches in the provided image for a face, determines its orientation and initiates the face tracking. It can also be provided a hint regarding where to look for a face or pass *NULL* to search the entire image – the first face found will be tracked. Another hint is the orientation of the head. The head

orientation may be derived from the Kinect skeleton data. Without providing a *hint* for the head orientation, the *Face Tracking SDK* will still try to track the face but initial results might be suboptimal.

Once *StartTracking* successfully started tracking a face as indicated by the returned *pFTResult*, an application should continue face tracking by subsequently calling *ContinueTracking*. *ContinueTracking* uses information from the previous calls of *StartTracking* or *ContinueTracking*. The developer should continue calling *ContinueTracking* until it is desired to stop face tracking or face tracking failed, for example, because the person whose face is tracked steps outside the camera frame. A failure of face tracking is indicated by a *pFTResult* status. To re-start face tracking, an application calls *StartTracking* again and subsequently calls *ContinueTracking*.

ContinueTracking is a relatively fast function that uses state information about a tracked face. It is much cheaper than *StartTracking*. In some rare occasions, the developer might call only *StartTracking* if the application frame rate is very low or if the face is moving very fast between frames (so that continuous tracking is not possible). For tracking multiple users, determine the faces desired to track, e.g. by instantiating *IFTFaceTracker* and calling *DetectFaces*. Creating additional instances of *IFTFaceTracker* for each extra face that it is desired to track should be done afterwards.

It is also possible to retrieve (*GetShapeUnits*) and set (*SetShapeUnits*) *Shape Units* (SUs). By providing SUs for known users, face tracking returns better results in the beginning as it does not have to “learn” the SUs for a tracked face. It takes about approximately 2 minutes to learn SUs in real-time for a given user. The current SUs can be retrieved from *IFTFaceTracker* (for example, for persisting the SU between runs of the application).

Typically, *IFTFaceTracker* keeps computing the SUs to improve them. If this is undesirable for an application, the developer can choose to not compute SUs by passing *False* to *SetShapeComputationState*. An application might want to not compute SUs if SUs for a user have been previously created by other tools other than *Face Tracking SDK* that are more suited for the application and it wants to save its computations cost. However, the developer can use *GetShapeComputationState* and *SetShapeComputationState* to fully control when to compute SUs.

IFTResult interface

The *IFTResult* interface provides access to the result from face tracking calls (*IFTFaceTracker.StartTracking*, *IFTFaceTracker.ContinueTracking*). *IFTResult* is created by calling *IFTFaceTracker.CreateFTResult*. *IFTFaceTracker* provides *CreateFTResult* as results are related to the underlying model that *IFTFaceTracker* has been initialized with.

GetStatus should be called to determine if face tracking is successful (in which case *S_OK* is returned). Upon a successful face tracking call, *IFTResult* provides access to the following information:

- **GetFaceRect:** The rectangle in video frame coordinates of the bounding box around the tracked face.
- **Get2DShapePoints:** 2D (X,Y) coordinates of the key points on the aligned face in video frame coordinates. It tracks the 87 2D points indicated in the figure 6.9 (as well as 13 others that aren't represented there).

Each time *StartTracking* or *ContinueTracking* is called, *FTResult* will be updated, which contains the following information about a tracked user: tracking status, 2D points, 3D head pose and AUs.

IFTModel interface

The *IFTModel* interface provides a way to convert tracking results to a mesh of 3D vertices in the camera space. Its instance is returned by *IFTFaceTracker::GetFaceModel* method. The interface provides several methods to get various model properties:

- **GetSUCount:** Returns number of shape units (SU) used in the 3D linear model
- **GetAUCount:** Returns number of animation units (AU) used in the 3D linear model.
- **GetTriangles:** Returns 3D model mesh triangles (indexes of vertices). Each triangle has 3 vertex indexes listed in the clockwise fashion.
- **GetVertexCount:** Returns number of vertices in the 3D model mesh.

Also, *IFTModel* provides two methods to get a 3D face model in either the video camera space or projected onto the video camera image plane. These methods are:

- **Get3DShape:** Returns the 3D face model vertices transformed by the passed *Shape Units*, *Animation Units*, scale stretch, rotation and translation.
- **GetProjectedShape:** Returns the 3D face model vertices transformed by the passed *Shape Units*, *Animation Units*, scale stretch, rotation and translation and projected to the video frame.

Face Tracking outputs: 2D Mesh and Points

The *Face Tracking SDK* tracks the 87 2D points indicated in the following figure (in addition to 13 points that aren't shown in it):

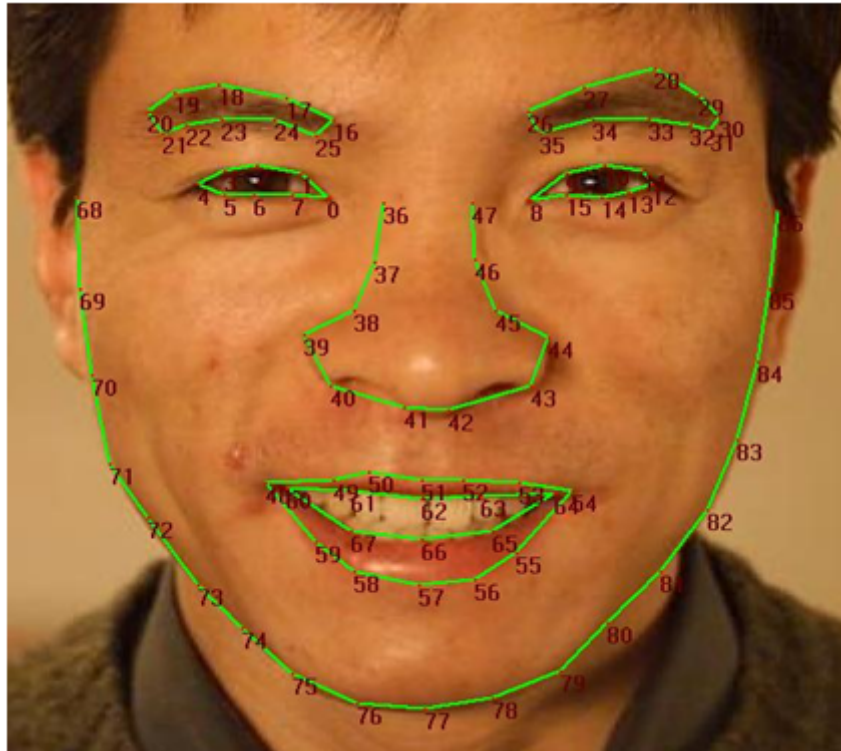


Figure 6.9: Tracked points on a successful face tracking call.

These points are returned in an array, and are defined in the coordinate space of the RGB image (in 640 x 480 resolution) returned from the Kinect sensor. The additional 13 points include: the center of the eye, the corners of the mouth, the center of the nose and a bounding box around the head.

Face Tracking outputs: 3D Head Pose

The X,Y and Z position of the user's head are reported based on a right-handed coordinate system (with the origin at the sensor, Z pointed towards the user and Y pointed UP – this is the same as the Kinect's skeleton coordinate frame). Translations are in meters and the user's head pose is captured by three angles: pitch, roll and yaw (See figure 6.10). The angles are expressed in degrees, with values ranging from -180 degrees to +180 degrees. In the table 6.1 we show a more in depth analysis on the angles.

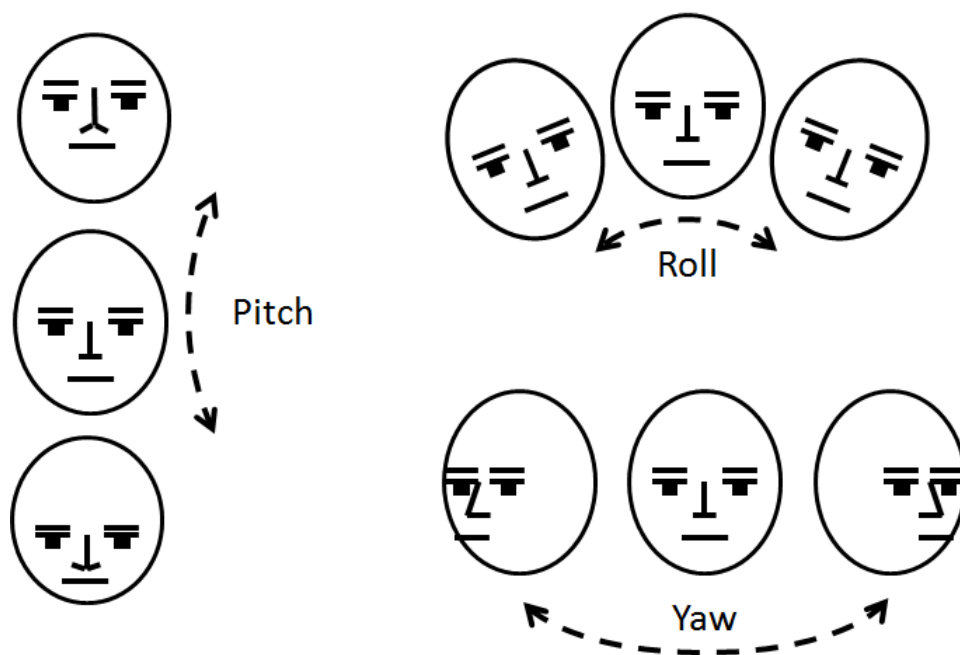


Figure 6.10: Head pose angles.

Angle	Value
Pitch angle	-90 = looking down towards the floor 0 = neutral +90 = looking up towards the ceiling
Roll angle	-90 = horizontal parallel with right shoulder of subject 0 = neutral +90 = horizontal parallel with left shoulder of the subject
Yaw angle	-90 = turned towards the right shoulder of the subject 0 = neutral +90 = turned towards the left shoulder of the subject

Table 6.1: Head pose angles analysis.

The SDK tracks when the head pitch is less than 20 degrees, the head roll is less than 90 degrees and the head yaw is less than 45 degrees, but works best when they are less than 10 degrees, 45 degrees and 45 degrees, respectively.

Face Tracking outputs: Animation Units

The *Face Tracking SDK* results are also expressed in terms of weights of six AUs and 11 SUs, which are a subset of what is defined in the *Candide-3 model*[1]. The SUs estimate the particular shape of the user's head: the neutral position of their mouth, brows, eyes and so on. The AUs are deltas from the neutral shape that can be used

to morph targets on animated avatar models so that the avatar acts as the tracked user does. The *Face Tracking SDK* tracks the AUs shown in table 6.2 (each AU is expressed as a numeric weight varying between -1 and +1).

Face Tracking outputs: Shape Units

The *Face Tracking SDK* tracks the following 11 SUs in *IFTFaceTracker*. They are discussed here because of their logical relation to the *Candide-3 model*. Each SU specifies the vertices it affects and the displacement (x,y,z) per affected vertex.

The SUs are head height (SU 0 in *Candide-3*), eyebrows vertical position (SU 1 in *Candide-3*), eyes vertical position (SU 2 in *Candide-3*), eyes width (SU 3 in *Candide-3*), eyes height (SU 4 in *Candide-3*), eye separation distance (SU 5 in *Candide-3*), nose vertical position (SU 8 in *Candide-3*), mouth vertical position (SU 10 in *Candide-3*), mouth width (SU 11 in *Candide-3*), eyes vertical difference and chin width. The last two are SUs not contemplated by *Candide-3* that this SDK supports. However, it does not support cheeks z (SU 6 in *Candide-3*), nose z-extension (SU 7 in *Candide-3*) and nose pointing up (SU 9 in *Candide-3*).

Face Tracking outputs: 3D Face Model

The *Face Tracking SDK* also tries to fit a 3D mask to the user's face and it is also based on the *Candide-3 model*. Even though it is not returned directly at each call to the SDK, it can be computed from the AUs and SUs with the function *Get3DShape* of the *IFTModel*.

Other data structures

The following are the most relevant data structures from this SDK that we have not explained in the previous lines but are used in our project:

- **FT_SENSOR_DATA**: Structure that contains all input data for a face tracking operation.
- **FT_CAMERA_CONFIG**: Structure that contains the configuration of the video or depth sensor which frames are being tracked.
- **FT_VECTOR3D**: Structure that contains the points of a 3D vector.


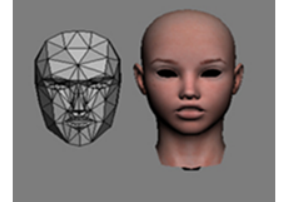





AU name and value	Avatar illustration	AU value interpretation
Neutral Face		<ul style="list-style-type: none"> All AUs are 0
AU0 – Upper Lip Raiser		<ul style="list-style-type: none"> 1 = Showing teeth fully 0 = Neutral, covering teeth -1 = Maximal possible pushed down lip
AU1 – Jaw Lowerer		<ul style="list-style-type: none"> 1 = Fully open 0 = Closed -1 = Closed, like 0
AU2 – Lip Stretcher		<ul style="list-style-type: none"> 1 = Fully stretched (Joker’s smile) 0 = Neutral -0.5 = Rounded (Pout) -1 = Fully rounded (Kissing mouth)
AU3 – Brow Lowerer		<ul style="list-style-type: none"> 1 = Fully lowered, to the limit of the eyes 0 = Neutral -1 = Raised almost all the way
AU4 – Lip Corner Depressor		<ul style="list-style-type: none"> 1 = Very sad frown 0 = Neutral -1 = Very happy smile
AU5 – Outer Brow Raiser		<ul style="list-style-type: none"> 1 = Raised as in an expression of deep surprise 0 = Neutral -1 = Fully lowered as a very sad face

Table 6.2: Anima4.0n Units analysis.

6.4.2 Extending the Kinect's interface

After knowing how the *Face Tracking SDK* works, we decided to extend the Kinect's interface and add its functionalities. First, we will explain how we initialize the proper data structures and the SDK, and then how we do the *Face Tracking*, treat its results and render the mask.

Initializing the Face Tracking

Like we did when we initialized Kinect and checking always that there has been no errors, we create our *IFTFaceTracker* interface with the *FTCreateFaceTracker* function and initialize it with its *Initialize* function and the configurations that we already set for the Kinect sensor (we make use of the functions *GetVideoConfiguration* and *GetDepthConfiguration* that check the Kinect's sensor configurations for each camera).

Then we create the *IFTResult* interface with the *CreateFTResult* function of the *IFTFaceTracker* and two more buffers of video and depth data with the *FTCreateImage* function. We decided to do this as to maintain separated the buffers that belong to the Kinect sensor and the buffers that belong to the *Face Tracking* (we will explain later on how we transfer the data). So, it is needed again to use the *Allocate* function on both buffers as to define their format, which is the same as Kinect's buffers. After this, we assign them to our *FT_SENSOR_DATA* structure that will hold all our data when giving it to the *IFTFaceTracker* later on.

Lastly, we set a lot of initial data about the skeletons as we then use the *NuiSkeletonTrackingEnable* function of the Kinect sensor to enable the skeleton tracking with some flags (*NUI_SKELETON_TRACKING_FLAG_ENABLE_IN_NEAR_RANGE* and *NUI_SKELETON_TRACKING_FLAG_ENABLE_SEATED_SUPPORT*), because we will use this skeletons data in some point to try and get a *hint* of where the user is and accelerate the tracking of the face.

```
1 bool Kinect::initFaceTrack() {
2     // Create an instance of a face tracker
3     pFT = FTCreateFaceTracker();
4     if (!pFT) return false;
5     // Video camera config with width, height, focal length in pixels
6     FT_CAMERA_CONFIG videoCameraConfig;
7     GetVideoConfiguration(&videoCameraConfig);
8
9     // Depth camera config with width, height, focal length in pixels
10    FT_CAMERA_CONFIG depthCameraConfig;
11    GetDepthConfiguration(&depthCameraConfig);
12
13    // Initialize the face tracker
14    HRESULT hr = pFT->Initialize(&videoCameraConfig, &depthCameraConfig,
```

```

15     NULL, NULL);
16     if (FAILED(hr)) return false;
17
18     // Create a face tracking result interface
19     pFTResult = NULL;
20     hr = pFT->CreateFTResult(&pFTResult);
21     if (FAILED(hr)) return false;
22
23     // Prepare image interfaces that hold RGB and depth data
24     pColorFrame = FTCreateImage();
25     pDepthFrame = FTCreateImage();
26     if (!pColorFrame || FAILED(hr = pColorFrame->Allocate(640, 480,
27     FTIMAGEFORMAT_UINT8_B8G8R8X8))) return false;
28     if (!pDepthFrame || FAILED(hr = pDepthFrame->Allocate(320, 240,
29     FTIMAGEFORMAT_UINT16_D13P3))) return false;
30
31     sensorData.pVideoFrame = pColorFrame;
32     sensorData.pDepthFrame = pDepthFrame;
33     sensorData.ZoomFactor = 1.0f; // Not used, must be 1.0
34     sensorData.ViewOffset = POINT{0, 0}; // Not used, must be (0,0)
35
36     isTracked = false;
37     SetCenterOfImage(NULL);
38
39     m_hint3D[0] = m_hint3D[1] = FT_VECTOR3D(0, 0, 0);
40
41     for (int i = 0; i < NUI_SKELETON_COUNT; ++i)
42     {
43         m_HeadPoint[i] = m_NeckPoint[i] = FT_VECTOR3D(0, 0, 0);
44         m_SkeletonTracked[i] = false;
45     }
46
47     DWORD dwSkeletonFlags =
48     NUI_SKELETON_TRACKING_FLAG_ENABLE_IN_NEAR_RANGE |
49     NUI_SKELETON_TRACKING_FLAG_ENABLE_SEATED_SUPPORT;
50     hr = sensor->NuiSkeletonTrackingEnable(NULL, dwSkeletonFlags);
51     if (FAILED(hr)) return false;
52
53     return true;
54 }

```

Listing 6.3: Initialization of the Face Tracking SDK.

Doing the Face Tracking

Finally, we were able to start tracking faces. This had to be done each frame, so we decided to implement it in the update function of the interface (See listing 6.4).

We start by keeping the updates of Kinect's video and depth buffers (*getKinectVideo* and *getKinectDepth*) and adding to it a new function that obtains the skeleton data

for the hint in a similar manner (*getSkeleton*). Then, if Kinect is present and its video buffer as well (*GetVideoBuffer*), we copy this buffer onto *Face Tracking*'s video buffer. We do the same with the depth buffers if Kinect's depth buffer is present (*GetDepthBuffer*). And we also compute the hint with the use of the function *GetClosestHint*, that analyzes the skeleton data received from the Kinect sensor.

Now we have our data available, so if we are not already tracking a face, we use the *StartTracking* function of our *IFTFaceTracker* interface; otherwise, we use the *ContinueTracking* function. Both functions receive the hint and our data through the *sensorData*, which is a *FT_SENSOR_DATA* structure that holds our video and depth data, and output their results into the *IFTResult* interface.

Afterwards, we check if the tracking has been successful. If we haven't detected a face, we finish and wait for the next frame. But if we have detected it, we obtain all of the output data: the SUs with the use of the *GetShapeUnits* function and the *IFTModel* with the *GetFaceModel* function. Moreover, if we obtain the *IFTModel*, we are able to obtain the head pose with the *Get3DPose* function and the AUs with the *GetAUCoefficients* function, which we will use later on to animate our 3D model. But for now, we are able to paint the mask (2D mesh points) on top of the image that holds the video buffer with the use of the *VisualizeFaceModel* function, which will let the user and us know when a face is being detected and how accurately.

```

1 void Kinect::update() {
2     getKinectVideo();
3     getKinectDepth();
4     getSkeleton();
5
6     HRESULT hrFT = E_FAIL;
7
8     if (kinect && GetVideoBuffer()) {
9         HRESULT hrCopy = m_VideoBuffer->CopyTo(pColorFrame, NULL, 0, 0);
10
11         if (SUCCEEDED(hrCopy) && GetDepthBuffer()) {
12             hrCopy = m_DepthBuffer->CopyTo(pDepthFrame, NULL, 0, 0);
13         }
14
15         if (SUCCEEDED(hrCopy)) {
16             FT_VECTOR3D* hint = NULL;
17             if (SUCCEEDED(GetClosestHint(m_hint3D)))
18             {
19                 hint = m_hint3D;
20             }
21
22             if (!isTracked) {
23                 hrFT = pFT->StartTracking(&sensorData, NULL, hint,
                pFTResult);
            }
        }
    }
}

```

```

24         }
25         else {
26             hrFT = pFT->ContinueTracking(&sensorData, hint,
pFTResult);
27         }
28     }
29 }
30
31 isTracked = SUCCEEDED(hrFT) && SUCCEEDED(pFTResult->GetStatus());
32 SetCenterOfImage(pFTResult);
33
34 // Do something with pFTResult.
35 if (isTracked) {
36     BOOL suConverged;
37     pFT->GetShapeUnits(NULL, &pSU, &numSU, &suConverged);
38     POINT viewOffset = { 0, 0 };
39     FT_CAMERA_CONFIG cameraConfig;
40     GetVideoConfiguration(&cameraConfig);
41
42     IFTModel* ftModel;
43     HRESULT hr = pFT->GetFaceModel(&ftModel);
44     if (SUCCEEDED(hr))
45     {
46         // Register the results
47         HRESULT hrRes = pFTResult->Get3DPose(&scale, rotation,
translation);
48         if (!SUCCEEDED(hrRes)) std::cout << "Couldn't get the 3D pose
of the Face Model" << std::endl;
49
50         hrRes = pFTResult->GetAUCoefficients(&pAU, &numAU);
51         if (!SUCCEEDED(hrRes)) std::cout << "Couldn't get the
Animation Units of the Face Model" << std::endl;
52
53         hr = VisualizeFaceModel(pColorFrame, ftModel, &cameraConfig,
pSU, 1.0, viewOffset, pFTResult, 0x00FFFF00);
54
55         ftModel->Release();
56     } else {
57         std::cout << "Could not get the Face Model" << std::endl;
58     }
59
60     if (!SUCCEEDED(hr)) {
61         std::cout << "Could not draw the Face Model" << std::endl;
62     }
63 }
64 }

```

Listing 6.4: Initialization of the *Face Tracking* SDK.

Rendering the tracked mask

As we explained in the lines before, in the case that we detected a face, we overwrote our video buffer with the 2D mesh points of the tracked face as a mask. Therefore, as we modify a data structure that is identical to the one that is being used to render in our window, there is no need to do further changes to the rendering pipeline of the interface except changing which buffer is rendered, that is, changing it for the *Face Tracking's* video buffer. Nevertheless, the figure 6.11 shows the result of implementing this interface when a face is detected.

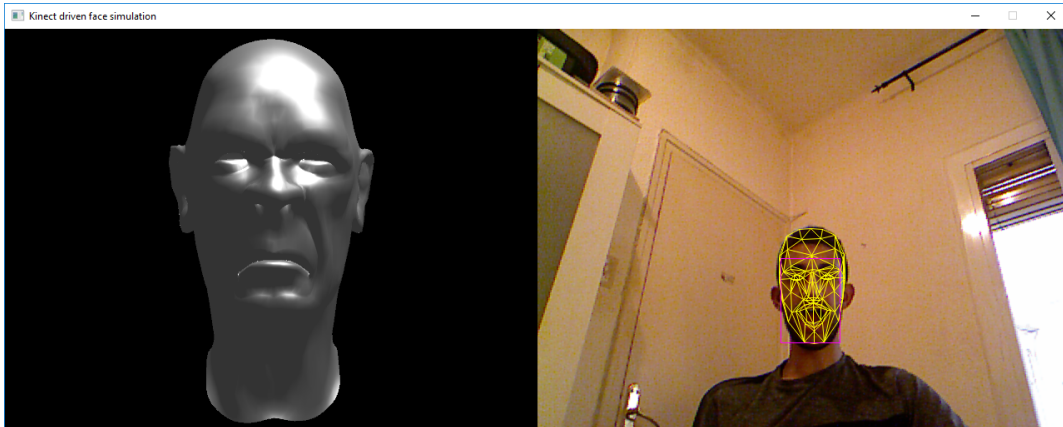


Figure 6.11: Mask rendered on top of Kinect's image when a face is detected.

6.5 Meeting the first obstacle

It was at this point that we faced our first obstacle. We had Kinect tracking the user's face and all the data regarding the *Animation Units* and *3D head pose* ready to use and animate our 3D model. However, we did not have an actual way of animating the 3D model, as it was just statically being rendered in our window. It all came down to the fact that we needed a pipeline that would allow us apply the AUs and head poses that we are tracking to the 3D model and do it in a way that the 3D model deforms its skin correctly and still maintain some coherence with real circumstances. For example, we shouldn't allow for the 3D model to rotate entirely, but instead apply some weight to the neck and make it stay in its position, as to maintain some realism.

After trying to come up with ideas to approach this problem, we decided that we should make use of what the FBX format offers us. In the 3.3 *Use of previous works* section, we explained that implementing a FBX pipeline of loading and rendering was the most difficult in comparison to what we had already implemented with the

OBJ format, but reaching such an obstacle in the development, it was the obvious decision. This implied some minor changes on the temporal planning, which will be detailed in the *7.1 Changes to the initial project* section.

In order to explain how we overcame this obstacle, we will detail the *FBX SDK* in the following lines and afterwards we will gather the main ideas that we extracted from our research on said SDK in a way that it will be easier to understand how we implemented the FBX helper and modified the model's interface, which is explained in the *6.6 FBX helper* section.

6.5.1 FBX SDK

The *Autodesk FBX SDK* is a free, easy-to-use, C++ software development platform and API toolkit that allows application and content vendors to transfer existing content into and from the FBX format with minimal effort. In the following lines we will describe some of the most important interfaces and classes that we use in our project, and also some of the samples that made it possible for us to understand this SDK.

FbxManager

SDK object manager. It is in charge of scene element allocation, scene element deallocation and scene element search and access. Upon destruction, all objects allocated by the SDK manager and not explicitly destroyed are destroyed as well.

FbxScene

Description of a 3D scene. It contains the nodes (including the root node), materials, textures, videos, gobos (filters placed over spots light to project light patterns through fog on a surface), poses, characters, character poses, control set plugs, generic nodes, scene information, global settings and a global evaluator. The nodes are structured in a tree under the scene's root node.

When an object is created using the *FBX SDK*, a scene is usually passed as argument to the object creation function to specify that the object belongs to this scene. At this point, a connection is made with the object as source and the scene as destination.

All objects in the scene can be queried by connection index. In addition, generic nodes, materials, and textures can also be queried by name. In this latter case, the first object with the queried name will be returned.

FbxDocument

Base class for the *FbxScene* class. It is a collection of objects, called the root member objects. This is because these objects each form the root of an object graph. The manager *FbxManager* has access to all documents, scenes and libraries. It can be contained in another document, thus, a hierarchy of documents can be built. The root of all documents is simply called the root document. And a document manages animation stacks and provides access to animation stack information.

FbxNode

Represents an element in the scene graph. The tree management services are self contained in this class. Besides the tree management, this class defines all the properties required to describe the position of the object in the scene. This information include the basic translation, rotation and scaling properties and the more advanced options for pivots, limits and IK joints attributes such the stiffness and dampening.

When it is first created, the *FbxNode* object is "empty" (i.e: it is an object without any graphical representation that only contains the position information). In this state, it can be used to represent parents in the node tree structure but not much more. The normal use of this type of objects is to add them an attribute that will specialize the node.

The node attribute is an object in itself and is connected to the the *FbxNode*. This also means that the same node attribute can be shared among multiple nodes. *FbxCamera*, *FbxLight*, *FbxMesh*, etc... are all node attributes and they all derive from the base class *FbxNodeAttribute*.

FbxMesh

A mesh is a geometry made of polygons. The class can define a geometry with as many n-sided polygons as needed. Users can freely mix triangles, quadrilaterals and other polygons. Since the mesh-related terminology of the *FBX SDK* differs a little from the known standards, here are their definitions:

- A control point is an XYZ coordinate, it is synonym of vertex.
- A polygon vertex is an index to a control point (the same control point can be referenced by multiple polygon vertices).
- A polygon is a group of polygon vertices. The minimum valid number of polygon vertices to define a polygon is 3.

FbxCluster

A cluster, or link, is an entity acting on a geometry. More precisely, the cluster acts on a subset of the geometry's control points. For each control point that the cluster acts on, the intensity of the cluster's action is modulated by a weight. The link mode specifies how the weights are taken into account. The cluster's link node specifies the node (*FbxNode*) that influences the control points of the cluster. If the node is animated, the control points will move accordingly. Usually, a cluster is part of a skin.

FbxAnimLayer

The animation layer is a collection of animation curve nodes. Its purpose is to store a variable number of *FbxAnimCurveNode*. The class provides different states flags, an animatable weight and the blending mode flag to indicate how the data on this layer is interacting with the data of the other layers during the evaluation.

FbxPose

This class contains the description of a Pose and provide some methods to access pose info in one FBX scene. The *FbxPose* object can be setup to hold *Bind Pose* data or *Rest Pose* data.

The *Bind Pose* holds the transformation (translation, rotation and scaling) matrix of all the nodes implied in a link deformation. This includes the geometry being deformed, the links deforming the geometry and recursively all the ancestors nodes of the link. The *Bind Pose* gives the transformation of the nodes at the moment of the binding operation when no deformation occurs.

The *Rest Pose* is a snapshot of a node transformation. A *Rest Pose* can be used to store the position of every node of a character at a certain point in time. This pose can then be used as a reference position for animation tasks, like editing walk cycles.

One difference between the two modes is in the validation performed before adding an item and the kind of matrix stored. In *Bind Pose* mode, the matrix is assumed to be defined in the global space, while in *Rest Pose* the type of the matrix may be specified by the caller, so local system matrices can be used. Actually, because there is one such flag for each entry (*FbxPoseInfo*), it is possible to have mixed types in a *FbxPose* element. It is therefore the responsibility of the caller to check for the type of the retrieved matrix and to do the appropriate conversions if required.

FbxTime

Class to encapsulate time units. *FbxTime* can measure time in hour, minute, second, frame, field, residual and also combination of these units. It is recommended to use

FbxTime for all time related operations. For example, currently it is used in all curve filters and all animation-related classes. *FbxTime* is just used to represent a moment; to represent a period of time, *FbxTimeSpan* should be used.

FbxBlendShape

A blend shape deformer takes a base shape (polygonal surface, curve, or surface) and blends it with other target shapes based on weight values. Blend shape deformer organize all target shapes via blend shape channel. One blend shape deformer can contain multiple blend shape channels, then each channel can organize multiple target shapes.

The blend effect of each blend shape channel is additive, so the final blend effect of a blend shape deformer is the sum of blend effect of all blend shape channels it contains, the blend effect of each blend shape channel is controlled by its property *DeformPercent*.

FbxMatrix

Basic 4x4 double matrix class. It has a lot of functionalities that makes its use advantageous.

FbxAMatrix

Affine matrix class. Matrices are defined using the *Column Major* scheme. When a *FbxAMatrix* represents a transformation (translation, rotation and scale), the last row of the matrix represents the translation part of the transformation. It also has great functionalities.

FbxVector4

A four double mathematic vector class.

Samples

This SDK offers a few samples that helps understand it better. The most relevant are the following:

- **ImportScene:** This example illustrates how to detect if a scene is password protected, import and browse the scene to access node and animation information. It displays the content of the FBX file which name is passed as program argument.
- **ViewScene:** This example illustrates how to display the content of a FBX or a OBJ file in a graphical window. This program is based on the OpenGL Utility Toolkit (GLUT). A menu is provided to select the current camera and the current animation stack.

6.5.2 Main ideas of the FBX SDK

Out of everything that we read on the *FBX SDK*, the samples were the ones that helped us understand it. In the following lines we will describe in what way each sample helped us and how much of it we will use in our project. It is necessary to note that at this point we had already exported our 3D model from *Autodesk Maya 2016* in the FBX format, as we wanted to also test it with the samples.

ImportScene

This sample taught us how to load and close a FBX file: initializing the SDK objects, such as the *FbxManager* and the *FbxScene*, and destroying the SDK objects when we have finished our work with them. Moreover, upon execution, this sample prints all kinds of information regarding the FBX file it loads. This is how we learnt the actual hierarchy of our 3D model and how it is composed.

Analyzing the thousands of lines of information, we discovered that the 3D model had already defined clusters of vertices, each one with its respective weight, throughout the different meshes of the whole model. It also had a *BlendShape* with several channels that imitated facial gestures such as a happy or shocked face and poses that defined the poses of the meshes. These pieces of information were key to understanding the next sample.

ViewScene

Now this sample had a way more complicated implementation. It loaded a FBX file responsively, that is, if it had animations it would let the user play them and, if it didn't, it would just render the model. Looking through its code we learnt how this pipeline worked: for a selected pose, a selected animation layer and a time, the pipeline would recursively navigate through the nodes of the scene analyzing the pose of each mesh for that moment in time and compute the skin and shape deformations with complicated algorithms that applied several deformations to specific clusters of vertices, adding up through the whole navigation and finally obtaining a resulting set of vertices that defined the model for that moment in time in the animation and rendering it.

It is important to note that, for testing purposes, we created some animations in *Autodesk Maya 2016* that made us able to understand how this animation layers worked: basically, for a moment in time, the animation layer defines a weight for the *BlendShape* channels that affect the model during the animation and, if there are, modifications to the poses of the meshes.

All of these facts made us realize that these *BlendShape* channels worked like the AUs from the *Face Tracking* and that we would be able to make use of this pipeline.

Instead of incrementing the time and playing the whole animation, we could always compute the resulting model for a fixed time in the animation, such as the first frame in the animation, and also instead of looking for the *BlendShape* weights in the animation layer, we could define these weights ourselves each time with the AUs data that we receive from Kinect. Moreover, the meshes' poses are defined by what *FbxPose* we select when using the pipeline. And from that *FbxPose* the deformations are computed, so if we modify the head's mesh pose with the angles we receive from Kinect before computing the resulting model, we will be able to also apply the angles without any further work.

6.6 FBX helper

With enough insight on the *FBX SDK*, in the following lines we will explain how we created the *FBX helper*, which is a collection of functions that implement the FBX pipeline that our project needs, how we applied it to our model's interface and improved its rendering, and also we will show a few tests on the pipeline as to see that it works properly.

6.6.1 Creation of the helper

As we stated in the 6.5.2 *Main ideas of the FBX SDK*, we understood that we could make use of the pipeline in the *ViewScene* sample and the loading and closing of the FBX file functions in the *ImportScene* sample. Therefore, we gathered all the functions necessary into this helper, which can be seen in the listing 6.5. We will describe them in the following lines but not in that much detail as they are quite large and technical.

We have at first the *InitializeSdkObjects*, which initializes the *FbxManager* and the *FbxScene* and let us make use of the SDK; the *LoadScene* function, which loads the FBX file into our scene; the *DestroySdkObjects* that should be used when exiting the application or if at some point the manager and scene are not used anymore, destroying all the objects; and the *PreparePointCacheData* function, which prepares the data in case there are vertex cache deformers.

Then we have some functions that are used along the pipeline: *GetGlobalPosition*, which gets the global position for a node, i.e. the global position of a mesh; *GetPoseMatrix*, which gets the matrix of a node in a specific pose; *GetGeometry*, which gets the geometry offset to a node (it is never inherited by the children); ; *ReadVertexCacheData*, which reads the vertex cache data of a mesh; *MatrixScale*,

which scales all the elements of a matrix with the passed value; *MatrixAddToDiagonal*, which adds a value to all the elements in the diagonal of the matrix; and *MatrixAdd*, which sums two matrices element by element.

The pipeline starts with the *DrawNodeRecursive* function, which navigates through the nodes in the scene recursively and calls the *DrawNode* function for a node and for all its children. Then, the *DrawNode* checks which type of node are we working with and we encounter a mesh, it calls the *DrawMesh* function, which calls the *ReadVertexCacheData* if the mesh has a vertex cache deformer, the *ComputeShapeDeformation* if it has a shape and *ComputeSkinDeformation* if it has skins associated with it. All of these calls to deform the mesh populate a temporal vertex array, which at the end has its vertices translated into global space by multiplying them with the *pGlobalPosition* matrix and then copied to our vertex array that holds the resulting deformed model. Both *ComputeShapeDeformation* and *ComputeSkinDeformation* make several calls to the functions *ComputeLinearDeformation*, *ComputeClusterDeformation* and *ComputeDualQuaternionDeformation*, which are algorithms that compute our desired deformations. However, it is in the *ComputeShapeDeformation* function where we make use of our own weights in the *BlendShape* channels, by means of calling the *evaluateChannel* function.

Finally, we have our *evaluateChannel* function that basically checks which channel is being consulted and translates our AUs values into its weight, and the *DisplayHierarchy* function that prints the hierarchy of the FBX file in the console for debugging purposes.

```
1 #pragma once
2
3 #include "Globals.hpp"
4 #include "Model.hpp"
5
6 void InitializeSdkObjects(FbxManager*& pManager, FbxScene*& pScene);
7 void DestroySdkObjects(FbxManager* pManager, bool pExitStatus);
8
9 bool LoadScene(FbxManager* pManager, FbxDocument* pScene, const char*
  pFilename);
10 void DisplayHierarchy(FbxNode* pNode, int pDepth);
11
12 FbxAMatrix GetGlobalPosition(FbxNode* pNode, const FbxTime& pTime,
  FbxPose* pPose = NULL, FbxAMatrix* pParentGlobalPosition = NULL);
13 FbxAMatrix GetPoseMatrix(FbxPose* pPose, int pNodeIndex);
14 FbxAMatrix GetGeometry(FbxNode* pNode);
15 void PreparePointCacheData(FbxScene* pScene, FbxTime &pCache_Start,
  FbxTime &pCache_Stop);
16 void ReadVertexCacheData(FbxMesh* pMesh, FbxTime& pTime, FbxVector4*
  pVertexArray);
```



```

17
18 void DrawNodeRecursive(FbxNode* pNode, FbxTime& pTime, FbxAnimLayer*
    pAnimLayer, FbxAMatrix& pParentGlobalPosition, FbxPose* pPose,
    std::vector<glm::vec3>* vertices, std::vector<double>* weights);
19 void DrawNode(FbxNode* pNode, FbxTime& pTime, FbxAnimLayer* pAnimLayer,
    FbxAMatrix& pParentGlobalPosition, FbxAMatrix& pGlobalPosition,
    FbxPose* pPose, std::vector<glm::vec3>* vertices,
    std::vector<double>* weights);
20 void DrawMesh(FbxNode* pNode, FbxTime& pTime, FbxAnimLayer* pAnimLayer,
    FbxAMatrix& pGlobalPosition, FbxPose* pPose, std::vector<glm::vec3>*
    vertices, std::vector<double>* weights);
21
22 void ComputeShapeDeformation(FbxMesh* pMesh, FbxTime& pTime,
    FbxAnimLayer* pAnimLayer, FbxVector4* pVertexArray,
    std::vector<double>* weights);
23 void ComputeSkinDeformation(FbxAMatrix& pGlobalPosition, FbxMesh* pMesh,
    FbxTime& pTime, FbxVector4* pVertexArray, FbxPose* pPose);
24 void ComputeLinearDeformation(FbxAMatrix& pGlobalPosition, FbxMesh*
    pMesh, FbxTime& pTime, FbxVector4* pVertexArray, FbxPose* pPose);
25 void ComputeClusterDeformation(FbxAMatrix& pGlobalPosition, FbxMesh*
    pMesh, FbxCluster* pCluster, FbxAMatrix& pVertexTransformMatrix,
    FbxTime pTime, FbxPose* pPose);
26 void ComputeDualQuaternionDeformation(FbxAMatrix& pGlobalPosition,
    FbxMesh* pMesh, FbxTime& pTime, FbxVector4* pVertexArray, FbxPose*
    pPose);
27
28 void MatrixScale(FbxAMatrix& pMatrix, double pValue);
29 void MatrixAddToDiagonal(FbxAMatrix& pMatrix, double pValue);
30 void MatrixAdd(FbxAMatrix& pDstMatrix, FbxAMatrix& pSrcMatrix);
31
32 double evaluateChannel(std::vector<double>* weights, FbxString name);

```

Listing 6.5: Header of the *FBX helper*.

6.6.2 Modifying the model's interface

At this point in time, we finally had the pipeline implemented, so we just needed to add it to the model's interface. Basically, we changed the loader into using the *FBX helper*, as seen in the listing 6.6, and then the updating and rendering.

We start by initializing the *FbxManager* and *FbxScene*, so we then load into the scene out FBX file. We do an initial loading of the model without using the pipeline as to set correctly *OpenGL's* data structures with the use of the *getFBXData* function, which navigates through the scene recursively taking the necessary information about vertices, normals and materials.

Then we triangulate all of the geometry with the use of a *FbxGeometryConverter*, that allows the conversion of geometry, and its *Triangulate* function. We also get the animation layers in the scene, prepare the point cache data with the *PreparePointCacheData* function and set our current animation layer and current time, which we will use in the pipeline to access the *Blendshape* channels properly.

Finally, we set some structures that will hold our AUs information (*shocked*, *happy*, *jawLow*, *upperLip*, *lipStr* and *outBrow*), the pose that we will use in the pipeline, we call the *setDefaultPose* function, which we will explain in the *6.7 Connecting both interfaces and the second obstacle* section, and we print the hierarchy with the *DisplayHierarchy* function if we are debugging.

```

1 bool Model::loadFBX(std::string filename) {
2     bool lResult = true;
3
4     // Prepare the FBX SDK.
5     InitializeSdkObjects(lSdkManager, lScene);
6
7     // Load the scene
8     FbxString lFilePath(filename.c_str());
9     lResult = LoadScene(lSdkManager, lScene, lFilePath.Buffer());
10
11    // Load the model
12    FbxNode* lNode = lScene->GetRootNode();
13    this->getFBXData(lNode);
14
15    // Convert mesh, NURBS and patch into triangle mesh
16    FbxGeometryConverter lGeomConverter(lSdkManager);
17    lGeomConverter.Triangulate(lScene, /*replace*/ true);
18
19    FbxArray<FbxString*> mAnimStackNameArray;
20    lScene->FillAnimStackNameArray(mAnimStackNameArray);
21
22    // Prepare the Point Cache data
23    mCache_Start = FBXSDK_TIME_INFINITE;
24    mCache_Stop = FBXSDK_TIME_MINUS_INFINITE;
25    PreparePointCacheData(lScene, mCache_Start, mCache_Stop);
26
27    // Initialize the frame period.
28    mFrameTime.SetTime(0, 0, 0, 1, 0,
29    lScene->GetGlobalSettings().GetTimeMode());
30
31    int pIndex = 0;
32
33    FbxAnimStack * lCurrentAnimationStack =
34    lScene->FindMember<FbxAnimStack>(mAnimStackNameArray[pIndex]->Buffer());
35    mCurrentAnimLayer = lCurrentAnimationStack->GetMember<FbxAnimLayer>();
36    lScene->SetCurrentAnimationStack(lCurrentAnimationStack);

```

```

35
36     FbxTakeInfo* lCurrentTakeInfo =
37     lScene->GetTakeInfo (* (mAnimStackNameArray [pIndex] ));
38     if (lCurrentTakeInfo)
39     {
40         mStart = lCurrentTakeInfo->mLocalTimeSpan . GetStart ();
41         mStop = lCurrentTakeInfo->mLocalTimeSpan . GetStop ();
42     }
43     else
44     {
45         // Take the time line value
46         FbxTimeSpan lTimeLineTimeSpan ;
47
48         lScene->GetGlobalSettings () . GetTimelineDefaultTimeSpan (lTimeLineTimeSpan) ;
49
50         mStart = lTimeLineTimeSpan . GetStart ();
51         mStop = lTimeLineTimeSpan . GetStop ();
52     }
53
54     // check for smallest start with cache start
55     if (mCache_Start < mStart)
56         mStart = mCache_Start ;
57
58     // check for biggest stop with cache stop
59     if (mCache_Stop > mStop)
60         mStop = mCache_Stop ;
61
62     mCurrentTime = mStart ;
63
64     shocked = happy = jawLow = upperLip = lipStr = outBrow = 0 ;
65     newResult = false ;
66
67     // Get the pose to work with
68     lPose = lScene->GetPose (0) ;
69
70     setDefaultPose () ;
71
72     // Display hierarchy
73     if (DEBUG_MODE) {
74         FBXSDK_printf (" \n \n ----- \n Hierarchy \n ----- \n \n " );
75         DisplayHierarchy (lNode , 0) ;
76     }
77
78     return lResult ;
79 }

```

Listing 6.6: FBX loader of the model's interface.

After having the *FBX SDK* working, we set to use the pipeline at each frame, so we modified the *update* function of the interface, as seen in the listing 6.7. We

just clear our *vertices* array, set the *weights* array with the weights (even though right now we are not setting any), call the *DrawNodeRecursive* function and finally rebind the *vertices*'s data into our *OpenGL* buffer.

```

1 void Model::update() {
2     FbxAMatrix IDummyGlobalPosition;
3
4     // We will transform here the AUs into weights later on
5
6     // Reset the vertices array
7     vertices.clear();
8
9     // Set our own weights
10    std::vector<double> weights;
11    weights.push_back(shocked);
12    weights.push_back(happy);
13    weights.push_back(jawLow);
14    weights.push_back(lipStr);
15    weights.push_back(upperLip);
16    weights.push_back(outBrow);
17
18    // Obtain the new deformed model
19    DrawNodeRecursive(IScene->GetRootNode(), mCurrentTime,
20    mCurrentAnimLayer, IDummyGlobalPosition, IPose, getVerticesArray(),
21    &weights);
22
23    glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer_triangles);
24    glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(glm::vec3),
25    &vertices[0], GL_STATIC_DRAW);
26 }

```

Listing 6.7: Update function of the model's interface.

Lastly, we modified the rendering by changing the light's position and its intensity and doing some minor changes to the *OpenGL* shaders (See figure 6.12).

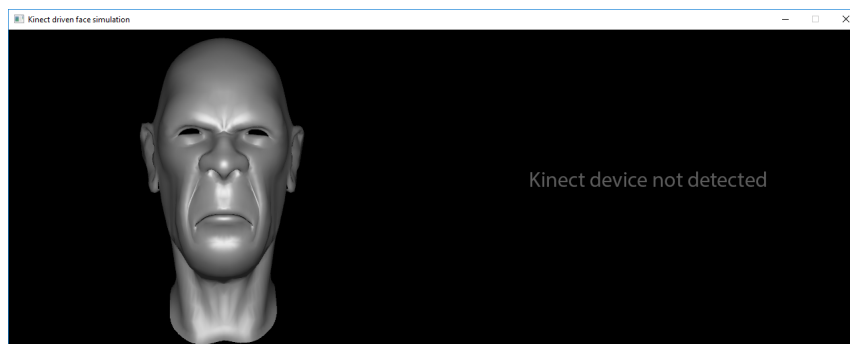


Figure 6.12: Improved rendering with the *FBX SDK*.

6.6.3 Testing the rendering and some deformations

Also, just for testing purposes, we manually applied weights to certain *BlendShape* channels to see the outcome. In the figure 6.13 we apply full weight to the *happy* channel and it is visible that the deformation works correctly. On top of it, we also applied full weight to the *shocked* channel and both deformations applied and added themselves properly, as seen in the figure 6.14.

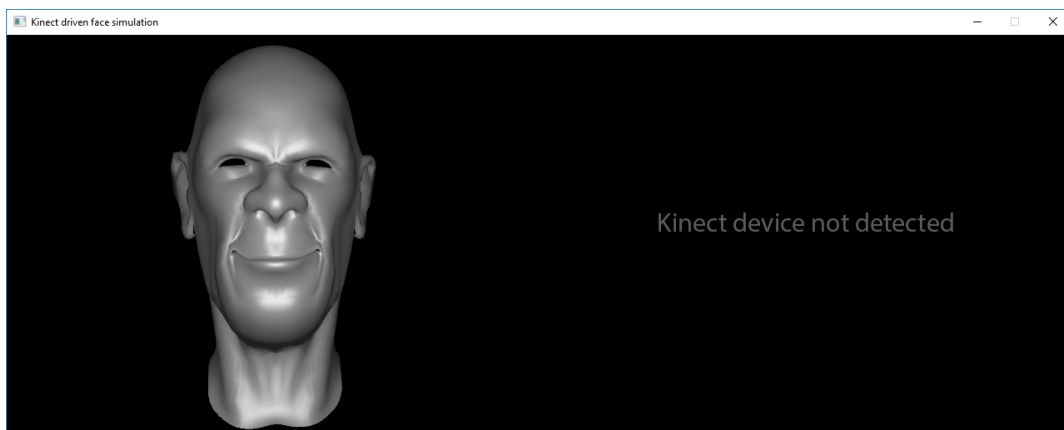


Figure 6.13: Applying full weight to the happy *BlendShape* channel.

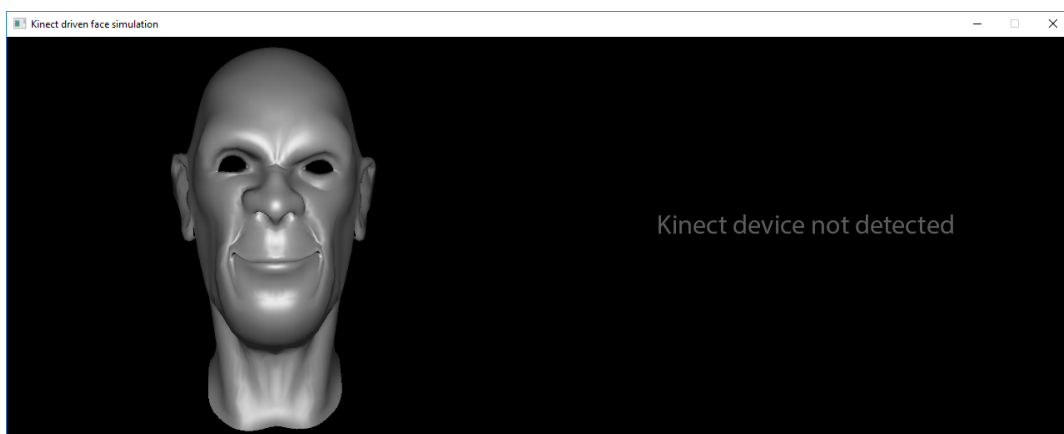


Figure 6.14: Applying full weight to both the happy and shocked *BlendShape* channels.

6.7 Connecting both interfaces and the second obstacle

Having the Kinect's and the model's interfaces already implemented, there was only one step left, which was connecting both properly. For this purpose, we will

explain in the following lines how we transferred the data from one interface into the other, how we interpreted it in the model's interface and how we encountered our second obstacle.

Transfer of data

Whenever we are tracking a face, we obtain in the Kinect's interface all of the results necessary, that is, the AUs, SUs and the head pose. Therefore, the transfer was just a matter of checking at each frame in our 3D viewer's *main* function (after updating the Kinect interface) if there was a tracked face. If we are, we obtained all of these results and posted them into our model's interface; if not, we told the model's interface to stop using any results data, stop the animation and we just continued with the execution. After doing this check, we updated the model's interface, so it was up to the model's interface to interpret these results.

Data interpretation

Once we had the results available in our model's interface, we converted them into values that would work with our implementation. We started by adding the head pose's data onto our model, which meant that we had to modify the pose that we were using. That's why we implemented the functions *SetDefaultPose*, which made a copy of the original matrices of the *head's* mesh in the pose into our own matrices, as seen in the listing 6.8; and the *modifyHead*, which modified the pose of the head mesh matrix with the incoming data (we applied it to the second matrix because of how the model is built).

```
1 void Model::setDefaultPose() {
2     // Index of the head matrices are 0 and 3, because of how the model
3     // is built
4     FbxNode * node;
5     FbxMatrix matrixJaw;
6     IPose->SetIsBindPose(false);
7     int index = 0;
8     matrixHeadO = IPose->GetMatrix(index);
9     index = 3;
10    matrixHead = IPose->GetMatrix(index);
11    lRotation[0] = lRotation[1] = lRotation[2] = 0.0f;
12 }
```

Listing 6.8: Function that saves the original state of the matrices in the pose.

```
1 void Model::modifyHead(FbxVector4 T, FbxVector4 R, FbxVector4 S) {
2     FbxNode * node;
3     FbxMatrix matrix;
4     FbxVector4 pTranslation, pRotation, pShearing, pScaling;
5     double pSign;
6     int index;
```

```

7  bool isLocalMatrix = false;
8
9  // Translate the head and rotate the head
10 index = IPose->Find("head");
11 node = IPose->GetNode(index);
12 IPose->Remove(index);
13 IPose->Add(node, matrixHeadO, isLocalMatrix);
14
15 index = IPose->Find("head");
16 node = IPose->GetNode(index);
17 matrix = IPose->GetMatrix(index);
18 isLocalMatrix = IPose->IsLocalMatrix(index);
19
20 matrixHead.GetElements(pTranslation, pRotation, pShearing, pScaling,
21 pSign);
22 matrix.SetTRS(pTranslation + T, pRotation + R, pScaling*S);
23 IPose->Remove(index);
24 IPose->Add(node, matrix, isLocalMatrix);
25 }

```

Listing 6.9: Function that modifies the pose's matrices with the new head's pose.

Then, as seen in the listing 6.10, we extended the *update* function to perform these modifications on the head with a call to the *modifyHead* function (with a slight change of reference frames) if there were new results, and if not, we just reestablished the head to its original state. Also, if there were new results we did the transformation of the AUs into weights with some adjustments (at that point we only worked with the *happy* (AU4) and *shocked* (AU3) channels).

```

1 void Model::update() {
2     FbxAMatrix IDummyGlobalPosition;
3
4     if (newResult) {
5         modifyHead(FbxVector4(0, 0, 0, 0),
6 FbxVector4((double)-IRotation[1], (double)IRotation[0],
7 (double)IRotation[2], 1), FbxVector4(1, 1, 1, 0));
8
9         if (IAU != NULL) {
10            shocked = -(IAU[3] * 100) * 4 + 100;
11            happy = -(IAU[4] * 100) * 3 + 20;
12        }
13        newResult = false;
14    } else if (stopAnim) {
15        modifyHead(FbxVector4(0, 0, 0, 0), FbxVector4(0, 0, 0, 1),
16 FbxVector4(1, 1, 1, 0));
17        shocked = 50;
18        stopAnim = false;
19    }
20 }

```

```

18 // Reset the vertices array
19 vertices.clear();
20
21 // Set our own weights
22 std::vector<double> weights;
23 weights.push_back(shocked);
24 weights.push_back(happy);
25 weights.push_back(jawLow);
26 weights.push_back(lipStr);
27 weights.push_back(upperLip);
28 weights.push_back(outBrow);
29
30 // Obtain the new deformed model
31 DrawNodeRecursive(1Scene->GetRootNode(), mCurrentTime,
32 mCurrentAnimLayer, IDummyGlobalPosition, IPose, getVerticesArray(),
33 &weights);
34 glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer_triangles);
35 glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(glm::vec3),
36 &vertices[0], GL_STATIC_DRAW);
37 }

```

Listing 6.10: Extended *update* function of the model's interface.

Second obstacle

When tested the work done, we found out that, even though our transferring of data and interpretation was being done correctly, there was something wrong with the 3D model, as seen in the figure 6.15.

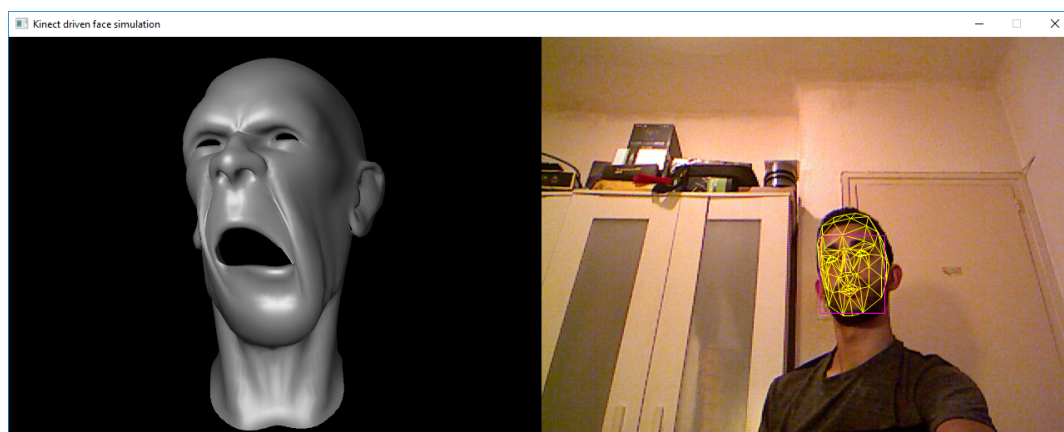


Figure 6.15: Jaw not moving properly with the head.

After investigating the reason behind this, we came to the conclusion that the pose that we were using was a *Bind Pose*, so the *jaw* and *jawEnd* meshes were not receiving the transformation on the head mesh because they weren't local matrices.

6.8 Workaround

We decided to modify the pose directly into a *Rest Pose* in our *SetDefaultPose* function and build their matrices again. As seen in the listing 6.11, we set the pose as *Rest Pose* and computed the local matrices for the *jaw* and *jawEnd* with the inverse matrices of their parents.

```
1 void Model::setDefaultPose() {
2     // Index of the head matrices are 0 and 3, because of how the model
3     // is built
4     FbxNode * node;
5     FbxMatrix matrixJaw, matrixJawEnd;
6     IPose->SetIsBindPose(false);
7     int index = 0;
8     matrixHeadO = IPose->GetMatrix(index);
9     index = 3;
10    matrixHead = IPose->GetMatrix(index);
11
12    // In the case of the others, we just look for their indices
13    // in the pose and rebuild the model with local matrices
14    matrixJaw = IPose->GetMatrix(IPose->Find("jaw"));
15    matrixJawEnd = IPose->GetMatrix(IPose->Find("jawEnd"));
16
17    matrixJawEnd = matrixJaw.Inverse() * matrixJawEnd;
18    index = IPose->Find("jawEnd");
19    node = IPose->GetNode(index);
20    IPose->Remove(index);
21    IPose->Add(node, matrixJawEnd, true);
22
23    matrixJaw = matrixHead.Inverse() * matrixJaw;
24    index = IPose->Find("jaw");
25    node = IPose->GetNode(index);
26    IPose->Remove(index);
27    IPose->Add(node, matrixJaw, true);
28
29    lRotation[0] = lRotation[1] = lRotation[2] = 0.0f;
30 }
```

Listing 6.11: Extended *SetDefaultPose* function of the model's interface.

After we did this, it worked as it was intended to, as shown in the figure 6.16. But there were still some glitches on how the *Face Tracking* is not optimal, which we will explain in the 7.2 *Development analysis* section.

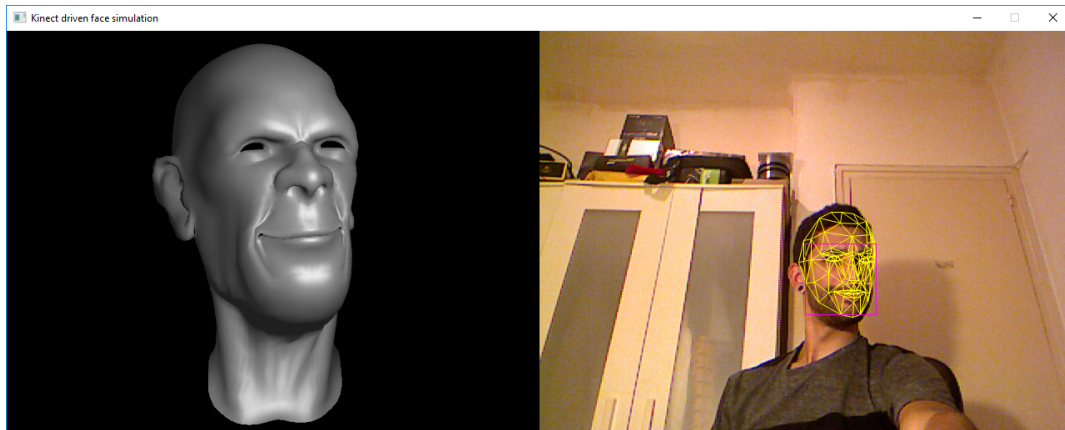


Figure 6.16: Correct head movement and animations.

6.9 Tweaking the animations

As the last part of the project, we tried to tweak the adjustments on the channels that we had already transformed into weights in the *update* function. We also tried to add some more AUs, the upper lip raiser (AU0) and the jaw lowerer (AU1), with good and bad results, as seen in figure 6.17.

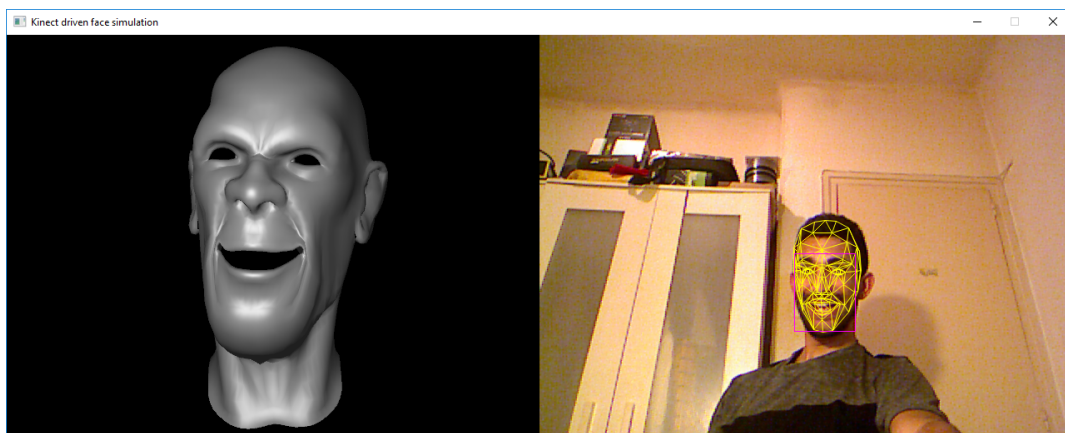


Figure 6.17: Implemented more AUs.

We couldn't get better adjustments and the limitations of the Kinect sensor started to appear more frequently, not detecting correctly the face and turning the animation into a poorly one. Nevertheless, the final results are quite satisfying.

7. Analysis and conclusion

This section aims to compare this project's initial milestone with its final milestone, which implies analyzing all the changes made to the project throughout its development, making an analysis on the actual development of the project and its possible extensions, commenting the level of achievement of the technical competences and finishing with a conclusion.

7.1 Changes to the initial project

Taking into consideration what we have achieved in the development of the project, there has been no relevant changes to the scope and context, as we have met all of our main objectives except the extension. However, as it was stated on the *6.5 Meeting the first obstacle* section, we actually came to a point in the development of the project where we had to change our temporal planning.

In order to finish the project by the end of June, we decided not to do the final extension to the video conference application and to implement the *FBX helper* between the time we encountered the obstacle and until three weeks before the project's completion date. Then there were left one week and a half to connect Kinect's data to the model and another week and a half to do the final documentation.

And as we were able to finish on time, this new work plan ensured that there were no additional costs, so there was no need to modify the original budget. Therefore, the sustainability stayed the same as well.

7.2 Development analysis

We were able to finish the project in time, but at the cost of some quality in it. We will analyze in the following lines some aspects of our project's development.

Our 3D viewer is quite simple, didn't give any delays in the development and does its work, but it could still be upgraded into a more user-friendly design, with functionalities that may appeal the users. Apart from that, the 3D model gave us some problems, but with the decision of changing into the FBX format, everything changed. The *FBX helper* supposed a boost in the quality of this project, but still needs some refinement, such as making a complete rendering pipeline that works on GPU and not in CPU and supporting other types of data that the FBX format offers (like *NURBS* or patches).

On the other side, the main lose in quality was the Kinect sensor. We could clearly see moments where a slight change on the light of a room could make the tracked face's results be mostly wrong and how the low frame rate and resolutions of the camera made impossible to track fast movements or even track accurately.

The *Face Tracking* could be greatly improved if instead of letting Kinect compute the SUs, we could compute them ourselves, because it normally takes up to two minutes for Kinect to compute them correctly. Also, a leap on quality would be changing the device for its upgraded version, the *Kinect Sensor for Xbox One*, but that would jeopardize the objective of this project of making it expenseless.

Even though we faced obstacles and limitations on hardware and software and we couldn't do the final extension, the results of the project are satisfactory with what could be achieved with this kind of equipment.

7.3 Technical competences

Here we will describe how the technical competences were achieved in our project.

- **CCO1.1:** *Avaluar la complexitat computacional d'un problema, conèixer estratègies algorísmiques que puguin dur a la seva resolució, i recomanar, desenvolupar i implementar la que garanteixi el millor rendiment d'acord amb els requisits establerts. [Bastant]*

We were trying to achieve some kind of realism in the facial expressions, so we were in the need of doing almost everything that this technical competence

entails. We analyzed our problem, came up with strategies to solve it in an efficient way and, finally, decided which one was the best. But as we had a tight schedule, we were not able to achieve or explore the finest way of implementing the facial expressions, so we could say that we achieved this technical competence with the second highest level of achievement.

- **CCO2.6:** *Dissenyar i implementar aplicacions gràfiques, de realitat virtual, de realitat augmentada i videojocs. [En profunditat]*

In this project we have designed and implemented a graphic application in the form of a 3D viewer, which is this technical competence in its essence. And we have carried it out completely or, in other words, in depth.

- **CCO3.1:** *Implementar codi crític seguint criteris de temps d'execució, eficiència i seguretat. [En profunditat]*

We worked on a real time environment, so all of the code implemented had to be efficient and secure enough as to not drop frames. Because of the fact that we did take into account the efficiency of all the code that we implemented, this technical competence has been achieved in depth.

7.4 Conclusion

There has been success and failures throughout the whole project, making us act accordingly to assess the situations and overcome the obstacles. There are also many things that could be improved and extensions that would make it scale in quality. Nevertheless, it is safe to say that Kinect and its variety of SDKs are powerful tools that allow anyone with low income do facial animations with decent results, saving up in money but losing in quality.

Finally, this project's code implementation is fully available as a repository in the GitHub website[11].

Bibliography

- [1] Jörgen Ahlberg. *CANDIDE - a parameterized face*. URL: <http://www.icg.isy.liu.se/candide/> (visited on June 20, 2016).
- [2] Autodesk. *FBX SDK documentation*. URL: <http://help.autodesk.com/view/FBX/2016/ENU/> (visited on June 16, 2016).
- [3] Aware. *Nexa | Face™*. URL: <http://www.aware.com/biometrics/nexa-face/> (visited on June 16, 2016).
- [4] Ashish Sharma, Mukesh Agarwal, Anima Sharma, Pankhuri Dhuria. *Motion capture process, techniques and applications*. Vol. 1. 4. International Journal on Recent, Innovation Trends in Computing, and Communication (IJRITCC), Apr. 2013, pp. 251–257. URL: https://paginas.fe.up.pt/~prodei/dsie12/papers/paper_7.pdf (visited on Mar. 1, 2016).
- [5] *To see or not to see: A study comparing four-way avatar, video, and audio conferencing for work*. ACM Conference on Supporting Groupwork, Oct. 2012, pp. 31–34. URL: <http://research.microsoft.com/apps/pubs/default.aspx?id=178842> (visited on Mar. 2, 2016).
- [6] *libQGLViewer*. URL: <http://libqglviewer.com/> (visited on Feb. 29, 2016).
- [7] Luxand. *FaceSDK*. URL: <http://www.luxand.com/facesdk/> (visited on June 16, 2016).
- [8] Microsoft. *Face Tracking SDK*. URL: <https://msdn.microsoft.com/en-us/library/jj130970.aspx> (visited on Feb. 27, 2016).
- [9] Microsoft. *Kinect for Windows*. URL: <http://www.microsoft.com/en-us/kinectforwindows/> (visited on Feb. 27, 2016).
- [10] Microsoft. *Kinect for Windows SDK*. URL: <https://msdn.microsoft.com/en-us/library/hh855347.aspx> (visited on Feb. 27, 2016).
- [11] Guillermo Ojeda Noda. *Kinect driven facial animation repository*. URL: <https://github.com/Kv0the/Kinect-face-animation> (visited on June 21, 2016).

- [12] Pedro Nogueira. *Motion Capture Fundamentals*. Nov. 2011. URL: https://paginas.fe.up.pt/~prodei/dsie12/papers/paper_7.pdf (visited on Mar. 1, 2016).
- [13] Daniel Otero, Guillermo Ojeda. *keep running N nobody gets hurt*. URL: <https://github.com/d-otero/keep-running-n-nobody-gets-hurt> (visited on Feb. 29, 2016).
- [14] *Overleaf*. URL: <https://www.overleaf.com/> (visited on Mar. 11, 2016).
- [15] VCL. *Face Recognition Algorithms*. URL: <http://www.face-rec.org/algorithms/> (visited on June 16, 2016).
- [16] Wikipedia. *FBX*. URL: <https://en.wikipedia.org/wiki/FBX> (visited on June 16, 2016).
- [17] Wikipedia. *Wavefront .obj file*. URL: https://en.wikipedia.org/wiki/Wavefront_.obj_file (visited on June 16, 2016).