HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

COMPUTER SCIENCE

Universität Zürich[UZH]

Zentrum für Zahnmedizin

Jaw Viewer:

Displaying jaw movement for medical diagnosis

Bachelor Thesis

Department of Computer Science
University of Applied Science Rapperswil

Spring Term 2017

Authors:	Konrad Höpli, Roberto Cuervo Alvarez
Advisor:	Prof. Oliver Augenstein
Project Partner:	Clinic of Masticatory Disorders from the Center of Dental Medicine of the Zürich University
External Co-Examiner:	Reto Bättig
Internal Co-Examiner:	Prof. Peter Sommerlad

Table of Contents

Acknowledgements

We would like to thank specially to Prof. Augenstein for his support during the whole bachelor thesis. It was a luxury for us having him as advisor.

Also we thank the Clinic of Masticatory Disorders for its kindness and availability. For the help from the Assistants Thomas Corbat and Lukas Kretschmar we are grateful too.

1 Abstract

1.1 Introduction

The Clinic of Masticatory Disorders offers treatment for facial pain and mandibular joint problems. For better diagnosis, the clinic has developed a proprietary 3D camera system (Optis) to record the patient's mastication movement. They are also able to extract bones from an MRI image of a patient and store them as Stereo Lithography (STL) Files. In another proprietary software called TMJViewer, these resources can be merged to display the movement of the teeth and bones in a 3D animation, which is used for medical analysis.

The current process to achieve this goal requires multiple systems, time consuming manual input, and is therefore error-prone. In addition, the current solution does not allow to display movement data in real time, which complicates the diagnosis process, because immediate feedback to the patient is not possible.

The goal of this thesis is to develop a single, OpenGL based application that simplifies the above process and further allows real time data analysis.

1.2 Approach

To get familiar with OpenGL as well as various aspects of graphics rendering, we consumed some literature on the topic and walked through a tutorial that implemented Phong shading in OpenGL. We then used this code as a base for our project and extended its functionality according to requirements and thus obtained a working C++ prototype for our project. For that we imported anatomical objects (STL files) and transferred them to OpenGL as vertex buffer objects.

In a second step we were then using movement information from Optis to calculate rigid transformations which were afterwards applied to vertex data within a vertex shader. Finally we refactored the prototype before integrating it into a new WindowsForms application providing a graphical user interface for its configuration.

Due to an incomplete specification of the Optis calibration process, the displayed movement ended up not being completely accurate and a lot of reverse engineering was required to reach the resulting version of our software.

1.3 Results

This project resulted in an application consisting of a library component for the graphical display embedded into a minimalistic GUI for configuration. As the functionality of the existing application is not yet fully implemented, the software is not suited for productive use. However, it serves as a decent base for a replacement of the existing application and can be used - in combination with our documentation - as a solid introduction to OpenGL.

2 Terms of Reference

Echtzeitanalyse von Kaufunktionsstörungen mit Hilfe von Open GL

Studiengang:	Informatik (I)
Semester:	FS 2017 (20.02.2017-17.09.2017)
Durchführung:	Bachelorarbeit, Studienarbeit
Fachrichtung:	Software
Institut:	Diverses
Gruppengrösse:	2 Studierende
Status:	zugewiesen
Verantwortlicher:	Augenstein, Oliver
Betreuer:	Augenstein, Oliver
Gegenleser:	Sommerlad, Peter
Experte:	Reto Bättig, m&f engineering
Industriepartner:	Uni Zürich, Zentrum für Zahnmedizin

Ausschreibung:

Ausgangslage:

Das Zentrum für Zahnmedizin an der Universität Zürich befasst sich unter anderem mit der Analyse von Kaufunktionsstörungen.

Für diese Analyse wird heute vom Kiefer eines Patienten zunächst eine MRI-Aufnahme angefertigt, aus der danach die Kieferknochen und das Gebiss extrahiert und als Stereo-Lithographie-Dateien (im .stl-Format) abgespeichert werden.

Im Anschluss daran wird das Kauverhalten des Patienten mit einer 3d-Kamera aufgenommen.

Sobald alle Daten vorhanden sind, können die MRI-Aufnahmen mit der 3d-Kameraaufzeichnung so verschmolzen werden, dass der Kauvorgang des Patienten realistisch nachverfolgt werden kann.

Dieser Vorgang ist heute nicht in Echtzeit möglich, weshalb Fehler im 3d-Aufnahmeprozess erst im Nachhinein erkannt werden.

Ziel der Arbeit

Nach einer gründlichen Einarbeitung in Open GL, soll eine Anwendung entwickelt werden, die beim Start die .stl-Dateien der MRI-Aufnahmen des

Patienten einliesst und diese Daten mit den Echtzeitinformationen der 3d-Kameras so verknüpft, dass die Kieferbewegungen des Patienten in Echtzeit beobachtet werden können.

Nach Abschluss dieses Aufgabenteils soll die Anwendung um Filter ergänzt werden, die gewisse Aspekte der Kieferbewegung besonders hervorheben.

Weitere Anforderungen

Bei der Spezifikation der Funktionalität ist eine enge Zusammenarbeit mit dem Zentrum für Zahnmedizin der Universität Zürich notwendig.

Die Anwendung soll dabei so entwickelt werden, dass die Funktionalität der Anwendung auch als eigenständiges Paket in die bereits bestehende Applikation integriert werden.

Voraussetzungen: Interesse am Einarbeiten in Open GL (z.B. Kapitel 1-3 auf der Website https://learnopengl.com/)

gute Vektorgeometriekenntnisse

Bewerbungen:

Gruppe:	CUERVO ALVAREZ/Höpli ✉
Einschreibung:	Bachelorarbeit
Status:	Arbeit zugewiesen (Priorität Student: 1)
Studierende:	CUERVO ALVAREZ, Roberto Höpli, Konrad
Kommentar:	Lieber Oliver, Wie bereits besprochen, anbei unsere Bewerbung für die BA. Vielen Dank! Liebe Grüsse Roberto und Konrad

3 Management Summary

3.1 Roles and Responsabilities

Advisor: Prof. Oliver Augenstein, Professor of Mathematics

Project Developer: Roberto Cuervo, HSR Computer Sciences Student

Project Developer: Konrad Höpli, HSR Computer Sciences Student

3.2 Start Position

The Clinic for Masticatory Disorders of the University of Zurich employs a mixture of proprietary software (TMJViewer), hardware (3D Cameras System, Optis) and components made with other commercial software (MatLab, among others) in order to display the jaw bones of a patient combined with its recorded movement in a 3D animation for medical analysis. Working with all these different applications/components, different file formats and various manual inputs is time consuming, error-prone and does not allow real-time interaction with the patient.

Due to the lack of feedback in real-time, it takes an extended period of time until the doctors can see and analyse the recordings of a patient. In the case of a mistake by the patient him-/herself during the recording or in the processing of the data afterwards, another appointment might need to be scheduled essentially starting the process all over again and resulting in an unpleasant experience for both parties involved.

As a result, the Clinic desires an application with the goal of eventually unifying all the steps involved and adding the ability to support real-time display and therewith immediate feedback for the patient. While this represents the basic idea behind this project, the resulting software of it was not intended to be fully implemented within the given timeframe.

3.3 Approach

In the very beginning of the project and even before our first meeting with the involved people of the Clinic for Masticatory Disorders, we had to familiarize ourselves with the OpenGL technology that was going to be used for the graphical components of this project. We did so by consuming literature on the topic and then developing a prototype based on a suited tutorial recommended by Prof. Augenstein.

After these first steps into the complex domain, we scheduled a meeting at the Clinic in order to get to know all the people involved, but also the tools and technology already in use. Due to the complexity of the solution in place and various proprietary components and file formats, we quickly realized that we would not be able to determine what is feasible over the course of this project off the bat and thus an agile approach would be best suited for this project.

We then set the first goal for our interim presentation to get a basic application running that would display the bones of a patient with the recorded motion using a set of demonstration data provided. After reaching this first goal, we identified the next most desirable features with all the people and ended up deriving from the initial focus on the real-time capability since the progress was much more sluggish than expected. The root causes for this form of

progress was both caused by the lack of experience on the side of the developers, but also the lack of documentation and clear information on the side of the Clinic.

3.4 Results

The results of this thesis consist of:

- An application which meets most of the essential requirements, but is not sufficient for productive use.
- Documentation covering an introduction into OpenGL, the development of this project as well as the mathematical foundations used therein

The resulting application ended up much more basic than initially hoped and planned. However, the derivation from the initially defined goals for the developed application was deemed necessary due to the underlying complexity of the domain and absence of good information as well as documentation on the components in use.

3.5 Conclusion

The results of this project are not suitable for a productive environment, but both the documentation and the software created can serve as a very good foundation for future developments in this area. The graphical part of the application was designed to be a standalone component of which individual features can be extracted with ease and the documentation does not just cover the course of this project, but also contain viable information that the students would have liked to have when they started out.

4 Analysis

4.1 Introduction

The Clinic of Masticatory Disorders of the Zurich University treats among other facial pain and mandibular joint problems. For better diagnosis the Clinic has developed an own 3D camera system, Optis to record the mastication movement of the patient. The 3D cameras record the position of 2 triangles of 3 LEDs each, triangles fixed to the patient's mandibular.

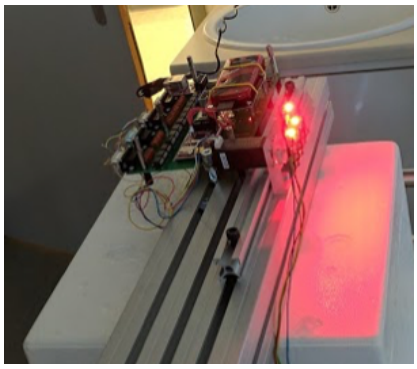


Figure 1: LED triangle example of the Optis system

One triangle is fix (static) in the upper jaw, while the second moves with the mandible. This movement is recorded by Optis in form of coordinates and saved to Motion Movement files (MVM) files, also a proprietary file extension.

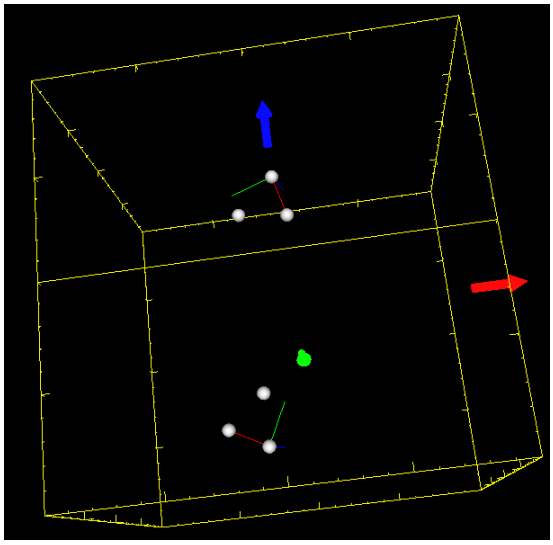


Figure 2: LEDs triangles in Optis application

The jaw movement is only one piece of the system. The another piece are the MRI images of the patient. These are stored in form of Stereo Lithography Files (STL) files (or Anatomy Objects). Together with the MVM files, another proprietary software of the Clinic, TMJViewer, can display the movement applied to the patient's anatomy. So the doctors can elaborate a diagnosis.

But the process required to achieve this goal is boring, tedious and above all, error-prone. All required files come from different sources and must be processed with different applications before using them with TeamViewer.

In the course of this project an application capable of importing STL files, movement data in form of MVM files or a stream, and displaying all of them with animation should be developed. The application should be able to display the movement in real-time or offline modus. The user should be able to choose which elements can be animated.

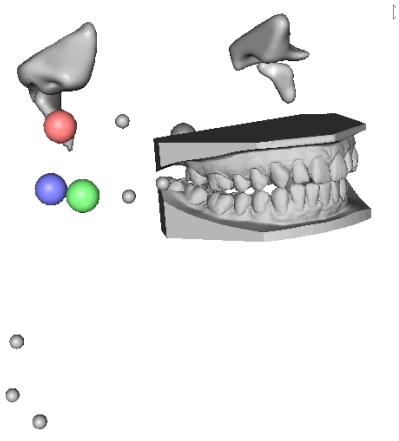


Figure 3: TMJViewer

4.2 Requirements Specification

4.2.1 General Description

There are two possible application scenarios:

- Offline Modus
- Real-time Modus

Offline Modus: The user has to enter manually all the data, STL and MVM files and other parameters needed by the application and configure it. The application can replay the movement any number of times.

Real-Time Modus: The user has to enter only the STL and the network parameters needed to connect the application with the 3D cameras via socket. The application can record the real-time movement and replay it.[1]

4.2.2 Functional Requirements

ID	Name	Description	Priority
FR1	Load anatomy	The application imports one or more graphic anatomy objects	★★★
FR2	Load movement	The application imports objects containing information about movement	★★★
FR3	Configuration	The application supports movements configuration	★★★
FR4	Display anatomy	The application shows the anatomy elements statically	★★★
FR5	Display movement	The application shows the movement of one or more anatomy elements	★★★
FR6	Real-time movement	The application shows the movement of one or more anatomy elements in real time	★★★
FR7	Record movement	The application is able to record movement at real-time	★★★
FR8	Select anatomy	The application supports selection of one or more anatomy elements	★★
FR9	Select movement	The application the selection of movement objects	★★
FR10	Show trajectories	The application shows the trajectory of a selected moving anatomy element	★
FR11	Color anatomy	The application supports coloring of one or more anatomy elements	★

Table 1: Functional Requirements

[1] Due to missing information from the project partner it was not possible to develop the Real-Time Modus. Explanations follow.

4.2.3 Non Functional Requirements

Technology OpenGL shall be used for graphics processing

GUI The application's GUI accomplishes modern standards

Frame Rate The must display the graphics with at least 24 FPS TODO: ask Konrad about this.

Platform The application shall run only in Windows, Windows 7 or above

Security

- **Confidentiality** The application shall not make the patient data available to unauthorized individuals or entities
- **Availability** The application must facilitate the delivery of patient's data from the doctor to the patient

Reliability

- **Recoverability** After a system crash or abnormal system end the application shall start again without problem.

Usability

- **Learnability** A new user should need maximal 10 minutes to learn how to operate with the application.
- **Operability** The application shall be a desktop application only.

Installability The installation of the application should not take more than 2 minutes.

Durability All user input in the GUI must be validated. Wrong input must be warned, in order to give the user the opportunity of correcting it.

4.2.4 Use Cases

Use Cases Diagram

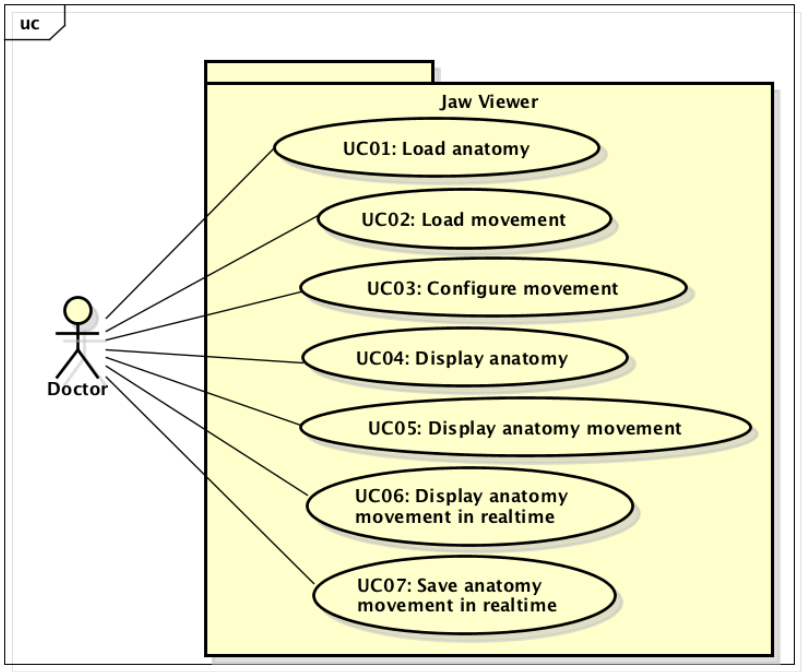


Figure 4: Use Cases

4.2.4.1 Actors

Name	Description
Doctor	An *Doctor* is a member of the medical personal of the Clinic of Masticatory Disorders who uses the *Jaw Viewer*
Technical Assistant	An *Technical Assistant* is a non-medical member of the personal of the Clinic of Masticatory Disorders who uses the *Jaw Viewer*

Table 2: Actors

4.2.4.2 UC01: Load anatomy

Mapped Requirement	4.2.2 FR1, FR8
Primary Actor	Doctor
Story	The *Doctor* starts *Jaw Viewer*. The configuration window is shown. The Doctor searches and selects the desired Anatomy Objects or STL files and load them. The files are validated. If they are right they will be displayed as loaded in the configuration window.

Table 3: UCO1: Load anatomy

4.2.4.3 UC02: Load movement

Mapped Requirement	4.2.2 FR2, FR9
Primary Actor	Doctor
Story	After 4.2.4.2 UC01, the configuration window is shown. The Doctor searches and selects the desired movement files or MVM files and load them. The files are validated. If they are right they will be displayed as loaded in the configuration window.

Table 4: UCO2: Load movement

4.2.4.4 UC03: Configure movement

Mapped Requirement	4.2.2 FR3, FR8, FR9
Primary Actor	Doctor
Story	After 4.2.4.2 UC01 and 4.2.4.3 the configuration window is shown to the Doctor. The Doctor can select one or more STL files and mark them as animated or stationary. The doctor must enter Reference Sphere and Calibration Data parameters. The Doctor can optionally enter network configuration parameters.

Table 5: UC03: Configure movement

4.2.4.5 UC04: Display anatomy

Mapped Requirement	4.2.2 FR4
Primary Actor	Doctor
Story	After 4.2.4.4 UC03, the Doctor starts the visualization and the Anatomy Objects are displayed. As only static STL files were selected, no movement is displayed.

Table 6: UC04: Display anatomy

4.2.4.6 UC05: Display anatomy movement

Mapped Requirement	4.2.2 FR5
Primary Actor	Doctor
Story	After 4.2.4.5 UC04, the Doctor starts the visualization and the Anatomy Objects are displayed. As both static and animated STL files were selected, both static and animated Anatomy Objects are displayed.

Table 7: UC05: Display anatomy movement

4.2.4.7 UC06: Display anatomy movement in real-time

Mapped Requirement	4.2.2 FR6
Primary Actor	Doctor
Story	After 4.2.4.5 UC04, the Doctor starts the visualization and the Anatomy Objects are displayed. As both static and animated STL files were selected and the network configuration was entered, both static and animated Anatomy Objects are displayed.

Table 8: UC06: Display anatomy movement in real-time

4.2.4.8 UC07: Save anatomy movement in real-time

Mapped Requirement	4.2.2 FR7
Primary Actor	Doctor
Story	During 4.2.4.7 UC06, the Doctor can save the current movement.

Table 9: UC07: Save anatomy movement in real-time

4.2.5 Implemented Use Cases

ID	Implemented	Use Case	Remark
UC01	Yes	Load anatomy	
UC02	Yes	Load movement	
UC03	Yes / No	Configure movement	Time constraints
UC04	Yes	Display anatomy	
UC05	Yes / No	Display anatomy movement	Time constraints and deficient information from the client
UC06	No	Display anatomy movement in real-time	Client changed functional requirements during development
UC07	No	Save anatomy movement in real-time	Client changed functional requirements during development

Table 10: Implemented Use Cases

4.2.6 Architecture

The *Jaw Viewer* is mainly a desktop based application with possibility of communication with a server providing the movement data stream. Although this second requirement was described in the terms of reference, the client decided during the project to discard it.

4.2.6.1 Deployment Diagram

The Jaw Viewer is the entry point for the user and displays the graphics with help of OpenGL and a C # graphical interface.

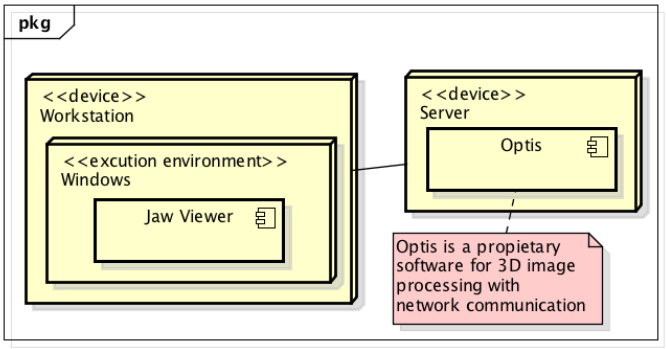


Figure 5: Deployment Diagram

4.2.6.2 Component Diagram

The Jaw Viewer is built with three components:

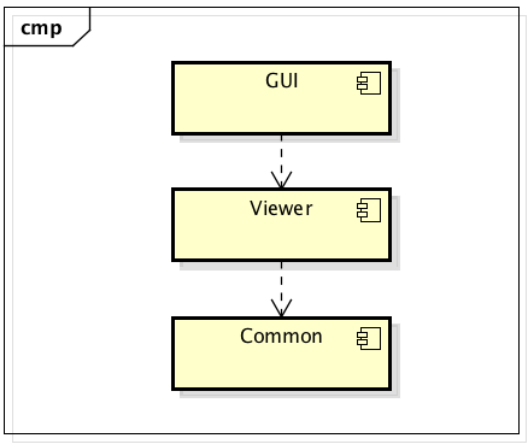


Figure 6: Component Diagram

4.2.6.3 Domain Model

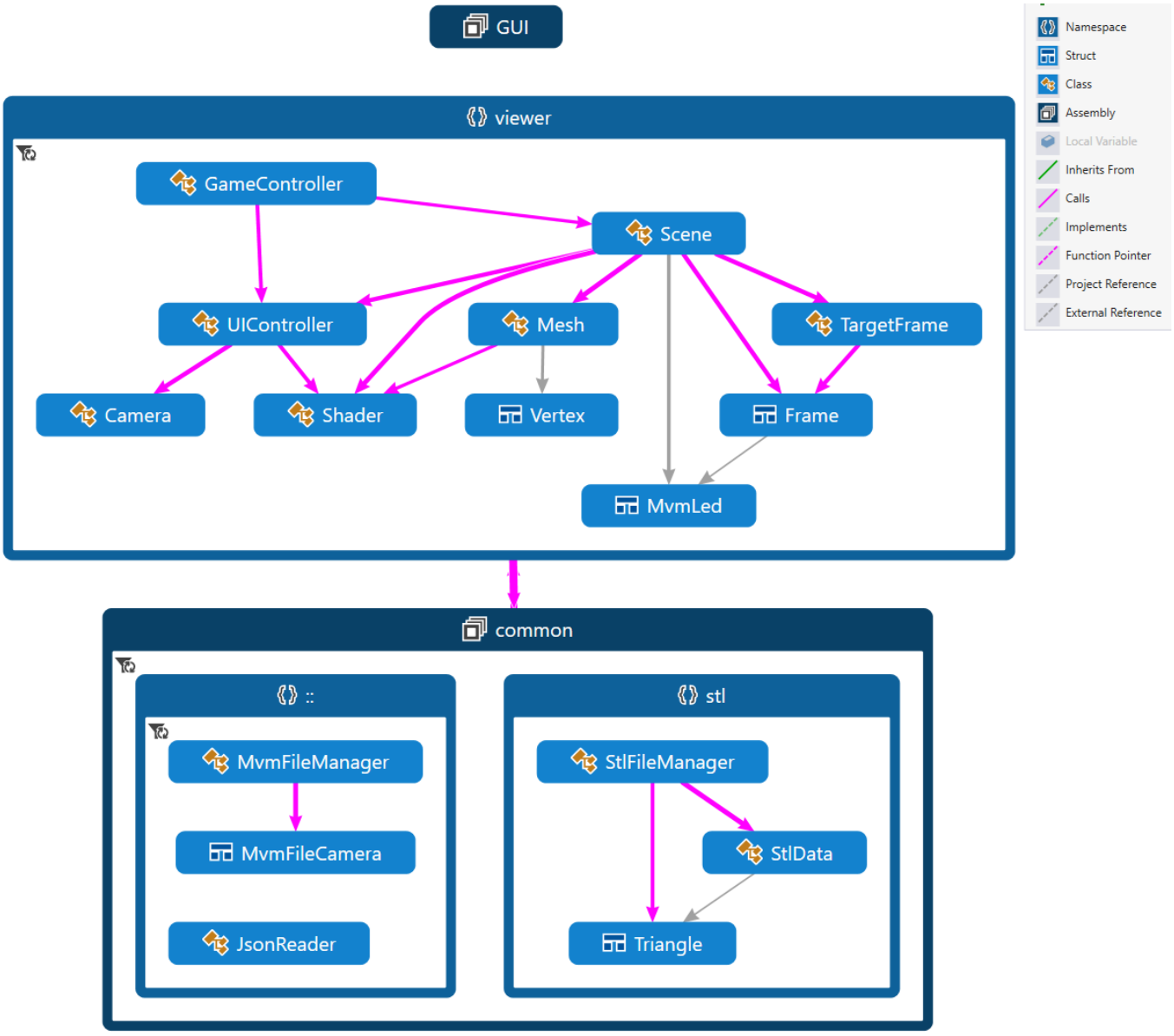


Figure 7: Domain Model

The Domain Model is divided in three parts, GUI, viewer and common. These build on each other hierarchically.

GUI The GUI is a thin C# client responsible of saving the user configurations and pass them to the viewer package in form of a `json` file. It also contains the OpenGL window (`GLFWindow*`) as a child window of the main Window Form. The GUI starts also the C++ application.

viewer Is the C++ application's core. It loads the configuration received from the GUI and uses the OpenGL libraries to generate the 3D graphics displayed. It obtains the needed data from the common package. The principal classes in the viewer package are:

`Mesh`: A Mesh contains the Vertex data representing an Anatomy Object

`Scene`: Contains the Anatomy Objects or Meshes, loading them from the STL files and ordering them to render. It is also responsible of the initialization of the calculations needed for displaying the movement.

`GameController`: Initializes the whole viewer package from the `json` configuration and runs the game loop.

`UIController`: Contains the `GLFWindow*`, registers the `callback functions` used by OpenGL to manage user input and directs this input to the `Camera`.

common Contains helper classes for reading the in `json` saved configuration and parsing STL and MVM files containing the graphical and movement data.

4.2.7 Development Environment

In this section we enumerate the technologies employed in the project and explain why we chose them.

4.2.7.1 Technologies

OpenGL Graphic processing technology, for more detailed information check the glossary OpenGL and the 5.2 OpenGL section

Programming Languages: C++, C #

Integrated Development Environment (IDE): Visual Studio 2015

Source Control Management (SCM): Git

Project Management Tool: Visual Studio Team Services [1]

4.2.7.2 Why these technologies

OpenGL OpenGL is an obligatory technology as mentioned in 2 Terms of Reference.

Programming Languages:

> **C++:** OpenGL is programmed in C++. Although none of us have experience in C++, we chose this language in order to avoid performance loses and maintain compatibility with the OpenGL libraries
>
> **C #:** As Konrad Höpli have experience with C #, and in order to comply with the 4.2.2 FR 10, a modern GUI, we chose C # for the Graphical Interface

Integrated Development Environment (IDE): Visual Studio 2015
Both project developers have already worked with Visual Studio and the framework supports both C sharp and C++ programming languages. Visual Studio has also plugins which support OpenGL Shading Language syntax highlighting.

Source Control Management (SCM): Git
Is also known by both developers, and is totally integrated in Visual Studio and in Visual Studio Team Services (see below).

Project Management Tool: Visual Studio Team Services [1]
Is a cloud-based project management tool with a very easy setup and maintenance with which Konrad Höpli works daily. The dashboard is clearly designed, allowing a clear view of the current tasks at a glance. All project members and client have access to the platform, which increases transparency. Among other services, it integrates Visual Studio and stores Git repositories.

5 Technical Report

5.1 Introduction and Summary

The contents of this chapter correspond almost to the temporal line or chronology of the project. First we explain the OpenGL concepts we learned and applied in the application, then we describe how we implemented the different file readers and the mathematics behind the movement calculation. After that we continue with the research process we followed in order to improve the accuracy of the movement displayed. Out of this time line are the tests, dependencies and the code statistics, besides of the results.

5.2 OpenGL

5.2.1 Introduction

By definition, OpenGL is a graphics API which provides an abstraction layer between the application and the underlying graphics subsystem. The commands [2] from the program are taken by OpenGL and sent to the underlying graphics hardware, which works on them in an efficient manner to produce the desired result as quickly and efficiently as possible.

There could be many commands lined up to execute on the hardware, and some may even be partially completed. This allows their execution to be overlapped in such a way that one command might run concurrently with an earlier stage of another. Furthermore, computer graphics generally consists of many repetitions of very similar tasks, and these tasks are usually independent of one another. For example, the result of colouring one pixel does not necessarily depend on any other. Just like a car plant can build multiple cars simultaneously, OpenGL can break up the work you give it and work on its fundamental elements in parallel. "Through a combination of pipelining and parallelism, incredible performance of modern graphics processors is realized" [2].

This *abstraction layer* [2] removes the necessity of knowing who made the graphics processor (or graphics processing unit [GPU]), how it works, or how well it performs for the application as well as its developer. It remains possible to determine this information, but the point is that you should not need to.

5.2.1.1 Core and Compatibility Profiles

Since the first OpenGL version in 1992 [2] the graphics hardware and software have continuously evolved at a fast pace. Over time, the price of graphics hardware came down, performance went up, and new features showed up in affordable graphics processors and were added to OpenGL. Most of these features originated in extensions proposed by members of the OpenGL Architecture Review Board. Some interacted well with each other and with existing features in OpenGL, and some did not.

For many years, the ARB held a strong position on backward compatibility, as it still does today. However, this backward compatibility comes at a significant cost. For these and other reasons, in 2008, the ARB decided it would "fork" the OpenGL specification into two profiles.

The first is the modern, **core profile**, which removes a number of legacy features, leaving only those that are truly accelerated by current graphics hardware. This specification is several hundred pages shorter than the compatibility profile. On top of that, on some platforms, newer features are available only if you are using the core profile of OpenGL.

The **compatibility profile** maintains backward compatibility with all revisions of OpenGL back to version 1.0. As a consequence, software written in 1992 should compile and run on a modern graphics card with a thousand times greater performance today than when that program was first produced.

This project is developed in the core profile, with OpenGL version 3.3. As of today, much higher versions of OpenGL have been released (at the time of writing 4.5). The answer to the logical question of why we used OpenGL 3.3 when version 4.5 is out, is that all future versions of OpenGL starting from 3.3 mainly just add additional features to OpenGL without changing the core mechanics; the newer versions just introduce slightly more efficient or more useful ways to accomplish the same tasks or support the latest graphics cards. As a result, all concepts and techniques remain the same within modern OpenGL versions and it is perfectly valid to learn and use OpenGL 3.3 and thereby supporting as many graphics processors as possible [3].

5.2.1.2 Recommended definitions

For a better comprehension of the following sections, we recommend to consult the following terms in the glossary:

- **Vertex**
- **Primitive**
- **Shader** (Extended explanations of this term follow below)
- **Object Space** or **Model Space**

- **World Space**
- **View Space**
- **Clip Space**
- **Window Space**

5.2.2 Graphics Pipeline

Current graphics processing units (GPUs) consist of large numbers of small programmable processors called shader cores, which run mini-programs called Shaders. Each core has a relatively low throughput, processing a single instruction of the shader in one or more clock cycles and normally lacking advanced features such as out-of-order execution, branch prediction, super-scalar issues, and so on.

However, each GPU might contain anywhere from a few tens to a few thousands of these cores, and together they can perform an immense amount of work. "The graphics system is broken into a number of stages, each represented either by a shader or by a fixed-function, possibly configurable processing block" [2]. Figure 8 shows a simplified schematic of the graphics pipeline.

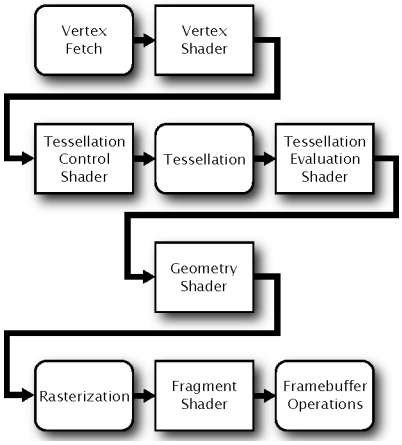


Figure 8: Simplified graphics pipeline [2]

In the figure above, the boxes with rounded corners are considered fixed-function stages, whereas the boxes with square corners are programmable since they can execute Shaders that you developed.

In the vertex fetch stage, we pass in a collection of Vertices or Vertex data as input to the graphics pipeline. These Vertices grouped by three should form a triangle.

The first part of the pipeline is the Vertex shader that takes a single Vertex as input. The main purpose of the Vertex shader is to transform Vertex coordinates into Normalized Device Coordinates. It also allows us to do some basic processing on the vertex attributes.

In the primitive assembly stage or Tessellation, the Tessellation Control shader takes all the Vertices from the Vertex shader and assembles all the point(s) to the Primitive shape given - a triangle for example.

The output of the primitive assembly stage is passed to the Geometry shader. The Geometry shader takes a collection of vertices that form a Primitive and has the ability to generate other shapes by emitting new vertices to form new (more of the same or even other) Primitive(s).

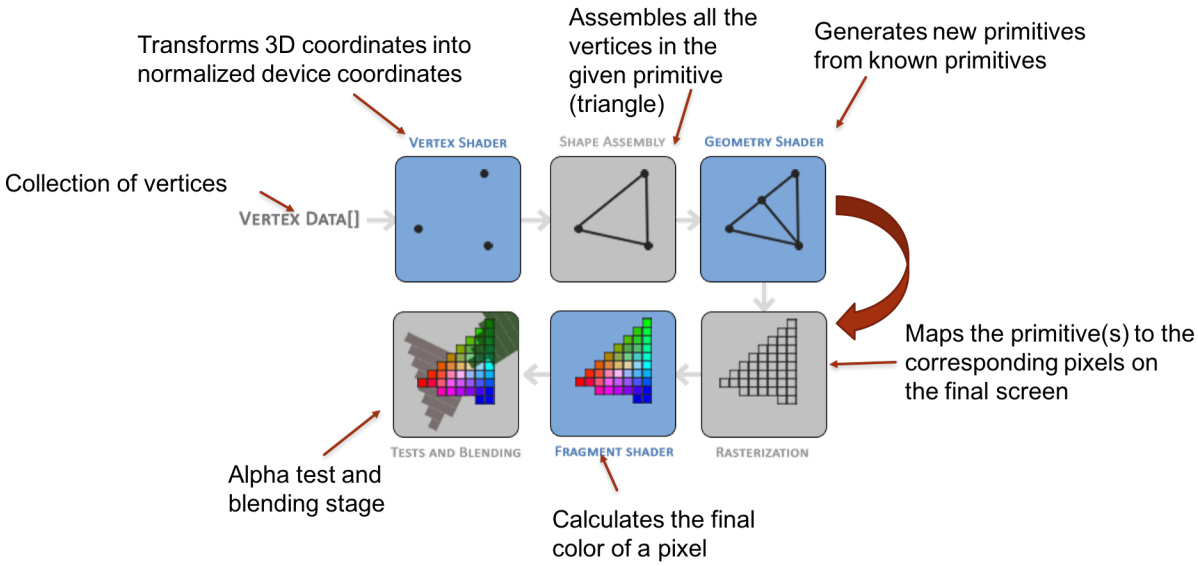
The output of the Geometry shader is then passed on to the rasterization stage, where it maps the resulting Primitive(s) to the corresponding pixels on the final screen (2D), resulting in fragments for the Fragment shader to use. Before the Fragment shader runs, clipping is performed. Clipping discards all fragments located outside of the viewing perspective fitting the screen, resulting in a better performance.

The main purpose of the Fragment shader is to calculate the final colour of a pixel and this is usually the stage where all the advanced OpenGL effects occur. The Fragment shader contains data about the 3D scene that it can use to calculate the final pixel colours (e.g. lights, shadows, colour of the light, reflections and so on).

After all the corresponding colour values have been determined, the final object will then pass through one more stage called the alpha test [4] and blending stage [5]. This stage checks the corresponding depth (and stencil) value of the fragment and uses those to check if the resulting fragment is in front or behind other objects and should be discarded accordingly. The stage also checks for alpha values (alpha values define the opacity of an object) and blends the objects as necessary. So even if a pixel output colour is calculated in the Fragment shader, the final pixel colour could still be something entirely different when rendering multiple primitives.

In the figure 9 below we can now observe the graphics pipeline with a short description of what each Shader does at its corresponding stage.

Figure 9: Extended graphics pipeline [3]



5.2.3 Shaders

"Shaders are little programs that rest on the graphics processing unit (GPU) and transform inputs to outputs. They usually form a chain in which the output of a first shader serves as input of the following shader." [6]

Each Shader executes in a different section of the OpenGL pipeline. All Shaders are executed on the GPU, and as the name implies, they (typically) implement the algorithms related to the lighting and shading effects of an image. However, Shaders are capable of doing much more than just implementing a shading algorithm. They are also capable of performing animation, tessellation, or even generalized computation.

OpenGL Shaders are written in the OpenGL Shading Language (GLSL). This language has its origins in C, but has been modified over time to make it better suited to be run on graphics processors, containing features targeted at vector and matrix manipulation. The compiler for this language is built into OpenGL.

The source code for a Shader is placed into a *shader object* and compiled. Multiple shader objects can then be linked together to form a *program object*. Each program object can contain Shaders for one or more shader stages. The shader stages of OpenGL are Vertex shaders, Tessellation Control shaders and evaluation shaders, Geometry shaders, Fragment shaders, as well as compute shaders.

5.2.4 Shaders Syntax

The goal of this section is to provide a short introduction to the OpenGL Shading Language and Shader syntax. For deeper comprehension please refer to the *OpenGL 4 Shading Language Cookbook* [6].

Shaders always begin with a version declaration, followed by the main function as well as an optional list of input and output variables, uniforms [3]:

Listing 1: Vertex Shader syntax [6]

```
#version version_number

in type in_variable_name;
in type in_variable_name;

out type out_variable_name;

uniform type uniform_name;

void main() {
    // Process input(s) and do some weird graphics stuff
    //...
    // Output processed stuff to output variable
    out_variable_name = weird_stuff_we_processed;
}
```

5.2.5 Vertex attributes and its number in a vertex shader

In GLSL, the mechanism for getting data in and out of Shaders is declaring global variables with the `in` and `out` storage qualifiers [2].

At the start of the OpenGL pipeline, the `in` keyword is used to bring inputs into the Vertex shader. Between stages, `in` and `out` can be used to form conduits from shader to shader and pass data between them.

The declaration of a variable with the `in` storage qualifier marks the variable as an input to the Vertex shader, which means that it is essentially an input to the OpenGL graphics pipeline. It is automatically filled in by the fixed-function vertex fetch stage. The variable becomes known as a *vertex attribute* [6].

Vertex attributes are how vertex data is introduced into the OpenGL pipeline. To declare a vertex attribute, you declare a variable in the Vertex shader using the `in` storage qualifier.

The hardware limits the maximum number of vertex attributes. In OpenGL there are always at least 16 4-component vertex attributes. [3]

5.2.6 Ins and Outs

In order to send data from one Shader to another it is necessary to declare an output in the sending shader and a similar input in the receiving shader. When the types and the names are equal on both sides OpenGL will link those variables together and then it's possible to send data between shaders.

In the code examples below a `vertexColor` variable as a `vec4` output is set in the (Listing 2) vertex shader and a similar `vertexColor` input variable is declared in the (Listing 3) fragment shader. Since they both have the same type and name, the `vertexColor` in the Fragment shader is linked to the `vertexColor` in the Vertex shader.

Because the colour is set to a dark-red color in the Vertex shader, the resulting fragments should be dark-red as well. However, without the output colour specification in the Fragment shader, OpenGL would render the object black or white [3].

Listing 2: Vertex Shader

```
#version 450 core
layout (location = 0) in vec3 position;//The position variable has attribute position 0

out vec4 vertexColor; // Specify a color output to the fragment shader

void main() {
 gl_Position = vec4(position, 1.0); //We give directly a vec3 to vec4's constructor
 vertexColor = vec4(0.5f, 0.0f, 0.0f, 1.0f);//Set the output variable to a dark-red color
}
```

Listing 3: Fragment Shader

```
#version 450 core
in vec4 vertexColor; // The input variable from the vertex shader
                     //(same name and same type)
out vec4 color; // the output color variable

void main() {
    color = vertexColor;
}
```

5.2.7 Uniforms

Uniforms are another way to pass data from the application on the CPU to the Shaders on the graphics processing unit (GPU), but uniforms are slightly different compared to vertex attributes [3]

- Uniforms are **global**. A uniform variable is unique per shader program object, and can be accessed from any shader at any stage in the shader program
- Uniforms keep their values until they are reset or updated

Making a uniform is as simple as placing the keyword `uniform` at the beginning of the variable declaration:

Listing 4: Uniform

```

#version 450 core
out vec4 FragColor; // Since uniforms are global variables, we can define them in
                    //any shader we'd like so no need to go through the vertex
```

```
                    //shader again to get something to the fragment shader.

uniform vec4 ourColor; // we set this variable in the OpenGL code.

void main() {
    FragColor = ourColor;
}
```

5.2.8 Using uniforms

The uniform in the example above (Listing 4) is still empty. In order to fill it, it is necessary to find the index / location of the uniform attribute in the respective Shader in order to update its values. In the following example, we change the colour of a triangle gradually over time [3]:

Listing 5: Using uniforms

```

GLfloat timeValue = glfwGetTime();  // retrieve the running time in seconds
GLfloat greenValue = (sin(timeValue)/2) + 0.5;//vary the color in the range of 0.0 - 1.0

// query for the location of the ourColor uniform
GLint vertexColorLocation = glGetUniformLocation(shaderProgram, "ourColor");
glUseProgram(shaderProgram); //updating a uniform requires to first use the program
glUniform4f(vertexColorLocation, 0.0f, greenValue, 0.0f, 1.0f);// set the uniform value
```

Once the uniform values have been set, we can use them for rendering:

Listing 6: Using uniforms for rendering

```
//Changing the color gradually updating the uniform each render iteration
while(!glfwWindowShouldClose(window)) {
    // Check and call events
    glfwPollEvents();

    // Render
    // Clear the colorbuffer
    glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    // Be sure to activate the shader
    glUseProgram(shaderProgram);

    // Update the uniform color
    GLfloat timeValue = glfwGetTime();
    GLfloat greenValue = (sin(timeValue) / 2) + 0.5;
    GLint vertexColorLocation = glGetUniformLocation(shaderProgram, "ourColor");
    glUniform4f(vertexColorLocation, 0.0f, greenValue, 0.0f, 1.0f);

    // Now draw the triangle
    glBindVertexArray(VAO);
    glDrawArrays(GL_TRIANGLES, 0, 3);
    glBindVertexArray(0);
}
```

Resulting in:

Figure 10: Uniform usage result [3]

5.2.9 Model Space, World Space, Matrices and Transformations

After describing the basics of OpenGL and its building blocks, the Shaders, we continue with the different coordinates spaces and transformations used to finally display the graphics on the screen.

OpenGL expects all the Vertices we want to become visible, to be in Normalized Device Coordinates after each vertex shader run; coordinates outside this range will be clipped and therefore not be visible. These NDC coordinates are then given to the rasterizer to transform them to 2D coordinates/pixels on a screen [3].

Transforming vertex coordinates to NDC and then to screen coordinates is usually accomplished in a step-by-step fashion where we transform an object's vertices to several coordinate systems before finally transforming them to screen coordinates. The most popular transformations are referred to as model, view and projection.

The coordinate systems commonly used in 3D computer graphics are:

- **Object Space or Model Space**
- **World Space**
- **View Space**
- **Clip Space**

In this section, we examine each of the coordinate spaces, and the transforms used to move vectors between them.

5.2.10 Object Space

Object coordinates, *Model Space* or *Local Space*. The positions of Vertices are interpreted relative to a local origin. Consider a cube as model. The origin of the model would probably be the centre of gravity or one of the vertices (corners) of the cube, (0,0,0) for example. The origin is often the point used to rotate the model to place it into a new orientation [2].

5.2.11 World Space and Model Matrix

"This is where coordinates are stored relative to a fixed, global origin" [2]. This is the coordinate space where you want your objects transformed to in such a way that they are all arranged within one place (preferably in a realistic fashion). The coordinates of your objects are transformed from model to world space using the **model matrix**.

5.2.11.1 Model Matrix

"The model matrix is a transformation matrix that translates, scales and/or rotates your object to place it in the world space at a location/orientation they belong to" [3].

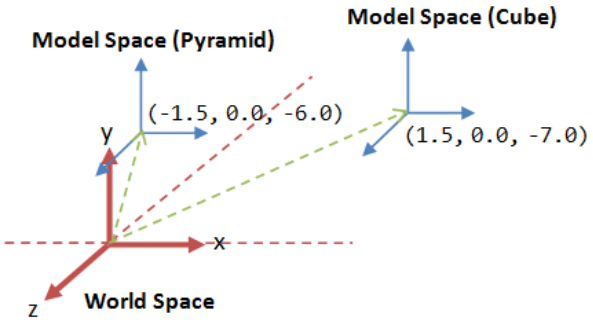


Figure 11: Model and World Spaces [7]

5.2.12 View Space and View Matrix

View Coordinates, *Eye Space* or often simply *Camera*. View coordinates are relative to the position of the observer (hence the terms "camera" or "eye space") regardless of any transformations that may occur; you can think of them as "absolute" coordinates [2].

The view space is the result of transforming the world space coordinates to coordinates that are in front of the user's view. These transformations are generally stored inside the so called **view matrix**.

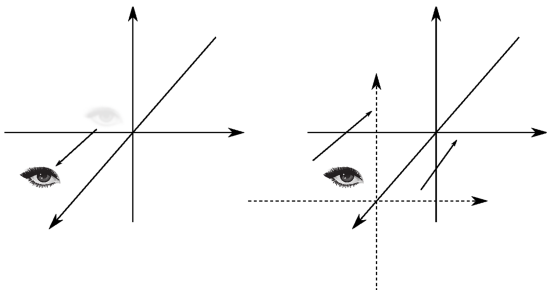


Figure 12: Two perspectives of View Coordinates [2]

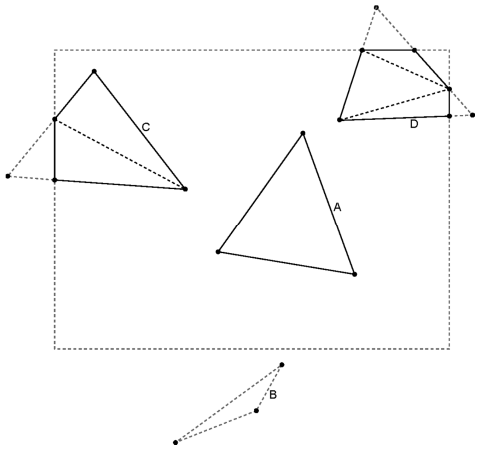


Figure 13: Clip Space [2]

5.2.13 Clip Space and Projection Matrix

Clipping is an action consisting in discard coordinates which are not in a specified range. The remaining coordinates will end up as visible fragments on the screen. Therefore, the *Clip Space* is the space within which the coordinates are eventually displayed on the screen:

To transform vertex coordinates from view to clip-space a **projection matrix** is defined specifying a range of coordinates. The projection matrix then transforms coordinates within this specified range to Normalized Device Coordinates. All coordinates outside this range will not be mapped and therefore be clipped [3].

5.2.14 Orthographic and Perspective Projections

The *viewing box* product of the **projection matrix** is also called a Frustum. Each coordinate that ends up inside the frustum will end up on the user's screen.

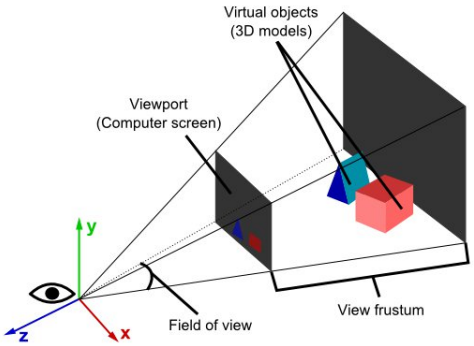


Figure 14: Frustum [8]

The process to convert coordinates within a specified range to NDC is called **projection** or **projection transformation** since the projection matrix projects 3D coordinates to the 2D Normalized Device Coordinates."The projection transformation specifies how a finished scene is projected to the final image on the screen" [2].

The projection matrix to transform view coordinates to clip coordinates can take two different forms, where each form defines its own unique frustum. It is possible to either create an **orthographic projection matrix** or a **perspective projection matrix**.

5.2.14.1 Orthographic Projection Matrix

An orthographic projection matrix defines a cube-like frustum box for the clipping space where each vertex outside this box is discarded. When creating an orthographic projection matrix we specify the width, height and length of the visible frustum. All the coordinates which end up inside this frustum after transforming them to clip space with the orthographic projection matrix, will not be clipped and therefore visible on the resulting display. The frustum looks a bit like a container:

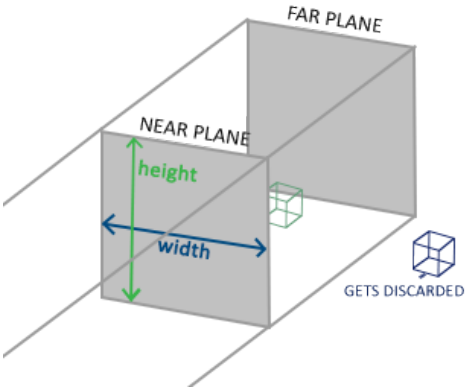


Figure 15: Orthographic frustum [3]

An orthographic projection matrix directly maps coordinates to the 2D plane that is the screen, but in reality a direct projection produces unrealistic results since the projection does not take the perspective into account.

5.2.14.2 Perspective Projection Matrix

The projection matrix maps a given frustum range to clip space too, but also manipulates the `w` value or *homogeneous coordinate* of each vertex coordinate in such a way that the further away a vertex coordinate is from the viewer, the higher this `w` component becomes. This makes distant objects appear smaller than nearby objects of the same size:

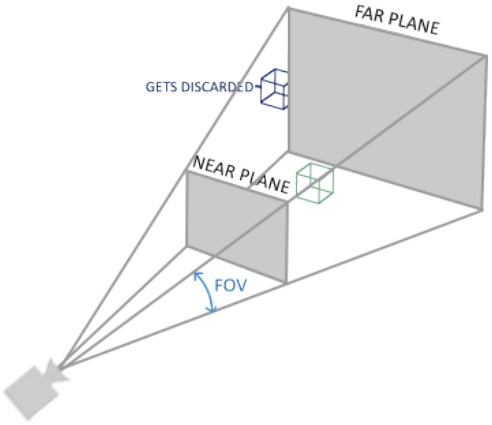


Figure 16: Perspective frustum [3]

A comparison of both perspectives:

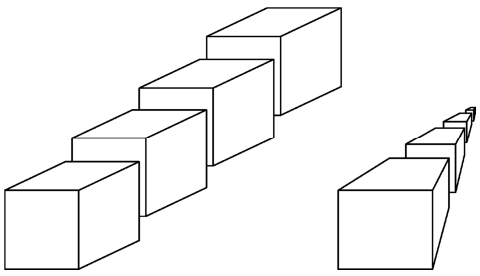


Figure 17: Orthographic and Perspective projections [2]

5.2.15 Spaces and Transformations summary

In the figure 18 we can observe all the transformations or steps needed to display an object or model from its Vertices to the screen:

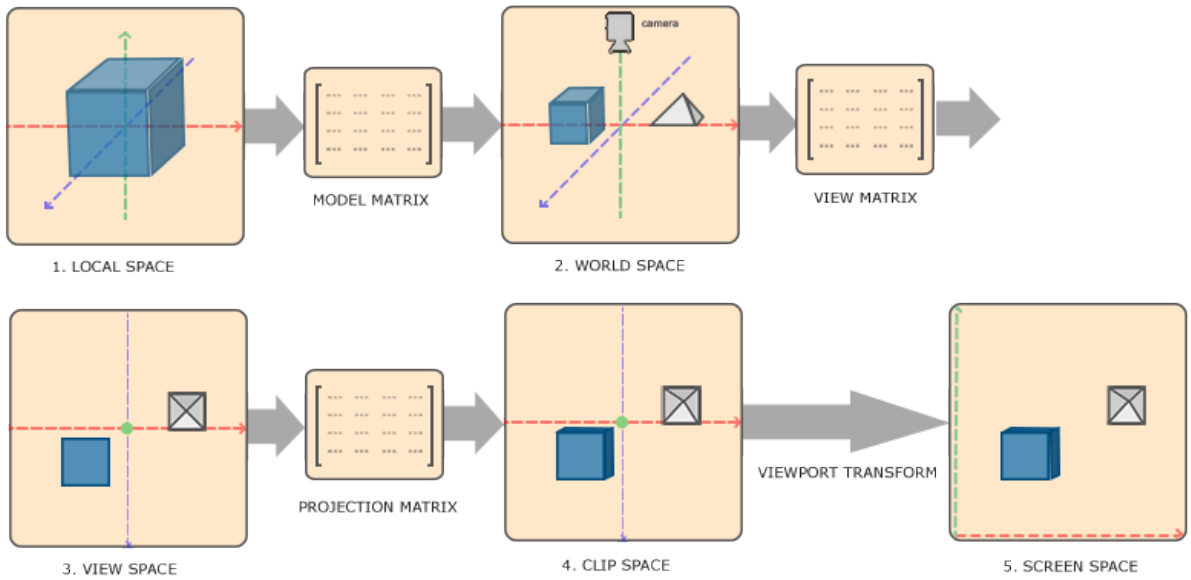


Figure 18: Coordinate systems [3]

5.2.16 Camera

In OpenGL there is no concept of a *camera* as in the video game world. Usually a camera simulation is created "by moving all objects in the scene in the reverse direction", giving the illusion that the observer is moving [3] when instead the objects are just rearranged.

In the View Space the vertex coordinates are seen from the camera's perspective, i.e, the camera perspective seems to be the origin of the scene.
In order to define a camera, the following points need to be known or identified:

- the camera's position in world space
- the direction the camera is facing
- a vector pointing to the right of the camera
- a vector pointing upwards from the camera

This process of forming the view / camera space coordinates system is known as the Gram-Schmidt process [9] in linear algebra and creates a coordinate system with the camera's position as the origin:

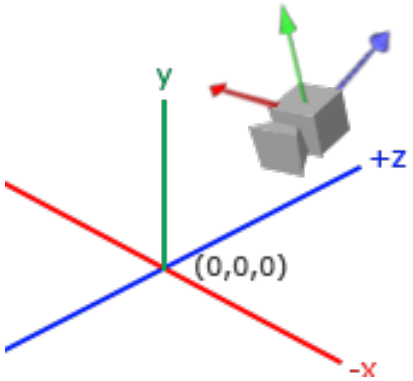


Figure 19: Camera as coordinate system [10]

5.2.16.1 Camera Position

"The camera position is basically a vector in world space that points to the camera's position" [10]. As you can observe in Figure 19, the positive $z-axis$ goes through the screen towards the user. As a result, movement along the $z-axis$ will result in a forward/backward movement of the camera in this case.

5.2.16.2 Camera Direction

Or at what direction is the camera pointing at. If we point the camera to the scene origin $(0,0,0)$, the direction vector will be the result of subtracting the camera position vector form the scene's origin vector.

5.2.16.3 Camera Right Axis

This vector can either be specified yourself (typically: $(1,0,0)$) or calculated based on the *up-vector* that points upwards in world space.

$$cameraRight = cameraUp \times cameraDirection$$

The calculation is using the cross product of the *up-vector* and the *direction-vector*. Since the result of a cross product is a vector perpendicular to both vectors, we will get a vector that points towards the right from the camera's point of view (PoV).

5.2.16.4 Camera Up Axis

This vector can also either be specified yourself (typically: $(0,1,0)$) or calculated based on the *right-vector* that points towards the right from the camera's position.

$$cameraUp = cameraRight \times cameraDirection$$

5.2.16.5 LookAt Matrix

The *LookAt matrix* is defined with the three camera axes plus a translation vector. With it you can transform any vector to that coordinate space by multiplying it with this matrix:

$$LookAt = \begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Where R is the right vector, U is the up vector, D is the direction vector and P is the camera's position vector. Note that the position vector is inverted since we eventually want to translate the world in the opposite direction of where we want to move.

"Using this *LookAt matrix* as our view matrix effectively transforms all the world coordinates to the view space we just defined. The LookAt matrix then does exactly what it says: It creates a view matrix that looks at a given target" [10].

The GLM library [11] provides a `glm::lookAt` function. It is only necessary to specify a camera position, a target position and a vector that represents the up vector in world space. GLM then creates the LookAt matrix that can be used as the 5.2.12 View Matrix using the process described above:

Listing 7: GLM lookAt function example

```
1 glm::mat4 view;
2 view = glm::lookAt(glm::vec3(0.0f, 0.0f, 3.0f),
3         glm::vec3(0.0f, 0.0f, 0.0f),
4         glm::vec3(0.0f, 1.0f, 0.0f));
```

5.2.17 Lighting

"Lighting in OpenGL is based on approximations of reality using models that are much easier to process than reality and look relatively similar" [12]. One of these models is the Phong Lighting Model and it consists of the following three components: *ambient*, *diffuse* and *specular* lighting.

5.2.17.1 Ambient Lighting

"Light in a scene that doesn't come from any specific point source or direction. Ambient light illuminates all surfaces evenly and on all sides" [2]. To simulate this an ambient lighting constant that always gives the objects some colour is used. This constant is added to the final resulting colour of the object's Fragment shader, thus making it look like there always is some scattered light even when there is no a direct source of light.

5.2.17.2 Diffuse Lighting

"Diffuse light is the directional component of a light source" [2] and simulates the directional impact a light object has on an object. This is the most visually significant component of the lighting model. The more a part of an object faces the light source, the brighter it becomes.

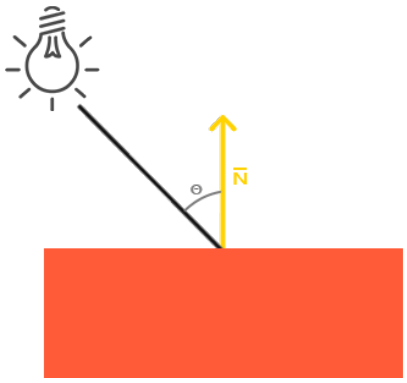


Figure 20: Diffuse Light [12]

In the figure 20 we can see a light source on the left with a light ray pointing towards the object. It is necessary to measure at what angle the light ray touches the fragment.

If the light ray is perpendicular to the object's surface the light has the greatest impact. To measure the angle between the light ray and the fragment we use a normal vector $\overrightarrow{N}$ [13] to the fragment's surface. The angle between the two vectors can then easily be calculated using the dot product.

So, the resulting dot product returns a scalar that can be used to calculate the light's impact on the fragment's colour, resulting in fragments with different lighting/illumination, based on their orientation towards the light.

Therefore, in order to calculate diffuse lighting the following factors are needed [12]:

- Normal vector $\overrightarrow{N}$: a vector perpendicular to the vertex' surface.
- The directed light ray: a direction vector, which is the result of the subtraction of the light's position from the fragment's position. Therefore we need both the light's position vector and the fragment's position vector for our calculation.

5.2.17.3 Specular Lighting

"Specular light is a highly directional property, but it interacts more sharply with the surface and in a particular direction" [2], f.e. from what direction the user is looking at the fragment.

A highly specular light tends to cause a bright spot on the surface it shines on, which is called the *"specular highlight"* [2]. Specular highlights are often more inclined to the colour of the light than the colour of the object.

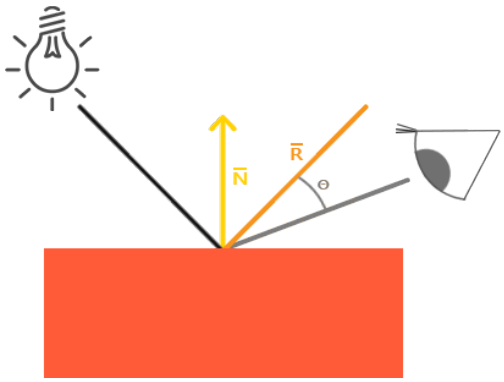


Figure 21: Specular Lighting [12]

We calculate a reflection vector $\overrightarrow{R}$ by reflecting the light direction around the normal vector $\overrightarrow{N}$. Then we calculate the angular distance between this reflection vector and the view direction. The closer the angle Θ between them, the greater the impact of the specular light. The resulting effect is the bright spot when we're looking at the light's direction reflected via the object.

The view vector is the one extra variable needed for specular lighting which can be calculated using the viewer's world space position and the fragment's position. Then we calculate the specular light's intensity, multiply this with the light colour and add this to the resulting ambient and diffuse components.[12]

In the Figure 22 you can see what the different lighting components might look like:

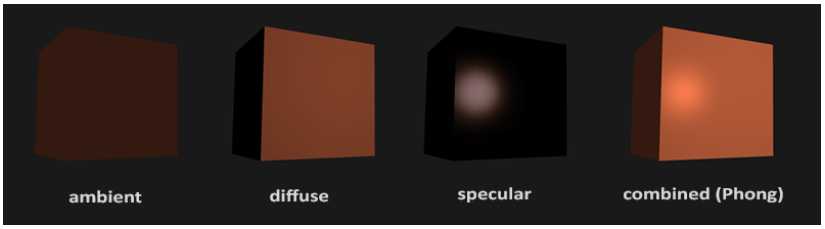


Figure 22: Lighting Components [12]

5.3 Implementation

After the necessary introduction to OpenGL in the previous chapter, this one is aimed at the functionality of the *Jaw Viewer* and the steps encountered over the duration of this project.

5.3.1 Import Anatomy

The first goal of the project was loading Anatomy Objects. As described in 4.2.4.2 UC01: Load Anatomy, the user selects and loads the Anatomy Objects in form of STL files. Before explaining how we import these files, we extend the Stereo Lithography Files explanations in the glossary with more information on the file structure.

5.3.1.1 STL file structure

The structure of a STL file can be ASCII or binary. The **ASCII representation** of a STL file always begin with the line:

`solid name`

where `name` is an optional string. The file contains the representation of any number of triangles with its coordinates:

```
facet normal n_i n_j n_k
    outer loop
        vertex v1_x v1_y v1_z
        vertex v2_x v2_y v2_z
        vertex v3_x v3_y v3_z
    endloop
endfacet
```

Figure 23: STL file Ascii representation [14]

The file concludes with:

`endsolid name`

A **binary STL file** has an 80-character header. Following the header, there is a 4-byte unsigned integer indicating the number of triangular facets in the file as well as the data describing each triangle in turn. The file simply ends after the last triangle.
Each triangle is described by twelve 32-bit floating-point numbers: three for the normal and then three for the $X/Y/Z$ coordinate of each vertex. After these follows a 2-byte ("short") unsigned integer that is the "attribute byte count" [14].

```
UINT8[80] – Header
UINT32 – Number of triangles

foreach triangle
REAL32[3] – Normal vector
REAL32[3] – Vertex 1
REAL32[3] – Vertex 2
REAL32[3] – Vertex 3
UINT16 – Attribute byte count
end
```

Figure 24: STL file Binary representation [14]

5.3.1.2 Importing STL files

The tutorial [3] we used as base for this project employed the C++ library `assimp` [15] to import STL files. As `assimp` introduced an external dependency with a lot more functionality and complexity than required by this project, we decided not to use `assimp` and implement our own STL file reader instead.

As already mentioned in 4.2.6 architecture, the `common` package contains the `StlFileManager` class, responsible of parsing STL files in ASCII and binary format.

The `StlFileManager` receives the path to the file (`parse_stl_file(const std::string filePath)`), checks if the file exists and if it is binary or ASCII, parses it and returns all the Vertex data wrapped in the `StlData` class:

Listing 8: StlData

```
1 class StlData {
2     public:
3     std::string name;
4     std::vector<Triangle> triangles;
5     StlData(const std::string name) : name(name) { }
6 };
```

5.3.2 Display Anatomy

The 4.2.4.6 UC05: Display anatomy movement describes how the system displays the Anatomy objects. In order to explain how this is done, we must first go through the whole application starting process.

We assume that 4.2.4.2 UC01: Load anatomy has already taken place. This means, that the C# GUI hat already saved all the from user entered paths to STL files and other necessary configuration parameters in a `JSON` configuration file. The path to the `JSON` configuration file is passed to the `GameController` class. As its name indicates, this class is responsible for the management of the application and is contained in the `viewer` package.

The `GameController` initializes OpenGL, GLEW [16], the displayable components and loads the configuration from the `JSON` file. For this, the path to the `JSON` file is passed to the `JsonReader`, another helper class in the `common` package, which extracts the configuration data and stores it in the `ConfigurationData` class. The `GameController` run the `GameLoop` and initializes also the `UiControllers`.

The above mentioned displayable elements are exactly that: software objects or classes which represent something displayed on the screen. Among these are the Scene and another elements like the Reference Cube, the Camera LEDs and Meshes. Being a displayable element as well, the scene contains all these elements. Once initialized, the scene initializes the Reference Cube, the Camera Leds as well as the Meshes. The meshes are displayed by means of the STL file paths saved in the `ConfigurationData` class.

Slowly we get closer to the initial goal, displaying the anatomy. As described in the glossary, a Mesh is *already* an anatomy object. After initializing the meshes, the `mesh.loadMeshDataFromStlFile(filePath)` method is called on each mesh object. In this method the `StlFileManager` class is used to get the `StlData` (see 5.3.1.2 Importing STL files), and finally the necessary vertices and indices are obtained from the `StlData`.

Listing 9: Loading Mesh data from Stl files

```
1  void Mesh::loadMeshDataFromStlFile(const std::string filePath) {
2      stl::StlFileManager stlFileManager{};
3      auto stlData = stlFileManager.parse_stl_file(filePath);
4      this->vertices = this->getVerticesFromStlData(stlData);
5      this->indices = this->getIndicesFromStlData(stlData);
6  }
```

At this point in time, each Mesh has loaded all the elements to be displayed on the screen. But all these vertices and indices must be first processed by OpenGL and submitted to the graphics processing unit (GPU). These operations are executed on each Mesh object by calling the method `setupMesh()`:

Listing 10: Setting up a Mesh

```
1  void Mesh::setupMesh() {
2      glGenVertexArrays(1, &this->VAO);
3      glGenBuffers(1, &this->VBO);
4      glGenBuffers(1, &this->EBO);
5      glBindVertexArray(this->VAO);
6      glBindBuffer(GL_ARRAY_BUFFER, this->VBO);
7      glBufferData(GL_ARRAY_BUFFER, this->vertices.size() * sizeof(Vertex),
8                   &this->vertices[0], GL_STATIC_DRAW);
9      glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, this->EBO);
10     glBufferData(GL_ELEMENT_ARRAY_BUFFER, this->indices.size() * sizeof(GLuint),
11                  &this->indices[0], GL_STATIC_DRAW);
12     glEnableVertexAttribArray(0);
13     glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
14                           static_cast<GLvoid*>(nullptr));
15     glEnableVertexAttribArray(1);
16     glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
17                           reinterpret_cast<GLvoid*>(offsetof(Vertex, Normal)));
18     glBindVertexArray(0);
19 }
```

The last step is done in the so called `{game loop}`. While running inside this loop, the `Scene` calls the `Draw(Shader shader)` method for each Mesh object, passing the necessary `Shader` to it. So each Mesh object is finally displayed on the screen (figure 25).

5.3.3 Import movement or MVM files

The Use Case 4.2.4.3 Load movement describes how the user selects and loads the movement files or MVM files. Once more, before going into the explanation of how the MVM files are loaded, we will describe their structure and function.

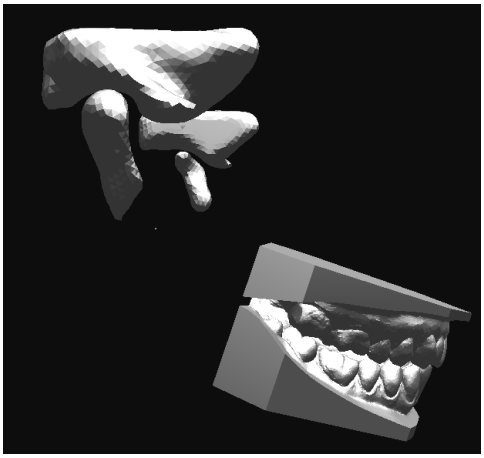


Figure 25: Anatomy Objects in the *Jaw Viewer*

5.3.3.1 MVM files

A MVM or motion movement file is a file generated by the proprietary Optis application used by the clinic. Each file contains snapshots of LED coordinates recorded by 3D cameras during jaw movement. In other words, a MVM file is a recording in coordinates of the patient's jaw movement. The LEDs are grouped in triangles and attached to the patient's jaw. Each of the recording cameras covers one axis, making them essentially 3D, and the coordinates of each camera (X, Y, Z) on one step combined represent one Frame. In addition to the calculated coordinates, the luminosity readings from the sensors used for the detection or identification of the LED are also stored in each set of data.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0x400	camera	data length	step number				data mapping		X/Y/Z-position LED1		X/Y/Z-position LED2		X/Y/Z-position LED3		X/Y/Z-position LED4	
0x410	X/Y/Z-position LED5		X/Y/Z-position LED6		X/Y/Z-position LED7		X/Y/Z-position LED8		X/Y/Z-position LED9		X/Y/Z-position LED10		X/Y/Z-position LED11		X/Y/Z-position LED12	
0x420	X/Y/Z-position LED13		X/Y/Z-position LED14		X/Y/Z-position LED15		X/Y/Z-position LED16		X/Y/Z-position LED17		X/Y/Z-position LED18		lum LED1	lum LED2	lum LED3	lum LED4
0x430	lum LED5	lum LED6	lum LED7	lum LED8	lum LED9	lum LED10	lum LED11	lum LED12	lum LED13	lum LED14	lum LED15	lum LED16	lum LED17	lum LED18		
0x440																

Legend
Camera number
Bit data length
Data number (timestamp)
0X00C8*(200)
X/Y/Z-Position
Intensity (Luminosity)
Reserve

Figure 26: MVM file structure

The figure 26 shows the structure of one camera. Placed after the 1024 bytes file header, the camera structure has a length of 62 bytes and contains among other data, the coordinates of each camera Led and its luminosity value. This structure is repeated until file ends.

In a MVM file these structures are grouped by three, forming a Frame, and within each frame the 18 LEDs forming triangles (f.e. LEDs 1, 3, 5 resemble a triangle). Each structure is separated from the next by an empty padding space of 194 bytes.

The set of frames saved in a MVM file constitute a movement. You can think of the frames like the individual film frames which together compose the complete moving picture.

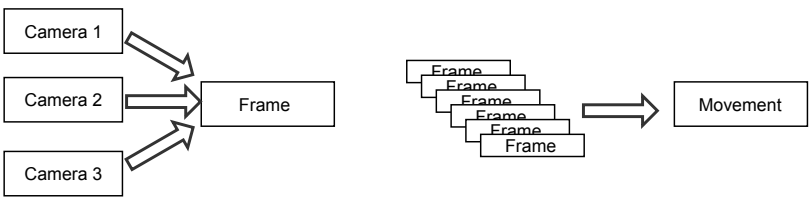


Figure 27: Frames and Movement

5.3.3.2 Reading MVM files

Now to the loading or reading process. The MVM files are a proprietary format of the Clinic for Masticatory Disorders, and therefore there are no C++ libraries which support parsing them. Due to that fact, we had to implement a custom MVM file reader. The `MvmFileManager` class, another helper class in the `common` package, accomplishes this functionality.

The `MvmFileManager` uses the `MvmFileCamera` struct, which corresponds to the MVM camera structure of Figure 26:

Listing 11: The MvmFileCamera struct

```
struct MvmFileCamera {
    uint8_t cameraNumber{};
    uint8_t bitDataLength{};
    uint32_t stepNumber{};
    uint16_t dataMapping{};
    std::vector<uint16_t> leds;
    std::vector<GLfloat> convertedLeds;
    std::vector<uint8_t> luminosities;

    static constexpr double upperOldLevel = 65535.0;
    static constexpr double interval = 327.68;
    static constexpr double bottomNewLevel = -163.84;

    MvmFileCamera() : leds(18), convertedLeds(18), luminosities(18) {}

    /**
    * \brief Translates the given led coordinate from the in MVM file saved format
    * (integer from 0 to 65535) to the new range (-163.84 to 163.84 millimeter)
    * \param originalCoordinate
    * \return GLfloat
    */
    static GLfloat getCoordinateInverseMapping(const uint16_t originalCoordinate) {
        return (originalCoordinate / upperOldLevel) * (interval) + (bottomNewLevel);
    }

    void translateLedCoordinates() {
        std::transform(
            leds.cbegin(),
            leds.cend(),
            convertedLeds.begin(),
            getCoordinateInverseMapping
        );
    }
};
```

Basically, the `MvmFileManager` receives the path to the MVM file, opens it, extracts the data to `MvmFileCamera` instances, processes the data in these instances and then returns the information as a collection of frames:

Listing 12: MvmFileManager getFramesFromMvmFile method

```
std::vector<Frame> MvmFileManager::getFramesFromMvmFile(const std::string pathTofile) {
    std::ifstream mvm_file(pathTofile, std::ios::in | std::ios::binary);
    static_assert(CHAR_BIT == 8, "Expecting char to consist of 8 bits");
    try {
        auto fileLenght = this->getFileLength(mvm_file);
        auto fileIndex{firstFileIndex};

        while (fileIndex < fileLenght) {
            auto camera1 = this->getCameraData(mvm_file, fileIndex);
            fileIndex += fileIndexOffset;   //fileIndexOffset = 256;
            auto camera2 = this->getCameraData(mvm_file, fileIndex);
            fileIndex += fileIndexOffset;
            auto camera3 = this->getCameraData(mvm_file, fileIndex);
            fileIndex += fileIndexOffset;
            auto frame = this->getFrameFromCameradata(camera1, camera2, camera3);
            this->frames.push_back(frame);
        }
        mvm_file.close();
        return this->frames;
    } catch //... error handling
```

The `MvmFileManager::getCameraData(...)` method recursively extracts the data from the `ifstream` to an instance of the `MvmFileCamera` struct and translates the LED coordinates. This translation is necessary because the LED

coordinates in the MVM file are saved as unsigned integers from 0 to 65535 (for the transmission protocol), and in the *Jaw Viewer* the coordinates must be calculated back into their original range from -163.84 to 163.84 millimeters. The method `translateLedCoordinates` in listing 11 executes this translation.

As already mentioned above, three `MvmFileCamera` instances are necessary to form a frame, and the camera structures are separated by 194 empty bytes. Because of that, the method `getFramesFromMvmFile` (Listing 12) iterates through the file in "jumps" of 256 bytes until the MVM file ends. In each of this iterations, when three `MvmFileCamera` instances are ready, the method `MvmFileManager::getFrameFromCameradata(camera1, camera2, camera3);` creates an instance of the `Frame` class (Listing 13) and saves it in a `std::vector`.

Listing 13: Frame class

```
1  #pragma once
2  #ifndef FRAME_H
3  #define FRAME_H
4  #include <vector>
5  #include "mvm_led.h"
6
7  /**
8  * \brief This struct contains all the data of a frame obtained from a mvm file.
9  * The frame step number, the time stamp and 18 Leds with its
10 * corresponding coordinates and luminosities
11 */
12 struct Frame {
13     int stepNumber;
14     float timeStamp;
15     std::vector<MvmLed> leds;
16 };
17 #endif
```

Once the end of a file is reached, the `MvmFileManager::getFramesFromMvmFile(...)` returns the filled `std::vector<Frame>`.

5.3.4 Display movement

The Use Cases UC03 4.2.4.4 and UC05 4.2.4.6 describe how the user configures the movement, and then how the motion of the Anatomy Objects is displayed. In this section, we will go into some of the mathematical foundations of the calculations needed to display the movement and then how these calculations are applied in our project.

5.3.4.1 Movement Calculations

Based on the mathematical analysis of Prof. Augenstein (see Appendix E.1 Movement in R3), we extract the movement contained in the coordinates of individual frames imported from the MVM files (see 5.3.3.2 Reading MVM files) into the form of a transformation matrix. This matrix contains both the rotation and translation needed to move the original triangle to the position of the new one. We can then apply this matrix to any Anatomy Object respectively all of its triangles and thereby moving it on the screen.

In a first approach we simply used the very first set of LED coordinates as a reference and then calculated the movement between those and every following set of coordinates. We then iterate through all the frames and calculate the transformation matrix between this first frame and each of them. In other words, the motion is always calculated between the reference coordinates and the current frame. The two LED triangles in the first set of demo-data we were given also contained distinctively different movements. One of the triangles yielded very limited movements that we interpreted as a head-shaking whereas the other triangle actually provided the movement of the jaw. Since the head-shaking is not necessarily a desired movement in the final display, we went ahead and used the inverse matrix of our calculation and then applied that to all of the Anatomy Objects. As a result, all the displayed objects remain completely still and the ones affected by another movement-matrix still have their shaking reduced visibly.

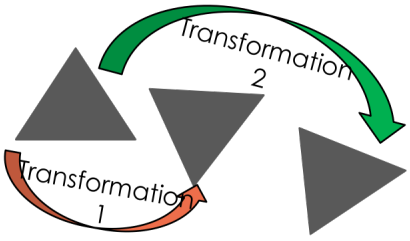


Figure 28: Movement transformation matrix

For each transformation matrix the following auxiliary variables are required (Appendix E.1, 1.3):

Reference Direction Vectors k, m

Transformed Direction Vectors k', l', m'

Reference Centroid s

Transformed Centroid s'

These are calculated for each LED triangle using the reference-set of coordinates and the respective coordinates for the same LEDs in each of the remaining frames. Eventually, this function (Appendix E.1, 1.4) allows us to calculate the transformation matrix (4x4) between any given set of two triangles and can then be applied wherever needed.

$$\boxed{f(x) = k'(k, x-s) + l'(l, x-s) + m'(m, x-s) + s'}$$ Where x is the desired matrix-component

Each transformation matrix is saved in an instance of the `TargetFrame` class and these instances in a collection in the `Scene` class. On each iteration of the game loop, a transformation matrix is extracted from the collection and applied to the vertices of the Anatomy Object selected to be animated.

5.3.4.2 Movement Display

We can assume that the Use Cases 4.2.4.3 Load Movement 4.2.4.4 Configure Movement have already took place. Similar to the 5.3.2 Display Anatomy chapter, the `GameController` initializes the `Scene` class and calls the `scene.init()` method.

The `init()` method among other things calls the `generateTargetFrames()` method, which accepts the path to the MVM file as parameter and creates an instance of the `MvmFileManager`. As described in 5.3.3.2 Reading MVM files, the `MvmFileManager` returns a collection of `Frames`.

The `generateTargetFrames()` method then first obtains the `Frames`, saves the reference frame locally and calculates the transformation matrices by the means of this first frame (see above, 5.3.4.1 Movement Calculations), saving each transformation matrix in its corresponding instance of the `TargetFrame` class and storing these instances in a `std::vector<TargetFrame>` in the `Scene`.

At this point in time, the `Scene` has already loaded the Anatomy Objects (STL files). So, when the `GameController` runs the game loop, in each iteration of the loop it calls the `render()` method in the `Scene`. In this method, the index necessary for the display of the currently displayed movement is calculated. With this index the corresponding movement matrix contained in the `std::vector<TargetFrame>` is obtained and passed to the `scene.drawMeshes()` method.

The `scene.drawMeshes()` method differentiates between the static Anatomy Objects (not animated) and the dynamic Anatomy Objects (animated). It passes the received transformation matrix with the Shader program to a subsequent method `translateModel`.

As its name indicates, the `translateModel` method uses the given matrix and shader to translate the Anatomy Objects' coordinates in the world space and ends up generating the motion feeling with the continuous transformation in each game loop-iteration.

5.3.5 Reverse Engineering

In the interim presentation (held on the 10th of April, 2017) we were advised by the Clinic of Masticatory Disorders that the mastication movement was not displayed correctly both regarding the angle and range of movement.

In order to correct this, we arranged a meeting with the Clinic the day after. In this meeting we received new information that was not previously known or understood by us or Prof. Augenstein. This section is named after the process we unfortunately had to follow after receiving this information in order to find a solution for the movement display, as even the Clinic was not able to explain the origin of some of the new given data.

Below we describe the different solution attempts as well as their results.

5.3.5.1 Reference MVM file

As mentioned in 5.3.4.1 Movement Calculations, we initially used the first Frame extracted from the MVM file as reference for the calculation of the transformation matrices.

Until that moment we were informed that MVM files contained the movement data. But we were not aware of the fact, that there were **two** kinds of MVM files:

- A **reference** MVM file
- A **movement** MVM file

A **reference MVM file** is a first recording taken with the patient in an almost motionless situation aside from some natural head-shaking. This is a reference recording for the movement recording after. A **movement MVM** is the recording of the mastication movement of the patient, taken after the reference recording and with the patient actually moving his jaw in a similar position.

Due to this lack of information, we were simply using the first frame of the **movement MVM** as reference. Knowing this, our solution attempt consisted in averaging all the coordinates contained in the reference MVM file and then using this set of coordinates as reference to calculate the movement in the form of transformation matrices (as explained in 5.3.4.1 Movement Calculations).

But this approach did not result in the correct movement either. The displayed movement was still wrong.

5.3.5.2 Calibration Files

Among the new information from the Clinic also was the existence of a new file type unknown until that moment. This file type is a **Calibration** file in a **.scn** format. This also is a proprietary file of the Clinic of Masticatory Disorders. These files are generated from TMJViewer and serve it as some form of configuration for a specific scene. The for our case apparently useful parts of its structure are the following:

```
[Reference1]
Name                = Back
Diameter            = 10.00
Type                = negative
Stack               = 2
Position            = (36.90, -15.84, 76.66)
Calibration         = (-15.00, -9.02, 7.82)

[Reference2]
Name                = Front
Diameter            = 10.00
Type                = negative
Stack               = 2
Position            = (55.58, -22.26, 76.82)
Calibration         = (14.97, -9.06, 7.90)

[Reference3]
Name                = Top
Diameter            = 10.00
Type                = negative
Stack               = 2
Position            = (52.59, -0.21, 76.69)
Calibration         = (-0.01, 17.04, 7.90)

[Led1]
Name                = Back
Calibration         = (-9.96, -11.84, 35.24)

[Led2]
Name                = Front
Calibration         = (9.76, -11.90, 35.10)

[Led3]
Name                = Top
Calibration         = (0.33, 8.04, 35.14)
```

The explanations of the Clinic revealed that the **Calibration** values are some numerical values obtained from a device responsible of calibrating the 3D cameras. We could not figure out in which measurement unit or from which reference coordinates system these values come from. The **Reference1, 2, 3** are the coordinates of virtual *Reference Spheres*, spheres from them until that moment we had no information about, and from them again we do not know to which reference coordinate system they belong. Unfortunately the explanations from the different members of the Clinic about the reference spheres and calibration values were divergent and scattered, so they did not help us to get on.

Here is where the true reverse engineering process started, as neither we or Prof. Augenstein know what are the meaning of the reference spheres, the calibration values and above all, what the relationships between the coordinates systems created by each of these elements are or to which coordinates system they belong to.

After following through with our own analysis and multiple attempts with incorrect results, we also had to realize that the Clinic was using different versions of their software within the demo-sets we were provided with. Since that difference in the software version was not documented or visible anywhere, the calibration data-sets also were unreliable and even if the calculation might have been correct for one set, it could have been incorrect for another.

Calibration Attempt #1

After our initial analysis of the resulting shapes from the calibration-, LED- and reference-coordinates in the one SCN file mentioned above, we came to the conclusion that the triangle shapes are as follows:

- Reference Position: Equilateral Triangle
- Reference Calibration: Triangle
- LED Calibration: Triangle

Since the LED coordinates in the corresponding MVM file also formed an equilateral triangle, we came to the conclusion that a transformation matrix from the calibration had to be calculated first and applied to all coordinates in the MVM file. The resulting animation however still looked incorrect.

Calibration Attempt #2

Since the first attempt with the calibration files did not lead us to the desired result, we consulted our partners from the Clinic as well as Prof. Augenstein again in the hopes of fixing the issue. However, at this point we had already lost about a week of our time due to the caused confusion as well as uncertainty of how to process the provided data and still did not receive reliable answers to our questions. So, together we came to an agreement that we will only invest very limited, additional time resources into this aspect in favour of other, more essential parts of the project. Prof. Augenstein then came up with another theory based on a new set of demo data provided by the Clinic, which aligned better with the explanation and amount of values provided. This approach assumed, that there are three coordinate systems in place - all of which are essential for the correct matching of the coordinates in the MVM and STL files.

- MVM coordinate system
- Calibration/intermediate coordinate system
- STL/anatomy coordinate system

So, in the implementation of this theory, we calculated the following transformation matrices:

- from the so called 'target frame C' (one of the triangles in the MVM coordinates) to the calibration/intermediate coordinate system
- from the calibration/intermediate coordinate system to the STL/anatomy coordinate system

These matrices where then once again applied to all of the MVM coordinates before calculating the movement between the individual frames. The result of a quick attempt of applying this process looked promising, but unfortunately we were not provided with the correct animation on the one set of demo data and thus unable to verify it. Since the mentioned demo data also was manually made for us, it would not have been a suitable to make a general statement about its reliability in productive use anyways.

5.3.6 Perspective, Movement and Rotation

This chapter focuses on explaining the implemented overhaul of the used perspective and navigation within the OpenGL component of the `Jaw Viewer`. It mainly consists of a theoretical explanation of the individual aspects followed by the mathematical foundations behind it.

5.3.6.1 General idea and the supporting Variables

The basics of the perspective we have tried to implement evolve around knowing which objects are displayed within our scene and where they roughly are located. The two primary components needed therefore are the camera as well as the center of the scene (calculated via all displayed objects). These components allow us to build a theoretical centre-plane for our scene, which is always being oriented according to the viewing direction of the camera. The origin of that centre-plane (assigned with the letter 'd' below) will then serve us for every aspect described in the following chapters and can interpreted as the generalized rotation centre of the complete scene.

aspect ratio = 1

$\alpha = fov$

$\overrightarrow{s}$ = centroid position

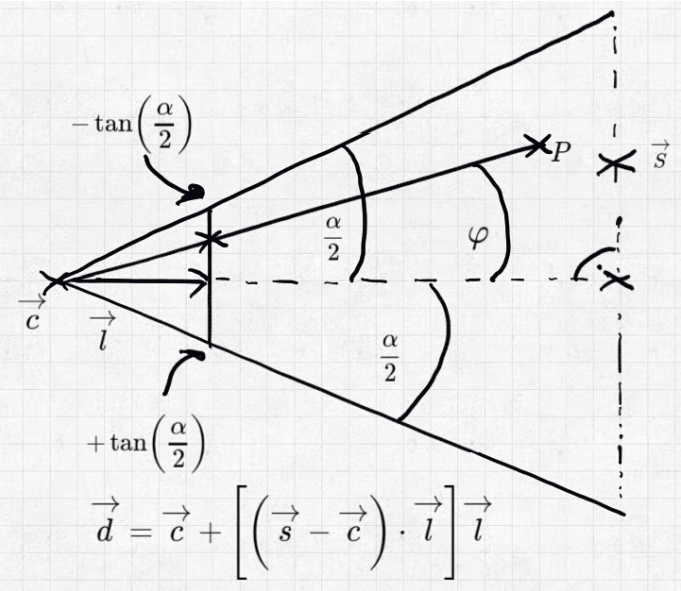
$\overrightarrow{c}$ = camera position

$\overrightarrow{l}$ = look at direction (normalized camera front) $|\overrightarrow{l}| = 1$

$\overrightarrow{d}$ = rotation centre (must be recalculated at each beginning of an operation)

P = A point located within the scene, but also displayed directly on the screen

Visualization:



Display of point P on screen:
X/Y-Coordinates on screen for Point P

$$2\left(\frac{x - x_0}{screen - width/-height} - 0.5\right) \cdot tan\left(\frac{\alpha}{2}\right) = tan(\varphi)$$

Where:
x_0 = left Pixel
w = screen width

5.3.6.2 Zoom

This approach to a zoom implementation does not use a change of the field of view (FoV) of the used camera, but simply works with changes in position of it. Instead of simply using the look at direction (camera front) and a preselected speed value, the formula used here defines a zoom factor and calculates a new position for the camera based thereon. If this zoom factor was set to the value 2 for example, the camera would move to the position, where the whole image appears to be roughly two times the size of what it was before.

Per scroll-wheel rotation:

Enlargement Factor v (scroll backwards)

Enlargement Factor $\frac{1}{v}$ (scroll forwards)

Enlargement Factor after r rotations: v^r (The sign of r result in the scroll direction)

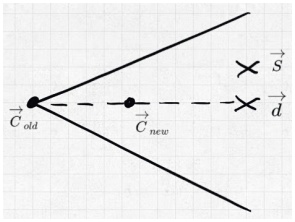
$\overrightarrow{c}_{old}$ = old camera position

$\overrightarrow{c}_{new}$ = new camera position

5.3.6.3 Shifting Motion (Hand-Movement)

The theory behind this aspect of camera movement is for what is referred to as hand-movement in other applications and occurs when the user selects a pixel on the screen with the mouse cursor and then drags (mouse-key held down) the cursor a new pixel, on which he releases the key again. This approach does not pay too much attention to the position calculated within the world-space with the `readPixel` command, that uses the X- and Y-coordinates of a selected pixel on the screen. Instead it projects the two selected points onto the centre-plane mentioned above and

$$v^r = \left(\frac{|\vec{c}_{new} - \vec{d}|}{|\vec{c}_{old} - \vec{d}|}\right)^{-1}$$



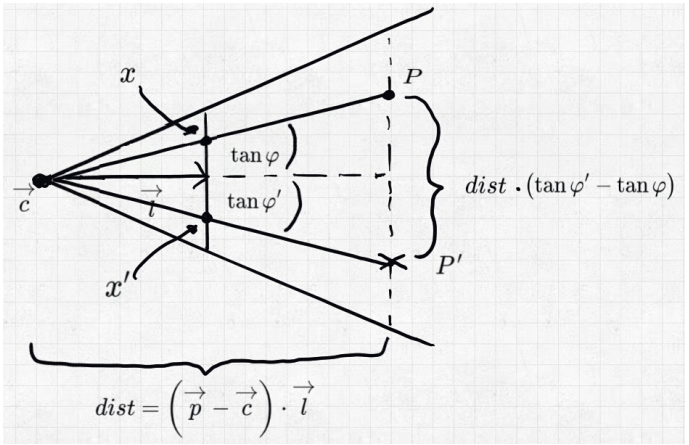
$$\Leftrightarrow |\vec{c}_{new} - \vec{d}| = |\vec{c}_{old} - \vec{d}| \cdot v^{-r}$$
$$\vec{c}_{new} - \vec{d} \parallel \vec{c}_{old} - \vec{d}$$
$$\Leftrightarrow \boxed{\vec{c}_{new} = \vec{d} + v^{-r}\left(\vec{c}_{old} - \vec{d}\right)}$$

Problem, if $\vec{c} = \vec{d}$ (corresponds to infinite enlargement)

then calculates the distance between those two values. This determined distance is then used to move the camera to a new position so whichever point the cursor was released on appears to have moved to the starting point of the executed dragging motion.

Shift Point with Coordinates x to x'

Approach *a*:
Formally correct shift of selected point based on the centre of the object it was located on. This approach only works if a point on any Mesh displayed within the scene (with $z > -1$) was selected and correctly identified by the `readPixel` command. If that condition is not met, a fall-back to approach *b* is required.



P Coordinates obtained via `readPixel` function. Shift of P in x-Direction (world coordinates):

$$\left(\vec{p} - \vec{c}\right)\vec{l} \cdot \left[2\left(\frac{x' - x_0}{w} - 0.5\right)tan\left(\frac{\alpha}{2}\right) - 2\left(\frac{x - x_0}{w} - 0.5\right)tan\left(\frac{\alpha}{2}\right)\right] = 2tan\left(\frac{\alpha}{2}\right)\left(\vec{p} - \vec{c}\right)\vec{l} \cdot \left(\frac{x' - x}{w}\right)$$

In vectorial form:

$$\Delta\vec{p} = \frac{2tan\left(\frac{\alpha}{2}\right)\left(\vec{p} - \vec{c}\right)\vec{l}}{w} \cdot \underbrace{\left(\vec{p}'_S - \vec{p}_S\right)}_{\text{Screen Coordinates of } P' \text{ and } P}$$

$$\vec{c}_{new} = \vec{c}_{old} - \Delta\vec{p}$$

Conversion of Screen Coordinates of P' and P in 3-D direction according to Camera Coordinate-System:

$$\vec{l} = \vec{l}_z \qquad \vec{l}_x = \frac{\vec{l} \times \vec{n}}{|\vec{l} \times \vec{n}|} \qquad \vec{l}_y = \vec{l}_x \times \vec{l}_z$$

Approach *b*:

This approach has the advantage of working with all points on the screen and not only for those on any given Mesh. It is based on a shifting velocity independent of the selected start and end points of the dragging motion, where the velocity is inverse proportional to the enlargement factor. This also results in the effect, that the mouse cursor ends up being inaccurate and therefore misleading for the user. This mouse-cursor issue can be avoided if the selected pixel is marked in the model, but the mouse cursor hidden afterwards. At operation's end the mouse cursor can be set back in the starting position using the `SetCursor` command and the previously determined position.

$$Modified: dist = \left(\vec{s} - \vec{c}_{old}\right) \cdot \vec{l}$$

$$\Rightarrow \Delta\vec{p} = \frac{2tan\left(\frac{\alpha}{2}\right)\left(\vec{s} - \vec{c}_{old}\right) \cdot \vec{l}}{w}\left(\vec{p}'_s - \vec{p}_s\right)$$

$$\vec{c}_{new} = \vec{c}_{old} - \Delta\vec{p}$$

Problem, if $\vec{c}_{old}$ is located on the centre-plane (infinite enlargement $\widehat{=}$ 0-shift factor)

5.3.6.4 Rotations

There are multiple concepts such as the `arcball` available in order to perform a rotation within a scene based on a selected point. The approach we intend to use here is based around the *d* axis also referred to as rotation centre within this chapter and uses a constant rotation velocity. The potentially negative aspect of this approach for the user is, that the rotation performed after selecting an initial starting point can not be reliably connected to the mouse cursor. This is only be possible with a low rotation velocity, but appears to be more appealing when compared to the alternative of inconsistent speeds.

Assumption: the rotation around the pixel distance $\sqrt{\Delta x^2 + \Delta y^2}$ is $\frac{\sqrt{\Delta x^2 + \Delta y^2}}{w} \cdot \beta$ (f.e. $\beta = T_0$)

Rotation Axis: We divide the rotation in 2 parts:

Rotation of $\begin{pmatrix} x \\ y \end{pmatrix}$ around the centre of the scene

After that, rotation from the centre of the scene to $\begin{pmatrix} x' \\ y' \end{pmatrix}$

In both rotations the plane perpendicular to the rotation axis contains the points $\vec{c}$ and $\vec{d}$, and additionally either the point $\vec{c} + \begin{pmatrix} tan(\alpha_x) \\ tan(\alpha_y) \\ 1 \end{pmatrix}$ or $\vec{c} + \begin{pmatrix} tan(\alpha_{x'}) \\ tan(\alpha_{y'}) \\ 1 \end{pmatrix}$

The rotation axis is thus:

$$\vec{l} \times \begin{pmatrix} tan(\alpha_x) \\ tan(\alpha_y) \\ 1 \end{pmatrix} or\ \vec{l} \times \begin{pmatrix} tan(\alpha_{x'}) \\ tan(\alpha_{y'}) \\ 1 \end{pmatrix}$$

Step 1

R_1 = rotation around $\vec{d}$ with rotation axis $\vec{l} \times \begin{pmatrix} tan(\alpha_x) \\ tan(\alpha_y) \\ 1 \end{pmatrix}$ and rotation angle $\sqrt{x^2 + y^2} \cdot \beta$

Step 2

R_2 = rotation around $\vec{d}$ with rotation axis $\vec{l} \times \begin{pmatrix} tan(\alpha_{x'}) \\ tan(\alpha_{y'}) \\ 1 \end{pmatrix}$ and rotation angle $-\sqrt{x^2 + y^2} \cdot \beta$

Step 3

Rotation of $\vec{c}$ around $R_1 R_2$

Rotation of the `lookAt` direction $\vec{l}$ around $R_1 R_2$

Problems: Adjustment of near and far plane necessary: yes

The new $\vec{c}$-Value can coincide with $\vec{s}$ $\widehat{=}$ Enlargement ∞ (shift $\vec{s}$ temporary)

5.4 Testing

5.4.1 Cute Framework

After trying several unit test frameworks we decided to employ the Cute Framework [17], as we already had gathered experience with it in the C++ course at the HSR and the integration into the project offered the least complications.

Cute has no graphical support in Visual Studio, meaning that there is no "Green Bar". On the contrary, as command line framework, it even is possible to execute the tests without an instance of Visual Studio running, which is an advantage in most of cases.

The installation of Cute as standalone version is straight forward, as Cute is a "header library" and thereby only needs to have its header files included in the project.

5.4.2 Testing Proceeding

The *Jaw Viewer* is mostly a graphical application with the primary goal of displaying objects on the screen. Because of this we decided to follow the Test Driven Development principles only for "testable" components, or components with a behaviour suited for unit and integration tests. The correctness of the displayed objects and movements was verified in the meetings by means of reviews by the Clinic, but due to the time loss described in 5.3.5 Reverse Engineering, it was not feasible to realize usability tests during our project.

5.4.3 Unit Tests

The components of the `common` package and some classes of the `viewer` were developed under TDD principles. These components perform mathematical calculations, read files or perform other activities, which can be tested bythe means of unit tests. In each case the equivalence partitioning and boundary-values testing approaches have been followed.

Thanks the data provided by the Clinic the use of mocks could be avoided in this first implementation, always employing original values. An example of this is the `mvm_file_parser_test_suite`, which tests the `MvmFileManager` class. As a reminder, the `MvmFileManager` reads a MVM file and extracts the coordinates saved in it to Frames.
This suite ranges from very specific tests (check if the x LED coordinate is correct) all the way to integration tests in which we check whole blocks of coordinates are read and processed correctly.

As already mentioned, each component or class is tested with a test suite. The creation of a test suite in `cute` is simple. First we declare the test suite header file:

Listing 14: Test Suite header file

```
1  #pragma once
2  #ifndef STL_FILE_PARSER_TEST_SUITE
3  #define STL_FILE_PARSER_TEST_SUITE
4
5  #include "cute_lib/cute_suite.h"
6
7  extern cute::suite make_suite_stl_file_parser_test_suite();
8
9  #endif
```

Each test suite must test only one class or struct and then be registered into Cute:

Listing 15: Test Suite registration

```
1  // ViewerTest.cpp : Defines the entry point for the console application.
2
3  #include "stdafx.h"
4  #include "cute_lib/cute.h"
5  #include "cute_lib/ide_listener.h"
6  #include "cute_lib/xml_listener.h"
7  #include "cute_lib/cute_runner.h"
8  #include <iostream>
9
10 #include "stl_file_parser_test_suite.h"
11
12 bool runSuite(int argc, char const* argv[]) {
13     using namespace std;
14     cute::xml_file_opener xmlfile(argc, argv);
15     cute::xml_listener<cute::ide_listener<>> lis(xmlfile.out);
16
17     auto runner = cute::makeRunner(lis, argc, argv);
```

```
18      auto stlFileParserTestSuite{make_suite_stl_file_parser_test_suite()};
19
20      auto success = runner(stlFileParserTestSuite, "STL_FILE_PARSER_TEST_SUITE");
21      std::cin.get();
22      return success;
23  }
24
25  int main(int argc, char const* argv[]) {
26      return runSuite(argc, argv) ? EXIT_SUCCESS : EXIT_FAILURE;
27  }
```

And finally we write the unit test in the test suite source file:

Listing 16: Test Suite unit tests

```
1   #include "stdafx.h"
2   #include "stl_file_parser_test_suite.h"
3   #include "cute_lib/cute.h"
4   #include "../common/stl_parser.h"
5
6   void test_file_no_exists_exception() {
7       stl::StlFileManager manager{};
8       ASSERT_THROWS(manager.parse_stl_file("foo"), std::logic_error);
9   }
10
11  cute::suite make_suite_stl_file_parser_test_suite() {
12      cute::suite s{};
13      s.push_back(CUTE(test_file_no_exists_exception));
14      return s;
15  }
```

5.5 Dependencies

5.5.1 OpenGL Dependencies

As already mentioned in 5.2.1.1 Core and Compatibility Profiles, we develop this project with OpenGL core profile, version 3.3. This is due first because the tutorial [3] the project bases on is written with the 3.3 version, and also in order to support "not state of the art" hardware. Higher versions of OpenGL (current 4.5) are not supported by older hardware. Despite of OpenGL's backwards compatibility, it's proven [2] that not all hardware supports all new features of newer versions of OpenGL. Therefore we decided to remain by the given 3.3 OpenGL version.

5.5.2 C++ Dependencies

A project goal was the use of the least possible number of external libraries, and less in other programming languages. In other words, the project shall remain pure C++ and C#.

glfw [18] C++ Library for OpenGL. It provides a simple, platform-independent API for creating windows, contexts and surfaces, reading input, handling events, etc.

glew [16] Cross-platform open-source C/C++ extension loading library. Run-time mechanisms for determining which OpenGL extensions are supported on the target platform

glm [11] Mathematics library for graphics software based on the OpenGL Shading Language specifications

jsoncpp [19] C++ library for `json` management

cute [17] C++ Testing Framework. Refer to 5.4 Testing

5.6 Code Statistics

The code lines were counted with Cloc [20]. The lines quantity contains only self written code. External libraries were excluded.
As already mentioned in 5.4 Testing, one of the disadvantages of `cute` is being a command line framework and the absence of integration in Visual Studio. This fact and the lack of code coverage tools in Visual Studio for C++ using this form of testing prevent us from showing code coverage statistics for this project.

Language	Files	code
C++	22	6264
C++ Header	29	672
GLSL	2	44
C#	14	1670
Total	67	8650

Table 11: Code Statistics

5.7 Results

We can divide the results of this bachelor thesis in **Software** and in **Documentation**. The **Software** consists logically on the *Jaw Viewer*, an application which fulfils the Use Cases 01 to 05 (see 4.2.4 Use Cases).

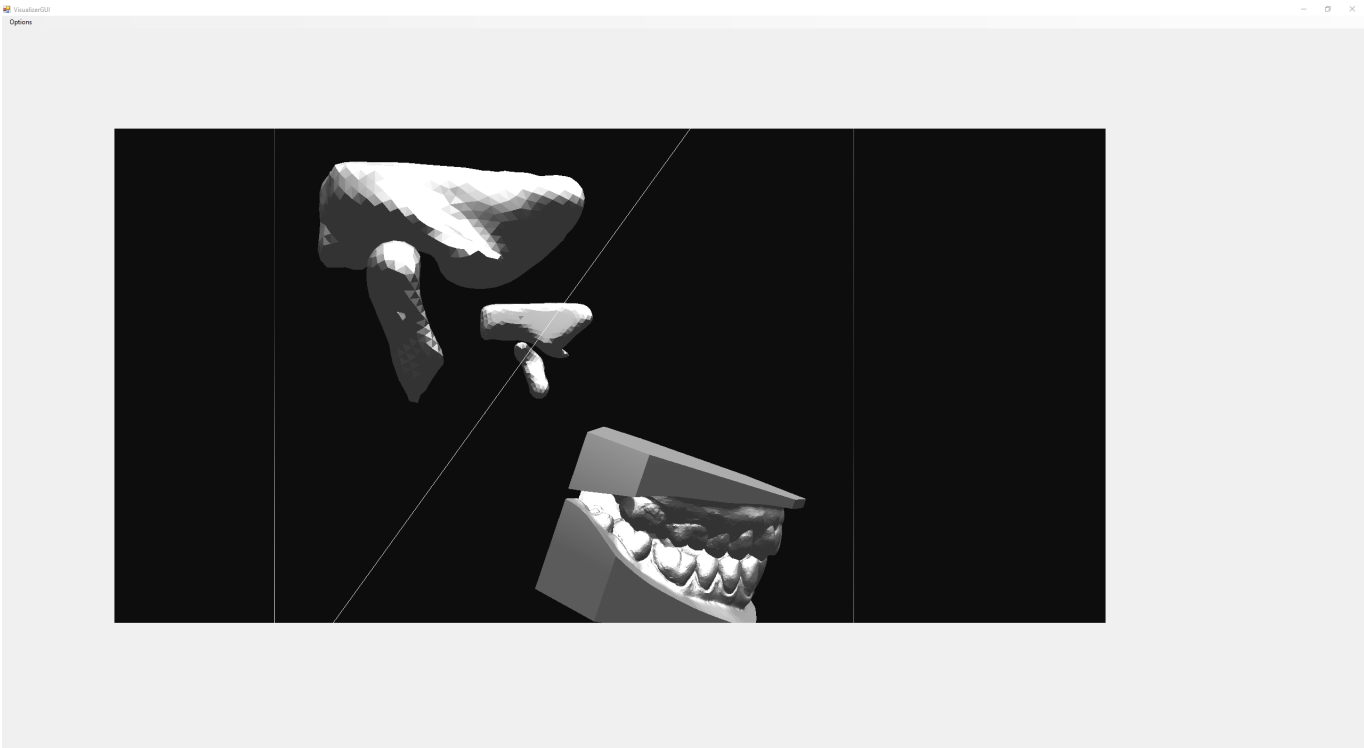


Figure 29: Jaw Viewer

This means that *Jaw Viewer* is an application capable of:

- Importing and processing STL and MVM files
- Configuring the movement and applying it to selected Anatomy Object
- Displaying the Anatomy Objects with and without movement (animation)
- Loading and saving the configuration for the application

But not all news are good. The Use Cases 4.2.4.7 Display anatomy movement in real time and 4.2.4.8 Save anatomy movement in real time could not be implemented within the time frame. During the different meetings with the Clinic, it became obvious that there was a discrepancy about the necessity of this initial requirement for this project, but also various other aspects. For some of the involved members of the Clinic that was a ”must have” requirement as stated in the projects definition, whereas for others it was not. In the meeting after the interim presentation (on the 11th April 2017) the Clinic has agreed to making the real time features not imperative, allowing us to focus on the more basic challenges.

This discrepancy about the requirements was prevalent during the whole project. The meetings often resulted in more scattered requirements (often not specified in the initial project description) and confusion instead of useful and concrete information or instructions. This, combined with our lack of experience in this complex domain as well as handling clients in general, led to a point where we felt forced to distance us from the Clinic and focus on implementing as much as we could. Even if that meant searching for information the client should have provided us with and potentially finding alternative solutions to the challenges occurring.

In 5.3.5 Reverse Engineering we described our attempts in order to find the correct transformation needed for the accurate display of the recorded movement for the Anatomy Objects. As already mentioned, due to the inconsistent and unclear information from the Clinic we could not reach this goal. This has led to the situation where although the *Jaw Viewer* fulfills the use cases mentioned above, it is not ready for use in a productive environment. The displayed movement is not accurate enough and as such unreliable for a good diagnosis for patients.

On the bright side, the application together with the explanations therein can be considered a good base for future development in the same area. All the functionality and most importantly the data structures are already implemented. As such, they could be directly used, although they are subject to another refactoring process and of course a correction of the movement calculations.

This **report**, as the other product of this bachelor thesis, can be considered an important documentation foundation for future projects in the same field. When we started the project there was next to no documentation of the proprietary software used by the Clinic (Optis and TMJViewer), the used coordinate systems and the mathematical theories needed for this specific usage of OpenGL for the dental medicine. With this report, any future developers tasked with the same can save a lot of time in the analysis stage while avoiding the undesired trial and error process we went through. Starting with this gathered information and from a fitting code structure should also allow them to concentrate on the more important and well identified tasks needed to actually create an alternative to the proprietary software currently in use.

Appendices

Appendix A Project Management

This document contains valuable information regarding the planning of the project and other important tasks related to that.

A.1 Risks

This chapter contains the biggest subjects regarding risks encountered and the decision making involved.

- C++
  In the very beginning of the project, after developing a prototype based on the given tutorial, we decided to keep using that prototype and therewith sticking with C++ as the primary programming language. Our lack of experience in development of both this programming language and OpenGL were suboptimal, but heavily favored after considering the use of a wrapping library for OpenGL in a more comfortable language for us (C#) due to the possibility to discover that some features needed by us were not feasible after all.

- WinForms
  Since the Clinic primarily uses both the Windows operating system and VisualStudio, we decided on using WindowsForms for the graphical unit interface even though the integration process of an OpenGL application into that was only vaguely researched and not previously done by either of the developers involved.

- Self-implemented filereading
  Although the prototype created after following the tutorial already supported the import of STL files via an external library, we decided to implement our own. This decision was heavily influenced by the size of the external library (ASSIMP) in comparison to our small usage and the already expected task to implement our own file-reader for the proprietary MVM format used by the clinic for the movement files.

A.2 Project Planning

The *Jaw Viewer* project was run following agile principles, with scrum [21] as development strategy and some milestones as reference points. Due to illness of one team member the usual 14 sprints [22] were extended to 18. The sprints have a duration of a week, with 26 working hours each. Thanks to the extension, now there are 19 weeks between the 20.02.2017 and the new deadline 30.06.17.

A.2.1 Milestones

The milestones mark the end of an implementation phase, the completion of a use case or feature, or an event like the interim presentation. Between the milestones the work procedure is agile, with weekly sprint planings and reviews. The dates in the table below are indicative.

Milestone	Description	Date
ML1	OpenGL Tutorial	28.2.17
ML2	Import STL files	17.3.17
ML3	Import MVM files	26.3.17
ML4	Display Movement	9.4.17
ML5	Interim presentation	10.4.17
ML6	Reverse Engineering	–
ML7	Refactoring	–
ML8	GUI	–
ML9	Perspective	–
ML10	Documentation	30.6.17

Table 12: Milestones

Until the interim presentation the project went as planned, with small delays between milestones caused above all by our lack of experience with C++ and OpenGL. The next milestones after the presentation were the implementation of the graphical user interface (GUI) and the improvement of the screen controls. But, as already mentioned, due to the confusing information from the Clinic we had to invest a big amount of time into trying discover the cause of the inaccurate display of the calculated movement (please, see 5.3.5 Reverse Engineering).

At some point Prof. Augenstein told us to stop the search and to continue with the development of our application because we were not getting where we needed to be. Following that advice, we initiated the necessary refactoring of our library component and then started with the integration into the WindowsForms (C#) GUI. Our lack of experience especially regarding C++ made it basically necessary for us to do a refactoring and also ended up being more time

consuming than expected since the complexity of our component exceeded the base provided by the tutorial by far at that point. The search of a suitable interface between the C++ OpenGL core and the C# GUI took also much more time as planned because the initially researched way to achieve that goal using the `Common Language Runtime (CLR)` by Microsoft did not work with our project specifically. The most likely cause for that failure were the various external dependencies. At this point we begun the preparation (mathematical foundations) for the change of controls, since that was the last of the most desired features we identified, when Konrad Höpli got ill. During his illness, Roberto Cuervo begun with the report, setting the infrastructure (LaTex) and the first report draft.

After his illness, Konrad Höpli primarily continued with the GUI and perspective development. The present report was elaborated and finished by both of the involved students though.

A.2.2 Expected Amount of Work

The theoretical amount of work measured in work hours is:

- Effort per Week: 52 hours
- Total planned effort: 52 hours * 14 Weeks = **728 Hours**
- Total planned effort per student: 728 hours / 2 = **364 Hours**

Until the official deadline on the 16.06.17 there are 14 working weeks plus 2 weeks dedicated to the documentation.

A.2.3 Effective Amount of Work

During the project we planned the tasks in Visual Studio Team Services [1] and booked the corresponding time in hours, resulting in:

Week	**Roberto Cuervo**	**Konrad Höpli**
1	27.8	43.5
2	30.75	25
3	14.5	18.25
4	40.5	10
5	27.75	10
6	8.75	24.5
7	35.75	29.5
8	23	13
9	29.5	24
10	27	37
11	28.75	–
12	24	–
13	22.75	–
14	46	–
15	32.5	8.75
16	16.25	12
17	15	8.25
18	14.5	–
19	40.75	20.5
20	–	26.5
Average	25.29	15.5
Total	505.8 hours	310.75

Table 13: Weekly Amount of work

Student	**Effort / Hours**
Konrad Höpli	310.75
Roberto Cuervo	505.8
Total	816.55

Table 14: Amount of work

In general the planned amount of work corresponded to the effective[2] work. The most laborious tasks were the programming tasks in C++, the comprehension of the mathematical foundations and the elaboration of the documentation, activities to whom we dedicated bigger amount of hours.

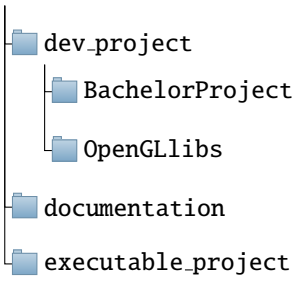
[2]Hours of Konrad Höpli empty due to illness

Appendix B Software Documentation

B.1 Development Setup

In order to facilitate further development of the *Jaw Viewer* we created a Github repository[3] containing all the necessary resources (source code, documentation, etc).

The repository's file structure is:



The setup of the source code project in Windows is straight on. It is just necessary to clone the repository, move the directory `OpenGLlibs` to `C:`, open the Visual Studio Solution contained in `BachelorProject` and build it.

B.2 Installation

The installation of *Jaw Viewer* in Windows is also very simple. You can import the source code to Visual Studio and build the source code (as Release) as described above. Then you must look for the directory `Release` in the Visual Studio project and locate the `GUI` and `main` executable files (`.exe`). You can then move this two files to your preferred location. In order to run *Jaw Viewer* you just have to double click on the `GUI.exe` file.

Alternatively you can also clone the repository. Then you just need to copy the both `GUI` and `main` executable files (`.exe`) contained in the `executable_project` directory to the desired location.

[3] https://github.com/robertocuervo/hsr-master-thesis-doc

Appendix C Field Report

C.1 Konrad Hoepli

As an enthusiast for new technologies, graphical technologies have always seemed very interesting to me, but the development of a graphical component itself has continuously been avoided until very recently since I pretty much feared the complexity it is said to incorporate. This project occurred to me as the perfect opportunity to stand up to this fear and gather my first experiences with it. I was definitely not disappointed by the complexity in the tutorial we followed in the beginning of the project to get familiar with OpenGL, but managed to understand it rather well to my surprise. The fact that this technology is commonly used together with C++ and I would therefore have to step out of my comfort zone even more admittedly was not the way I would have liked, but this also ensured an even bigger learning experienced in the end.

I have definitely extended my knowledge in the realm of OpenGL more than I have in C++, but learned to appreciate both of them especially when combined with powerful external libraries. Even mathematical foundations that were very difficult to understand at first ended up being easily implemented and extremely performant. There have been a lot of difficult problems and mistakes made over the course of this project and as my long illness indicates, it has not been easy to cope with at times. The amount of uncertainties and stress I got myself into in combination with an unhealthy mentality simply overwhelmed me in the end and it took some time to get a decent hold of myself again. Nevertheless I am quite satisfied with this project, even if it is to be interpreted more as a learning experience than a project that exceeded our expectations.

I am very glad to have had Roberto as my teammate and very impressed by his dedication and understanding. The interaction with and support by Prof. Augenstein also was a very positive factor in every aspect of this project once more and I really am extremely thankful for all he has done for us and the outcome of this project.

C.2 Roberto Cuervo

During this project new or only in theory known technologies like OpenGL or WindowsForms come up to me. Learning them was exciting, and I can affirm without doubt that I learned a lot. As usual in these studies, a lot of new doors opened to me.

The more detailed work with C++ allowed me to learn more specific features of this programming language and to appreciate how powerful it is despite of its complexity. And to realize, that I only have seen a fraction of its possibilities, being the same with OpenGL.

It is said that you learn more from bad experiences. In my case I found several points to improve, related to improve the communication with the client and improve the project planning, among others.

The collaboration with Konrad Höpli was again a pleasure, and I have not enough words to thank Prof. Augenstein for his support.

Appendix D Legal

D.1 Declaration of Originality

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

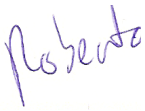
FHO Fachhochschule Ostschweiz

Declaration of originality

We hereby declare,

- That this project, including thesis, all other documents, and source code, are results of our own work and that we have not received any assistance other than what has been specified in the project specification, or what has been agreed upon in writing with the examiner,
- That we have referenced all source material according to generally accepted scientific citation guidelines,
- And that we have not used any resources protected under copyright law without appropriate permission.

Rapperswil, 16.06.2017



Roberto Cuervo Alvarez

Konrad Höpli

1

D.2 Copyright and Usage Rights Agreement

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Copyright and usage rights agreement

1. Object of agreement

With this agreement, the copyright, usage rights and future developments of the deliverables coming from the Bachelor Thesis “Jaw Viewer” of Roberto Cuervo Alvarez and Konrad Höpli under the supervision of Oliver Augenstein will be governed.

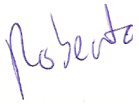
2. Copyright

Copyright is retained by the individual authors

3. Usage rights

The results of the term project can be used and developed by the students, the HSR and by the Universität Zürich Zentrum für Zahnmedizin after Bachelor Thesis conclusion.

Rapperswil, 16.06.17



The Students

Rapperswil, ...

.....
The Advisor

Rapperswil, ...

.....
The Cours Director

D.3 Publication Consent Form

Consent form for the publication in eprints.hsr.ch

○ Term Project

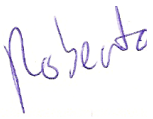
X Bachelor Thesis

Term Project Title:	Jaw Viewer
Team:	Roberto Cuervo-Alvarez & Konrad Höpli
Advisor:	Prof. Oliver Augenstein

We agree with the publication of our term project, as there is no arrangement of confidentiality signed.

Within 14 days after the grade's announcement we have the possibility of elevating an objection and retire the publication consent. In this case only the abstract will be published.

Rapperswil, 16.06.2017

Names	Signature
Roberto Cuervo Alvarez	
Konrad Höpli	

Appendix E Mathematical Demonstrations

E.1 Movement in R3

Property of Prof. Oliver Augenstein. This is a scanned pdf, therefore the design might be inconsistent with the other parts of the thesis.

1 Aufgabe

Gesucht ist die Formel für eine Bewegung

$$f(x) = Ox + t$$

die die nicht auf einer Gerade liegenden Punkte $a, b, c \in \mathbb{R}^3$ in die Punkte $a', b', c' \in \mathbb{R}^3$ überführt, wobei die Abstände der Punkte a, b, c identisch mit den Abständen der Punkte a', b', c' sein sollen.

1.1 Freiheitsgradbilanz

Die Freiheitsgradbilanz zeigt, dass eine solche Bewegung eindeutig festgelegt ist:

$3 \cdot 3 - 3$ Freiheitsgrade (nach Berücksichtigung der Constraints).

$3 + 3$ freie Parameter in der Funktion f (orthogonale Matrix ist durch drei Eulerwinkel festgelegt, die Translation durch die drei Koordinaten).

1.2 Benennung der System

Wir bezeichnen die Variablen im abgebildeten System wie die Variablen des Ausgangssystem ergänzt um einen Strich. Es gilt also

$$x' = f(x) \tag{1}$$

Zu beachten ist, dass wir zwischen Ortskoordinaten, die sich nach 1 transformieren, und Richtungsvektoren, die sich nach

$$v' = Ov$$

transformieren unterscheiden müssen.

Dabei sind Richtungskoordinaten immer Differenzen zweier Ortskoordinaten (oder Summen davon, da Vektorraum) und Ortskoordinaten können durch Addition eines Richtungsvektors in eine neue Ortskoordinate überführt werden.

Beispiel: Seien x, y Ortskoordinaten und

$$u = x - y$$

ein Richtungsvektor. Dann gilt

$$\begin{aligned} u' &= x' - y' = f(x) - f(y) = (Ox + t) - (Oy + t) \\ &= Ox - Oy = O(x - y) = Ou \end{aligned}$$

und

$$\begin{aligned} y &= x + u \\ y' &= x' + u' = f(x) + Ou = Ox + t + Ou = O(x + u) + t \\ &= f(y) \end{aligned}$$

1.3 Hilfsgrössen:

Wir definieren den Schwerpunkt

$$s = \frac{a+b+c}{3}$$

Dieser transformiert als Ortsvektor, denn

$$\begin{aligned} s' &= \frac{a'+b'+c'}{3} = \frac{Oa+t+Ob+t+Oc+t}{3} \\ &= \frac{Oa+Ob+Oc}{3} + t = Os + t = f(s) \end{aligned}$$

Ausserdem definieren wir die dem Transformationsgesetz $w' = Ow$ folgenden Richtungsvektoren

$$\begin{aligned} u &= \frac{b-a}{|b-a|} \\ v &= \frac{c-a}{|c-a|} \\ k &= \frac{u+v}{|u+v|} \\ l &= \frac{u-v}{|u-v|} \\ m &= k \times l \end{aligned}$$

sowie die entsprechenden gestrichenen Grössen.

1.4 Die Transformation

Nach Konstruktion sind k, l, m und die entsprechenden gestrichenen Grössen jeweils eine Orthonormalbasis die Transformation

$$Ow = k'(k,w) + l'(l,w) + m'(m,w)$$

ist damit orthogonal und es gilt

$$\begin{aligned} Ok &= k' \\ Ol &= l' \\ Om &= m' \end{aligned}$$

Die Transformation O entspricht damit der gesuchten Transformationsmatrix. Die Matrix-Komponenten dieser Matrix erhält man, indem man für w die Einheitsvektoren der Standardbasis einsetzt.

Definieren wir nun

$$f(x) = k'(k, x-s) + l'(l, x-s) + m'(m, x-s) + s'$$

so hat f die gesuchte Form und es gilt

$$f(s) \quad = \quad s'$$

Da alle Richtungsvektoren und ein Punkt korrekt transformiert werden, haben wir die gesuchte Funktion gefunden. So gibt es z.b. für a zwei Skalare λ, μ mit

$$a \quad = \quad s + \lambda k + \mu l$$

und es gilt

$$\begin{aligned} a' &= s' + \lambda k' + \mu l' = f(s) + \lambda Ok + \mu Ol \\ &= O(s - s') + s' + \lambda Ok + \mu Ol \\ &= O(s + \lambda k + \mu l - s') + s' \\ &= f(s + \lambda k + \mu l) = f(a) \end{aligned}$$

1.5 Interpretation

Die Vektoren u, v lässt sich mit einer Raute mit Zentrum s in Verbindung bringen. Dasselbe gilt im gestrichenen System. Die Transformation $\tilde{f}$ lässt sich unabhängig davon definieren, ob die Abstands-Constraints erfüllt sind, d.h. unabhängig davon, ob es überhaupt einen Bewegung gibt, die die alten in die neuen Punkte überführt. Die Konstruktion führt in so einem Fall zu einer Bewegung, die das Zentrum der alten Raute, auf das Zentrum der neuen Raute verschiebt und so dreht, dass die gestrichenen und die transformierten Rautendiagonalen übereinander zu liegen kommen. Gilt der Längen-Constraint, so sind die Rauten kongruent und $\tilde{f}$ ist die gesuchte Bewegung. Andernfalls sind die Bildpunkte nicht identisch mit den Punkten des gestrichenen Systems. Der Schwepunkt wird aber weiterhin auf den Schwerpunkt abgebildet, aber die Bildpunkt weichen von den gestrichen Punkten umso stärker ab, je grösser die Abweichung des Längen-Constraints ist.

Table of Images

List of Tables

Listings

Glossary

Anatomy Object

Is the real physical object corresponding to a Mesh, is a body part, in this context a jaw bone. Its description is contained in STL files and its software abstraction is a Mesh. 11, 14, 15, 17, 31, 35, 36, 44, 45, 62, 64

Figure 30: A Mesh, STL File or Anatomy Object

ARB

Architecture Review Board 19, *Glossary:* OpenGL Architecture Review Board

Calibration Data

Calibration data from the Zeiss Cameras ... TODO complete 14

Camera LEDs

The camera leds (Figure 31) are the graphical representation on screen of the real camera leds employed for the movement recording. 32

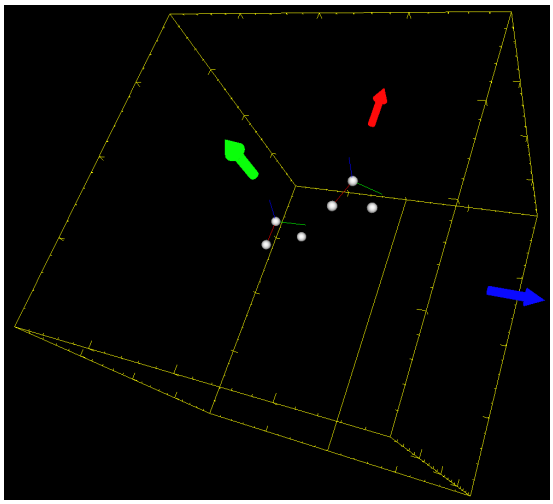


Figure 31: Camera Leds in Optis

Clip Space

Positions of Vertices after projection into a nonlinear homogeneous coordinate. [2] 20, 24

Fragment shader

A Shader that executes once per fragment and generally computes the final color of that fragment.[2] 20–22, 28, 64

Frame

Almost equivalent to the definition of a film frame. In the *Jaw Viewer* (Listing 13), a frame contains the data necessary for displaying an object in an exact position in an exact point in time. Like in a movie, several frames build the illusion of movement. 33, 36, 42

Frustum

A pyramid-shaped viewing volume that creates a perspective view. (Near objects are large; far objects are small). [2] 25

Geometry shader

A Shader that executes once per primitive, having access to all vertices making up that primitive.[2] 20, 21, 64

GLSL

OpenGL Shading Language 22, *Glossary:* OpenGL Shading Language

Khronos Group

The Khronos Group was founded in 2000 to provide a structure for key industry players to cooperate in the creation of open standards that deliver on the promise of cross-platform technology. Today, Khronos is a not for profit, member-funded consortium dedicated to the creation of royalty-free open standards for graphics, parallel computing, vision processing, and dynamic media on a wide variety of platforms from the desktop to embedded and safety critical devices. Khronos APIs are key technologies in their respective markets, such as Vulkan and OpenGL in graphics and gaming, WebGL in 3D web graphics, and OpenVX and OpenCL in embedded vision and compute [23]. 63

Mesh

A mesh is a software abstraction (a class) used for the graphical representation of coordinate collections or Vertex and can be static or animated. In the *Jaw Viewer*, a mesh represents an Anatomy Object. Each Mesh class instance has the logic which allow rendering itself. 17, 32, 40, 41, 61, 64

Model Space

Positions relative to a local origin. Also sometimes known as *Object Space*. [2] 19, 24, 62

Motion Movement file

See 5.3.3.1 MVM files 11, 62

MVM

Motion Movement file (inaccurate) 11, 12, 14, 17, 32, 34, 35, 44, *Glossary:* Motion Movement file

NDC

Normalized Device Coordinates 24, 25, 62, *Glossary:* Normalized Device Coordinates

Normalized Device Coordinates

The NDC [3] are in a small space where x, y and z values vary from -1.0 to 1.0. Any coordinates that fall outside this range will be discarded/clipped and won't be visible on your screen. Below you can see the triangle we specified within normalized device coordinates (ignoring the z axis): Unlike usual screen coordinates the

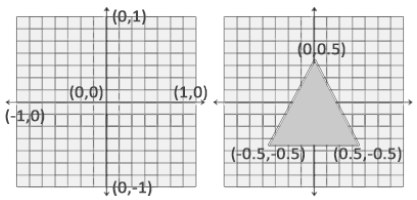


Figure 32: Normalized Device Coordinates [3]

positive y-axis points in the up-direction and the $(0,0)$ coordinates are at the center of the graph, instead of top-left. Eventually you want all the (transformed) coordinates to end up in this coordinate space, otherwise they won't be visible. 20, 24, 25, 62

Object Space

Positions relative to a local origin. Also sometimes known as *Model Space*. [2] 19, 24, 62

OpenGL

OpenGL [24] is the premier environment for developing portable, interactive 2D and 3D graphics applications. Since its introduction in 1992, OpenGL has become the industry's most widely used and supported 2D and 3D graphics application programming interface (API), bringing thousands of applications to a wide variety of computer platforms. OpenGL fosters innovation and speeds application development by incorporating a broad set of rendering, texture mapping, special effects, and other powerful visualization functions. Developers can leverage the power of OpenGL across all popular desktop and workstation platforms, ensuring wide application deployment. 9, 13, 16–20, 24, 27, 28, 31, 32, 43, 45, 47

OpenGL Architecture Review Board

The committee body [2] consisting of three-dimensional graphics hardware vendors (such as Compaq, DEC, IBM, Intel, Microsoft, Hewlett-Packard, Sun Microsystems or Evans & Sutherland), previously charged with maintaining the OpenGL specification. This function has since been assumed by the Khronos Group. 19, 61

OpenGL Shading Language

A high-level C-like shading language. [2] 18, 21, 43, 62

Optis

Proprietary software of the Clinic of Masticatory Disorders used to record in real time the jaw movement of the patient. 9, 11, 33, 45, 58, 61, 63

Phong Lighting Model

"One of the most common lighting models is the Phong lighting model. It works on a simple principle, which is that objects have three material properties: ambient, diffuse, and specular reflectivity. These properties are assigned color values, with brighter colors representing a higher amount of reflectivity. Light sources have these same three properties and are again assigned color values that represent the brightness of the light. The final calculated color value is then the sum of the lighting and material interactions of these three properties [2]" 28

Primitive

A group [2] of one or more Vertices formed by OpenGL into a geometric shape such as a line, point, or triangle. All objects and scenes are composed of various combinations of primitives. Everything you see rendered on the screen is a collection of primitives. 19, 20, 64

Reference Cube

As its name indicates, it's a cube drawn around the scene centroid for better appreciation of the scale. Each cube side has a length of 327.68 mm, and it corresponds with the size of the reference cube in the Optis software. 32

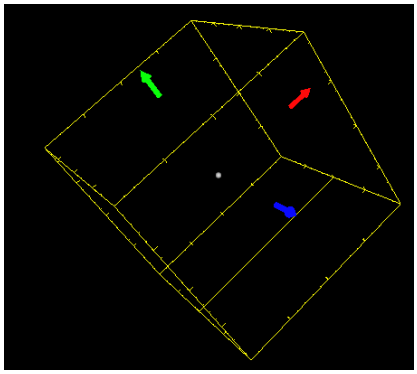


Figure 33: Reference Cube in Optis

Reference Sphere

Reference Sphere used for... TODO complete 14

scene

"Scenes contain the objects of your game. They can be used to create a main menu, individual levels, and anything else. Think of each unique Scene file as a unique level. In each Scene, you will place your environments, obstacles, and decorations, essentially designing and building your game in pieces." As described in the Unity Documentation [25] 32

Shader

A small program that is executed by the graphics hardware, often in parallel, to operate on individual Vertices or pixels [2]. There are Vertex shaders, Fragment shaders and Geometry shaders 19–24, 36, 61, 62, 64

Stereo Lithography Files

STL files describe the surface geometry of a three-dimensional object without any representation of color, texture or other common CAD model attributes. An STL file describes a raw unstructured triangulated surface by the unit normal and vertices of the triangles using a three-dimensional Cartesian coordinate system. STL coordinates must be positive numbers, there is no scale information, and the units are arbitrary [14]. The STL format specifies both ASCII and binary representations. Binary files are more common, since they are more compact.
In the *Jaw Viewer*, each STL-File represents one Mesh or one Anatomy Object. 11, 31, 64

STL

Stereo Lithography Files 11, 12, 14, 15, 17, 31, 44, 61, *Glossary:* Stereo Lithography Files

Tessellation

Is the process of breaking a high-order primitive (which is known as a *patch* in OpenGL) into many smaller, simpler primitives such as triangles for rendering. OpenGL includes a fixed-function, configurable tessellation engine that is able to break up quadrilaterals, triangles, and lines into a potentially large number of smaller points, lines, or triangles that can be directly consumed by the normal rasterization hardware further down the 5.2.2 Graphics Pipeline. [2] 20, 64

Tessellation Control shader

This Shader takes its input from the vertex-shader and is primarily responsible for two things: the determination of the level of Tessellation that will be sent to the tessellation engine, and the generation of data that will be sent to the tessellation evaluation shader that is run after tessellation has occurred. [2] 20, 21

TMJViewer

Proprietary software of the Clinic of Masticatory Disorders used to display the jaw movement of the patient together with its Anatomy Objects. 9, 11, 45

Vertex

A point in space is both a vertex and a vector [2]. Except when used for point and line primitives, it also defines the point at which two edges of a polygon meet. 17, 19, 20, 24, 27, 31, 61–64

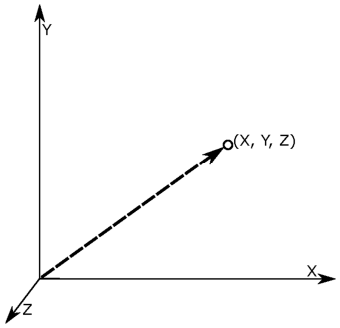


Figure 34: Vertex [2]

Vertex shader

A Shader that executes once per incoming vertex. [2] 20–22, 64

View Space

Positions relative to the viewer. Also sometimes known as *camera* or *eye space*. [2] 20, 24, 27

Window Space

Or *Screen space*. Positions of Vertices in pixels, relative to the origin of the window. [2] 20

World Space

This is where coordinates are stored relative to a fixed, global origin. [2] 20, 24

References

[1] *Visual Studio Team Services*. https://www.visualstudio.com/team-services/ Visited 17-02-2017.

[2] Graham Sellers, Richard S. Wright Jr., and Nicholas Haemel. *OpenGL Superbible: Comprehensive Tutorial and Reference*. Pearson Education, 7 edition, 7 2015.

[3] Joey de Vries. Learn OpenGL. https://learnopengl.com/ Visited 20-02-2017.

[4] *Alpha Test in OpenGL*. https://www.khronos.org/opengl/wiki/Transparency_Sorting#Alpha_test Visited 28-05-2017.

[5] *Blending in OpenGL*. https://www.khronos.org/opengl/wiki/Blending Visited 25-05-2017.

[6] David Wolff. *OpenGL 4 Shading Language Cookbook*. Packt Publishing, 2 edition, 12 2013.

[7] *3D Graphics with OpenGL By Examples*. https://www3.ntu.edu.sg/home/ehchua/programming/opengl/CG_Examples.html Visited 28-05-2017.

[8] *Real 3D Tutorials*. http://www.real3dtutorials.com/tut00002.php Visited 26-05-2017.

[9] *Gram-Schmidt Process*. http://mathworld.wolfram.com/Gram-SchmidtOrthonormalization.html Visited 04-05-2017.

[10] Joey de Vries. Learn OpenGL: Camera. https://learnopengl.com/#!Getting-started/Camera Visited 04-06-2017.

[11] *GLM: C++ mathematics library for graphics software*. http://glm.g-truc.net/0.9.8/index.html Visited 18-02-2017.

[12] Joey de Vries. Learn OpenGL: Lighting. https://learnopengl.com/#!Lighting/Basic-Lighting Visited 04-06-2017.

[13] *Normal Vector*. http://mathworld.wolfram.com/NormalVector.html Visited 05-06-2017.

[14] *STL file format*. https://en.wikipedia.org/wiki/STL_(file_format) Visited 10-06-2017.

[15] *Open Asset Import Library*. http://assimp.sourceforge.net/index.html Visited 10-03-2017.

[16] *GLEW: C/C++ extension loading library*. http://glew.sourceforge.net/ Visited 18-02-2017.

[17] AlDanial. *Count Lines of Code*. https://github.com/AlDanial/cloc Visited 30-05-2017.

[18] *GLFW: Multi-platform C++ for OpenGL*. http://www.glfw.org/index.html Visited 16-02-2017.

[19] *Jsoncpp: Json support in C++*. https://github.com/open-source-parsers/jsoncpp/wiki Visited 05-05-2017.

[20] *Cute Test Framework*. http://cute-test.com/projects/cute Visited 27-02-2017.

[21] *Scrum*. https://www.scrum.org/resources/what-is-scrum Visited 03-05-2017.

[22] *What is a Sprint in Scrum?* https://www.scrum.org/resources/what-is-a-sprint-in-scrum?gclid=COfZq9qwotQCFZRsGwod82AG_g Visited 03-05-2017.

[23] *Khronos Group*. https://www.khronos.org/about Visited 06-06-2017.

[24] *OpenGL*. https://www.opengl.org/ Visited 27-05-2017.

[25] *Unity-Manual: Scenes*. https://docs.unity3d.com/Manual/CreatingScenes.html Visited 12-03-2017.