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

1 Abstract

1.1 Introduction

In order to elaborate accurate diagnostics, the Clinic of Masticatory Disorders of the Zurich University uses MRI files and a self developed 3D camera system in order to record the mastication movement of its patients. The data obtained from these tools must be processed with different applications until it can be displayed for diagnostics. This process is tedious, error-prone and doesn't allow the observation of the mastication movement in realtime. A new application should just import the necessary data or data-stream and be capable of displaying the desired movement in realtime or not.

1.2 Approach

TODO

As a first step, we decided on using the Vuforia augmented reality SDK for the game engine Unity. This way, we had the option to use the additional features coming with both Unity and the SDK made therefore. Especially the various image effects and filters provided by Unity looked very promising for our intentions. The second step was developing a first prototype of an AR app, which simply used the 'blur' component provided by Unity on the running smartphone camera and thereby proofed the plausibility of the project idea. We then went on to attempt implementing the other visual effects described by ASN. Based on the research nature of this project, the whole development process as well as the information gathered on the topic of both Unity and AR capabilities is supposed to be documented well. In a manner to not just give a good insight on the project itself, but the topic and used tools as well. It is aimed at both potential future contributors to this project as well as developers interested in testing the water of augmented reality with the tools used here.

1.3 Results

TODO

This project resulted in a functional app which can be used by the attendees of ASN presentations to simulate the visual experience of being drunk. It serves as a proof of concept for the initially uncertain idea to simulate the visual effects caused by alcohol in AR. The app also identified the currently still severe limitations of the processing power of smartphones regarding the used technologies. The documentation serves as introduction to the world of AR and Unity while also giving insight on the project itself, the encountered limitations and possible extensions or future projects this could lead to.

2 Terms of Reference

Arbeitsdetails https://avt.hsr.ch/Pages/DetailAnsicht.aspx?Arbeit=21563

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

TODO

3.3 Approach

TODO

3.4 Planning

The *Jaw Viewer* project is run following the Agile [1] principles, with Scrum [2] as development strategy. The 18 sprints [3] have a duration of a week, with 3 working days each.

3.5 Technologies

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

OpenGL Employed for graphic processing, OpenGL is an obligatory technology as mentioned in the Terms of Reference 2. For more detailed information check the glossary OpenGL and the 5.2 OpenGL section

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 sharp: As Konrad Höpli have experience with C sharp, and in order to comply with the 4.2.2 FR 10, a modern GUI, we chose C sharp 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.

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** [4] 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 Git.

3.6 Risks

3.7 Effort

3.8 Results

4 Analysis

4.1 Introduction

Preliminary remark: this document is the Bachelor Thesis of Roberto Cuervo Alvarez and Konrad Höepli, created at the Computer Science Department of the University of Applied Sciences of Rapperswil. This work was developed under the supervision of Prof. Oliver Augenstein. Both students worked equally in its contents.

Vision: in the course of this project an application capable of importing Stereo Lithography Files (STL) files (or Anatomy Objects), movement data in form of Motion Movement files (MVM) 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.

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. Note: due to missing information from the project partner it was not possible to develop the Real-Time Modus. Explanations follow.

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

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

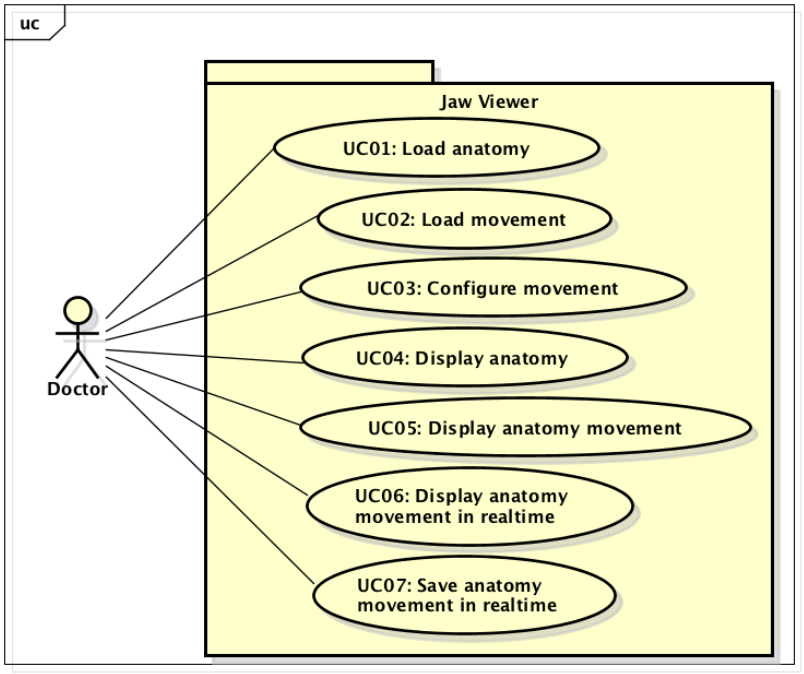


Figure 1: Use Cases

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.

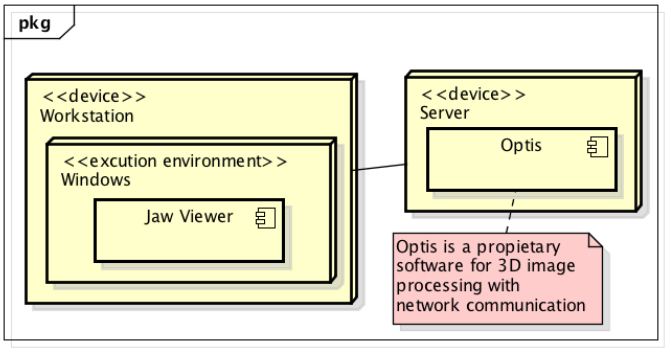


Figure 2: Deployment Diagram

4.2.6.2 Component Diagram

The Jaw Viewer is built with three components:

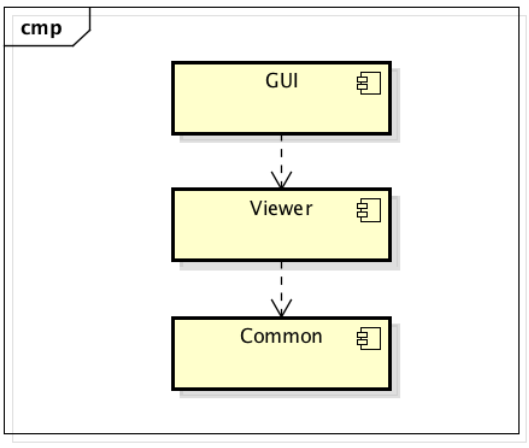


Figure 3: Component Diagram

4.2.6.3 Domain Model

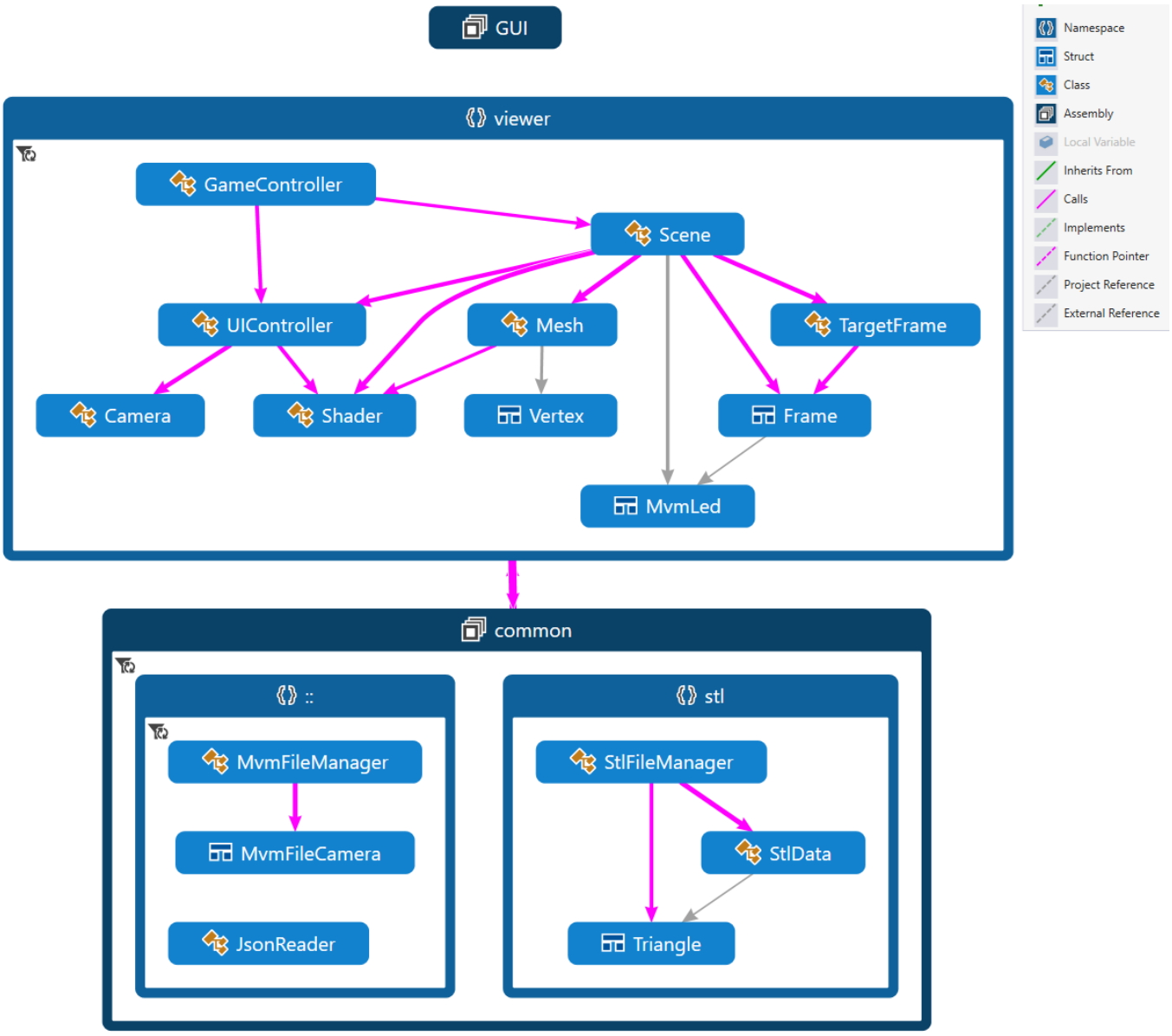


Figure 4: Domain Model

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.

5 Technical Report

5.1 Introduction and Summary

TODO COMPLETE

5.2 OpenGL

5.2.1 Introduction

After the glossary definition of OpenGL we can say that OpenGL is a 2D/3D Graphics API which provides an abstraction layer between the application and the underlying graphics subsystem. The commands [5] 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 such that a later stage of one command might run concurrently with an earlier stage of another command. Furthermore, computer graphics generally consists of many repetitions of very similar tasks, and these tasks are usually independent of one another. That is, the result of coloring one pixel doesn't depend on any other. Just as a car plant can build multiple cars simultaneously, so 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" [5].

This *abstraction layer* [5] allows the application to not need to know who made the graphics processor (or graphics processing unit [GPU]), how it works, or how well it performs. Certainly it is possible to determine this information, but the point is that applications don't need to.

5.2.1.1 Core and Compatibility Profiles

Since the first OpenGL version in 1992 [5] the graphics hardware and software have evolved fast and constantly. Over time, the price of graphics hardware came down, performance went up, and new features and capabilities 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 another 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 other version of the specification, the compatibility profile. In addition, 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 are published (at the time of writing 4.5). The answer to the logical question of why do we use OpenGL 3.3 when OpenGL 4.5 is out, is that all future versions of OpenGL starting from 3.3 basically add extra useful features to OpenGL without changing OpenGL's core mechanics; the newer versions just introduce slightly more efficient or more useful ways to accomplish the same tasks. The result is that all concepts and techniques remain the same over the modern OpenGL versions so it is perfectly valid to learn and use OpenGL 3.3 [6].

5.2.1.2 Recommended definitions

For a better comprehension of the following sections, we recommend to consult the next 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 GPUs consist of large numbers of small programmable processors called shader cores that 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" [5]. Figure 5 shows a simplified schematic of the graphics pipeline.

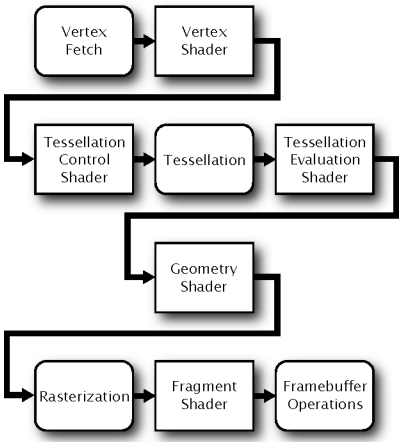


Figure 5: Simplified graphics pipeline [5]

In the Figure, the boxes with rounded corners are considered fixed-function stages, whereas the boxes with square corners are programmable, wohich means that they execute Shaders that you programmed.

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

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

In the primitive assembly stage or Tessellation the Tessellation Control shader takes as input all the Vertices from the vertex shader that form a Primitive and assembles all the point(s) in the Primitive shape given; in this case a triangle.

The output of the primitive assembly stage is passed to the Geometry shader. The Geometry shader takes as input a collection of vertices that form a Primitive and has the ability to generate other shapes by emitting new vertices to form new (or 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, resulting in fragments for the Fragment shader to use. Before the fragment shaders runs, clipping is performed. Clipping discards all fragments that are outside your view, increasing performance

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

After all the corresponding color values have been determined, the final object will then pass through one more stage that we call the alpha test [7] and blending stage [8]. 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 accordingly. So even if a pixel output color is calculated in the fragment shader, the final pixel color could still be something entirely different when rendering multiple triangles.

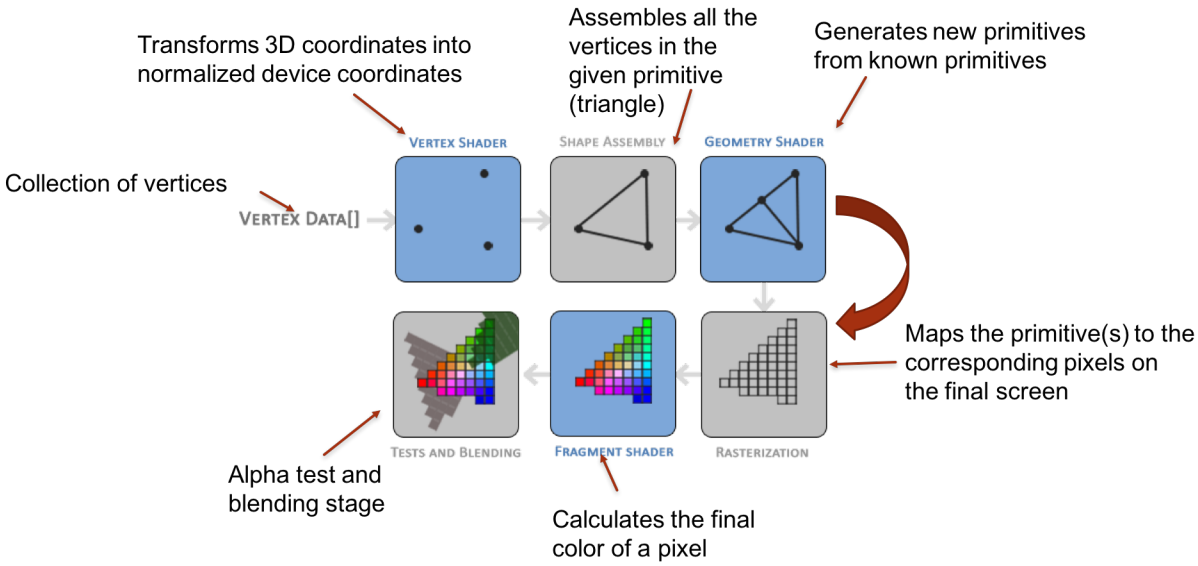
In the Figure 6 below we can now observe the graphics pipeline with a short description of what each Shader do at each corresponding state:

5.2.3 Shaders

"Shaders are little programs that rest on the 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." [9]

Each shader executes within a different section of the OpenGL pipeline. Each shader runs on the GPU, and as the name implies, (typically) implement the algorithms related to the lighting and shading effects of an image. However,

Figure 6: Extended graphics pipeline [6]



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 OpenGL Shading Language, or GLSL. This language has its origins in C, but has been modified over time to make it better suited to running 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, and then multiple shader objects can 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, and 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* [9].

Shaders always begin wit a version declaration, followed by a list of input and output variables, uniforms and the main function [6]:

Listing 1: Vertex Shader syntax [9]

```
#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 to declare global variables with the `in` and `out` storage qualifiers [5].

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* [9].

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. [6]

5.2.6 Ins and Outs

In order to send data from one shader to the other it's 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 color is set to a dark-red color in the vertex shader, the resulting fragments should be dark-red as well. Without output color specification in the fragment shader OpenGL will render the object black or white [6].

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 GPU, but uniforms are slightly different compared to vertex attributes [6]

- 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're either 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's necessary first to find the index / location of the uniform attribute in the shader in order to update its values. As example, we change the color of a triangle gradually over time [6]:

Listing 5: Using uniforms

```
1
2  GLfloat timeValue = glfwGetTime();  // retrieve the running time in seconds
3  GLfloat greenValue = (sin(timeValue)/2) + 0.5;//vary the color in the range of 0.0 - 1.0
4
5  // query for the location of the ourColor uniform
6  GLint vertexColorLocation = glGetUniformLocation(shaderProgram, "ourColor");
7  glUseProgram(shaderProgram); //updating a uniform requires to first use the program
8  glUniform4f(vertexColorLocation, 0.0f, greenValue, 0.0f, 1.0f);// set the uniform value
```

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

Listing 6: Using uniforms for rendering

```
1  //Changing the color gradually updating the uniform each render iteration
2  while(!glfwWindowShouldClose(window)) {
3      // Check and call events
4      glfwPollEvents();
5
6      // Render
7      // Clear the colorbuffer
8      glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
9      glClear(GL_COLOR_BUFFER_BIT);
10
11     // Be sure to activate the shader
12     glUseProgram(shaderProgram);
13
14     // Update the uniform color
15     GLfloat timeValue = glfwGetTime();
16     GLfloat greenValue = (sin(timeValue) / 2) + 0.5;
17     GLint vertexColorLocation = glGetUniformLocation(shaderProgram, "ourColor");
18     glUniform4f(vertexColorLocation, 0.0f, greenValue, 0.0f, 1.0f);
19
20     // Now draw the triangle
21     glBindVertexArray(VAO);
22     glDrawArrays(GL_TRIANGLES, 0, 3);
23     glBindVertexArray(0);
24 }
```

Resulting in:

Figure 7: Uniform usage result [6]

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, that we want to become visible, to be in Normalized Device Coordinates after each vertex shader run; coordinates outside this range will not be visible. These NDC coordinates are then given to the rasterizer to transform them to 2D coordinates/pixels on your screen [6].

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 usual transformations are modeling, viewing, 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 center of gravity or one of the vertices (corners) of the cube, (0,0,0) for example. The origin is often the point about which you might rotate the model to place it into a new orientation [5].

5.2.11 World Space and Model Matrix

"This is where coordinates are stored relative to a fixed, global origin" [5]. This is the coordinate space where you want your objects transformed to in such a way that they're all scattered around the place (preferably in a realistic fashion). The coordinates of your object are transformed from model to world space; this is accomplished with 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" [6].

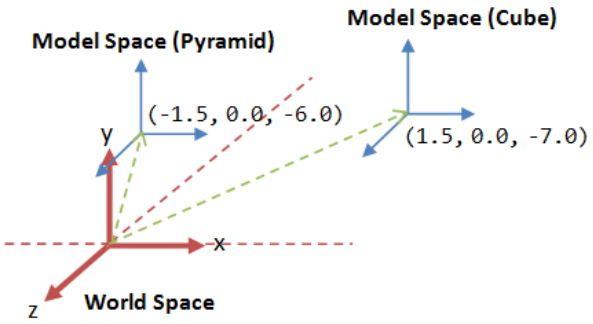


Figure 8: Model and World Spaces [10]

5.2.12 View Space and View Matrix

Or *View Coordinates*, or most usually the *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 [5].

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 a **view matrix** that transforms world coordinates to view space.

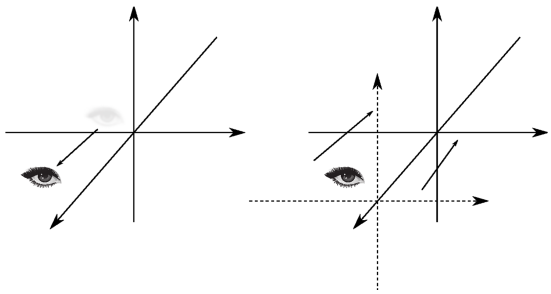


Figure 9: Two perspectives of View Coordinates [5]

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 the coordinates are displayed in the screen:

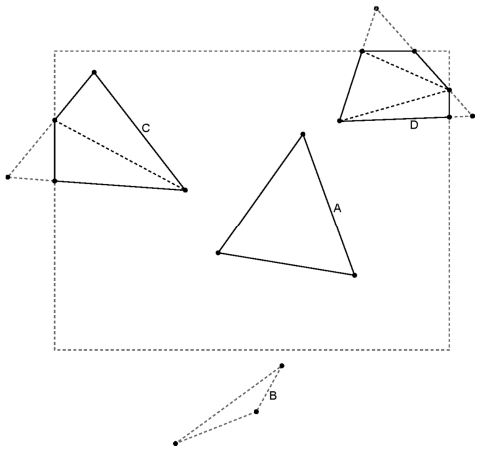


Figure 10: Clip Space [5]

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 [6].

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.

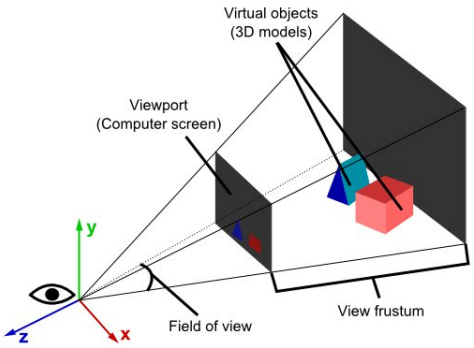


Figure 11: Frustum [11]

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" [5].

The projection matrix to transform view coordinates to clip coordinates can take two different forms, where each form defines its own unique frustum. It's 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 that defines the clipping space where each vertex outside this box is clipped. When creating an orthographic projection matrix we specify the width, height and length of the visible frustum. All the coordinates that end up inside this frustum after transforming them to clip space with the orthographic projection matrix won't be clipped. The frustum looks a bit like a container:

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 doesn't take perspective into account.

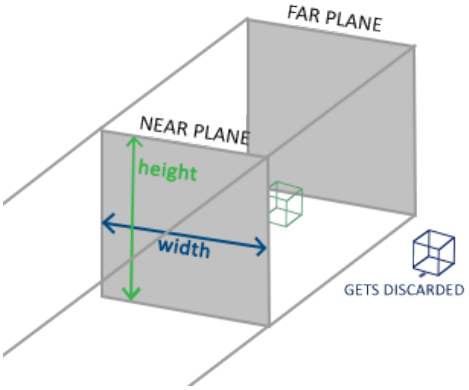


Figure 12: Orthographic frustum [6]

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:

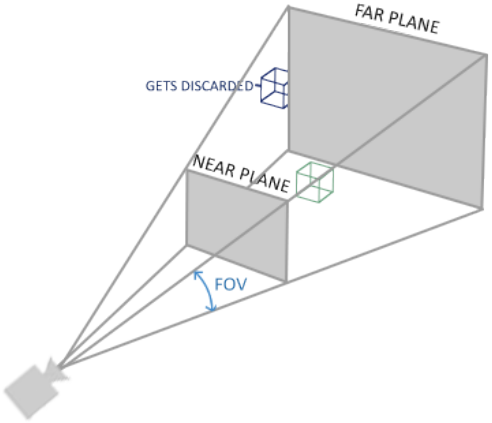


Figure 13: Perspective frustum [6]

A comparison of both perspectives:

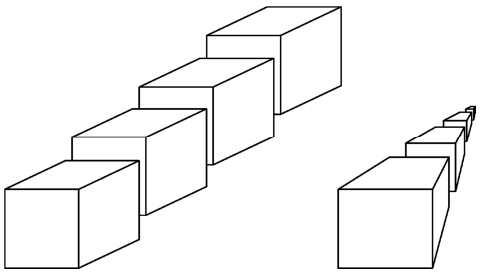


Figure 14: Orthographic and Perspective projections [5]

5.2.15 Spaces and Transformations summary

In the Figure 15 we can observe all the transformations or steps needed to display an object or model from its Vertices to the screen:

5.2.16 Camera

In OpenGL there is no concept of *camera* as in the video game world [12]. Usually a camera simulation is created "by moving all objects in the scene in the reverse direction", giving so the illusion that the observer is moving [6].

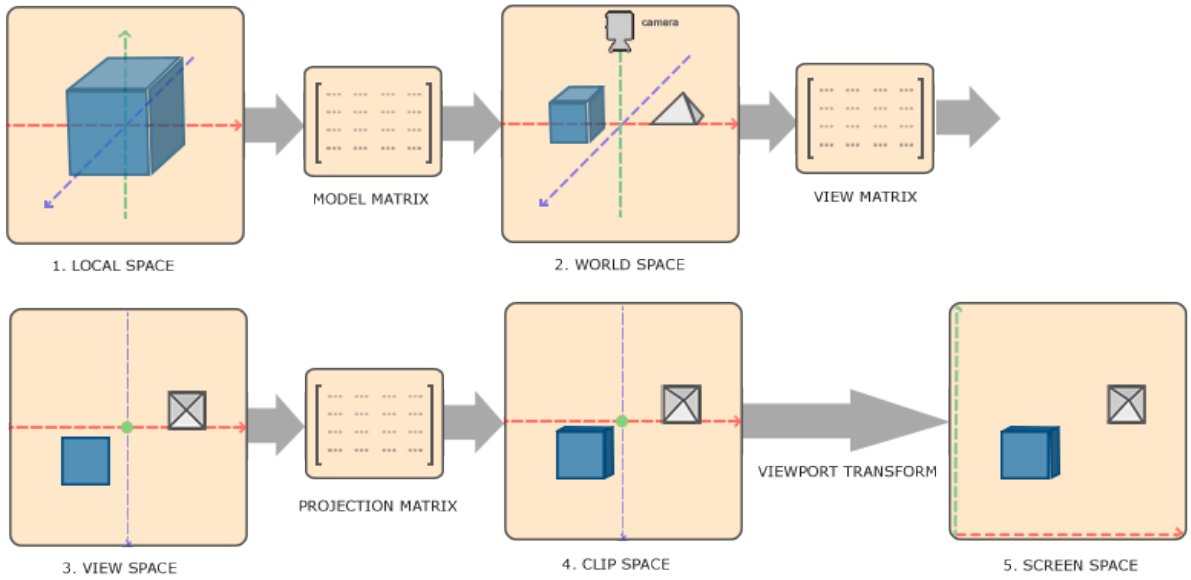


Figure 15: Coordinate systems [6]

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.
To define a camera its necessary:

- its position in world space
- the direction it's looking at
- a vector pointing to the right
- a vector pointing upwards from the camera

This creates a coordinate system with the camera's position as the origin:

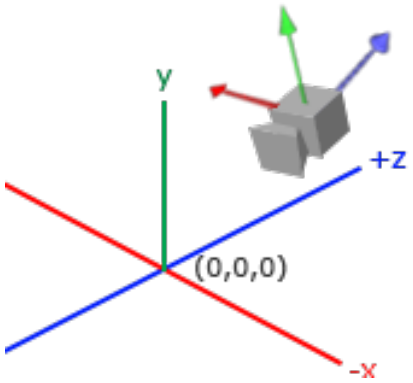


Figure 16: Camera as coordinate system [13]

5.2.16.1 Camera Position

"The camera position is basically a vector in world space that points to the camera's position" [13]. As you can observe in Figure 16, the positive $z-axis$ goes through the screen towards the user. That means, that if you want to move the camera backwards or forwards, the movement is along the $z-axis$.

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

Or a vector that represents the positive $x-axis$ of the camera space. To get the right vector we use a specify first an up vector that points upwards in world space $(0,1,1)$. Then we do a cross product on 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 in the positive x-axis's direction.

5.2.16.4 Camera Up Axis

Is the result of the cross product of the right and direction vector:

$$cameraUp = cameraRight \times cameraDirection$$

This process of forming the view / camera space coordinates system is known as the Gram-Schmidt process [14] in linear algebra .

5.2.16.5 LookAt Matrix

The LookAt matrix is defined with the 3 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" [13].

The GLM library [15] provides a `glm::lookAt` function. It's 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:

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" [16]. One of these models is the Phong Lighting Model. The Phong Model consists of 3 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" [5]. To simulate this an ambient lighting constant that always gives the objects some color is used. This constant is added to the final resulting color of the object's Fragment shader, thus making it look like there is always some scattered light even when there's not a direct light source.

5.2.17.2 Diffuse Lighting

"Diffuse light is the directional component of a light source" [5] 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.

In the Figure 17 we can see on the left a light source with a light ray pointing to the object. It's 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 $\overrightarrow{N}$ [17] to the fragment's surface. The angle between the two vectors can then easily be calculated with the dot product.

The resulting dot product thus returns a scalar that can be used to calculate the light's impact on the fragment's color, resulting in differently light fragments, based on their orientation towards the light.

Therefore, in order to calculate diffuse lighting is necessary [16]:

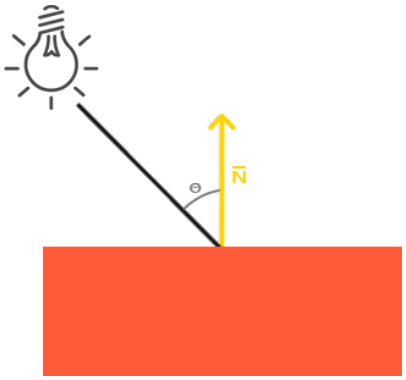


Figure 17: Diffuse Light [16]

- Normal vector $\overrightarrow{N}$: a vector perpendicular to the vertex' surface.
- The directed light ray: a direction vector that is the difference vector between the light's position and the fragment's position. To calculate this light ray we need the light's position vector and the fragment's position vector.

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" [5], 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"* [5]. Specular highlights are often more inclined to the color of the light than the color of the object.

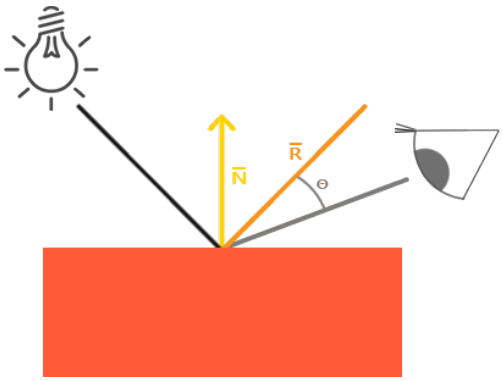


Figure 18: Specular Lighting [16]

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 and 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's intensity, multiply this with the light color and add this to the resulting ambient and diffuse components.[16]

In the Figure 19 you can see how the different light components look like:

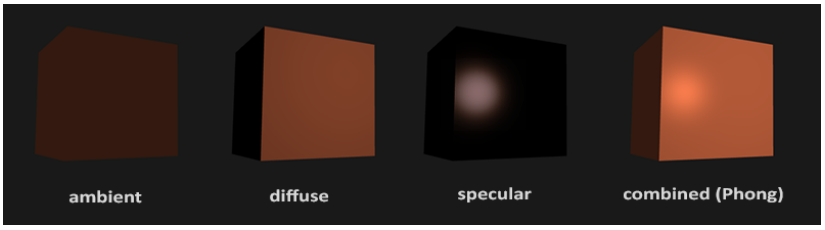


Figure 19: Lighting Components [16]

5.3 Implementation

After the necessary introduction to OpenGL we describe the functioning of the *Jaw Viewer*. In order to achieve this we will travel through all steps from the user input until the display of an object and its movement. TODO complete

5.3.1 Import Anatomy

The first goal in the project is 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 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 begin 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 20: STL file Ascii representation [18]

The file concludes with:

```
endsolid name
```

A **binary STL file** has an 80-character header. Following the header is a 4-byte unsigned integer indicating the number of triangular facets in the file. After that is 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" [18].

```
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 21: STL file Binary representation [18]

5.3.1.2 Importing STL files

The tutorial [6] we used as basis for the project employed the C++ library `assimp` [19] to import STL files. As `assimp` introduced a dependency to a data structure which did not fit well in ours, we decided not using `assimp` and implement our own STL file reader.

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 return 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
6       StlData(const std::string name) : name(name) { }
7   };
```

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 happen we must first go through the whole application starting process until arriving to this point.

We assume that 4.2.4.2 UC01: Load anatomy has already took 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 of the management of the application and is contained in the `viewer` package.

The `GameController` initializes OpenGL, GLEW [20], 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 are displayed on the screen. To these belong 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 and the Meshes as well. These last 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 moment, 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. These operations are executed on each Mesh object by the method call `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), &this->vertices[0],
8      glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, this->EBO);
9      glBufferData(GL_ELEMENT_ARRAY_BUFFER, this->indices.size() * sizeof(GLuint), &this->indic
10     glEnableVertexAttribArray(0);
11     glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), static_cast<GLvoid*>(null
12     glEnableVertexAttribArray(1);
13     glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), reinterpret_cast<GLvoid*>
14     glBindVertexArray(0);
15 }
```

The last step is done in the game loop. While running in the game loop, the `Scene` calls the `Mesh::Draw(const Shader shader` method again in each Mesh object, passing it the necessary `Shader`. So each Mesh object is finally displayed on the screen (Figure 22).

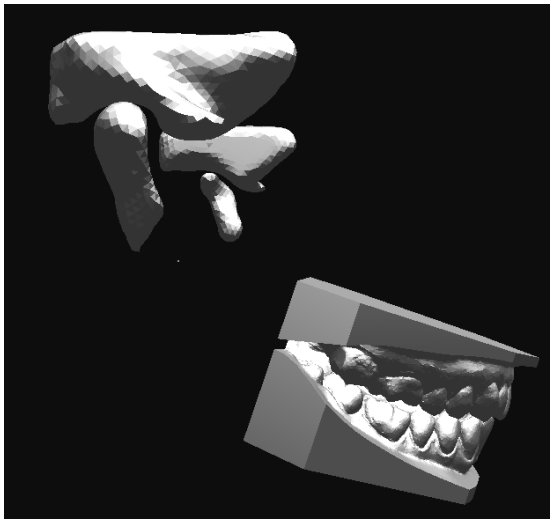


Figure 22: Anatomy Objects in the *Jaw Viewer*

5.3.3 Import movement or MVM files

We must here refer again to the Use Cases. 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 we explain how the MVM files are loaded we must first describe its structure and function.

5.3.3.1 MVM files

A MVM or motion movement file is a file generated by the Optis application. 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 attached to the patient's mandibular and are grouped in triangles. Each 3D camera covers one axis, and the coordinates of each camera (X, Y, Z) on one step combined represent one Frame. The Coordinates are calculated with luminosity readings from the cameras.

	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 23: MVM file structure

The Figure 23 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. The structure is repeated until file ends.

In a MVM file these structures are grouped by three, forming a Frame, and in a frame the 18 Leds form as well triangles (f.e. the triangle Leds 1, 3, 5). Each structure is separated from the next by an empty 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.

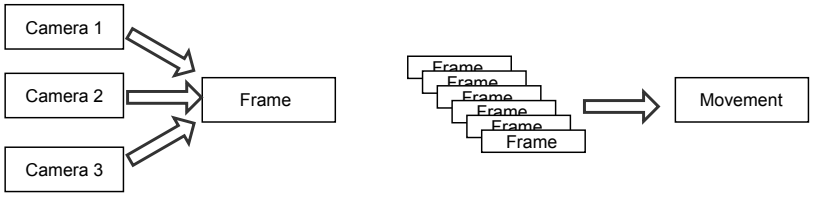


Figure 24: Frames and Movement

5.3.3.2 Reading MVM files

Once described the file structure, we continue with the loading or reading process. The MVM files are a proprietary format of the Clinic for Masticatory Disorders, and therefore there are no C++ library which supports parsing them. Because of this 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, a struct which corresponds to the MVM camera structure of Figure 23:

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 Mmv 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 returns 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);
```

```
16              this->frames.push_back(frame);
17          }
18          mvm_file.close();
19          return this->frames;
20      } catch //... error handling
```

The `MvmFileManager::getCameraData(...)` method extracts recursively the data from the `ifstream` to an instance of the `MvmFileCamera` struct, and translates the Led coordinates. This translation is necessary because in the MVM file the Led coordinates are saved in unsigned integers from 0 to 65535, and in the *Jaw Viewer* the coordinates range must be between -163.84 to 163.84 millimeter. The method `translateLedCoordinates` in Listing 11 executes this translation.

As already mentioned above, 3 `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 3 `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 reached the file end, the `MvmFileManager::getFramesFromMvmFile(...)` returns the filled `std::vector<Frame>`.

5.3.4 Display movement

The Use Cases UC03 and UC05 describe how the user configures the movement, and how then the motion of the Anatomy Objects is displayed. In this section we explain first the mathematical basis of the calculations needed to display the movement, and then how these calculations are applied.

5.3.4.1 Movement Calculations

Based on the mathematical analysis of Prof. Augenstein (see Appendix E.1 Movement in R3), we abstract the movement contained in the Frames imported from the MVM files (see 5.3.3.2 Reading MVM files) in form of a transformation matrix. We can then apply this matrix to any Anatomy Object and so move it on the screen.

In a first approach we use as reference coordinates the LED coordinates of the first LED triangle (which is theoretically static) formed by the LEDs 1, 2, and 3. This first LED triangle is contained in the first Frame extracted from the MVM file. 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.
For each transformation matrix the needed auxiliary quantities (Appendix E.1, 1.3):

Reference Direction Vectors k, m

Transformed Direction Vectors k', l', m'

Reference Centroid s

Transformed Centroid s'

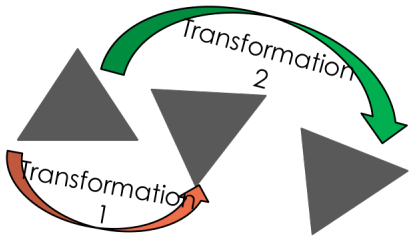


Figure 25: Movement transformation matrix

are calculated by means of the coordinates of the first LED triangle (1, 3, 5) of the first frame and the coordinates of the second LED triangle (7, 9, 11) of the remaining frames. The function (Appendix E.1, 1.4) allow us to calculate each component of the transformation matrix

$$\boxed{f(x) = k'(k, x-s) + l'(l, x-s) + m'(m, x-s) + s'}$$ 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 as in 5.3.2 Display Anatomy, the `GameController` initializes the `Scene` class and calls the `scene.init()` method.

The `init()` calls among others the `generateTargetFrames()` method, which accepts as parameter the path to the MVM file 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 locally the first Frame and calculates the transformation matrices by 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, storing these instances in a `std::vector<TargetFrame>` in the `Scene`.

At this time the `Scene` has already loaded the Anatomy Objects (STL files). Then, 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 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), and pass 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 vertices coordinates in the field of view and so generate a motion feeling.

5.3.4.3 Reverse Engineering

TODO complete

5.3.5 Perspective, Rotation

TODO check the math part, change the images for better ones (they can be computer generated oder by hand) and integrate the math part with the implementation

5.3.5.1 Page 1

aspect ratio = 1

$\alpha = fov$

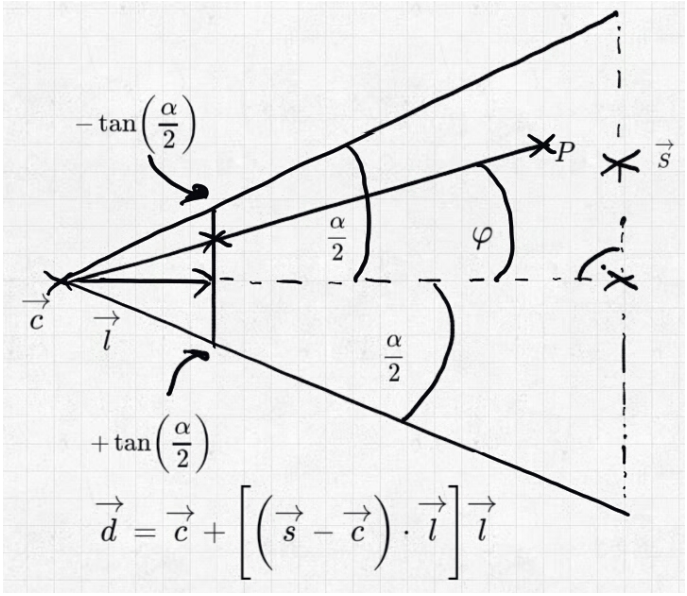
$\overrightarrow{s}$ = centroid position

$\overrightarrow{c}$ = camera position

$\overrightarrow{l}$ = look at position $|\overrightarrow{l}| = 1$

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

Aufsicht:



Display of point P on screen:

$$\vec{d} = \vec{c} + \left[\left(\vec{s} - \vec{c}\right) \cdot \vec{l}\right]\vec{l}$$

$X-$Coord x

$$\boxed{2\left(\frac{x - x_0}{screenwidth} - 0.5\right) \cdot tan\left(\frac{\alpha}{2}\right) = tan(\varphi)}$$

Where:
x_0 = left Pixel
w = screen width

5.3.5.2 Page 2 Zoom

Per scroll-wheel rotation:

Enlargement Factor v (scroll backwards)

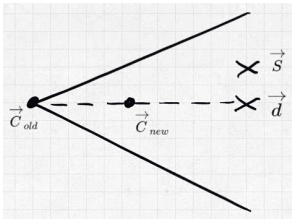
Enlargement Factor $\frac{1}{v}$ (scroll forwards)

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

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

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

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



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

Problem, if $\vec{c} = \vec{d}$ (correspond with infinite stark Enlargement)

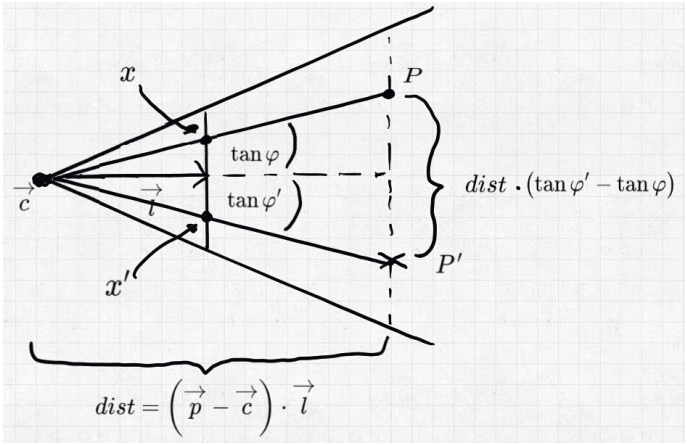
Adjustment of near and far plane not necessary.

5.3.5.3 Page 3 Shift Point

Shift Point with Coordinates x to x'

3a

Formal correct Shift of selected Point. This work only if a Point with $z > -1$ is selected. If the Point is on the Plane, $z = -1$, fall-back to 3b.



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

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

In vectorial form:

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

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

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

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

3b

3b has the advantage of working with all the points on the screen and not only for Points on the Mesh. It bases on a shift velocity independent of the selected point, velocity which is inverse proportional to the enlargement factor.

The selected point doesn't correspond exactly with the mouse pointer.

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

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

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

Problem, if $\overrightarrow{c}_{old}$ lays on the Centroid-plain (infinite Enlargement $\widehat{=}$ 0-shift factor)

This mouse-cursor problematic can be avoided when the selected pixel is marked in the model and the mouse cursor hidden. At operation's end the mouse cursor can be set back in the right position with `SetCursor`.

5.3.5.4 Page 4 Rotations

The rotation should be around the d axis or rotation center. There are different rotation concepts, f. e. arcball. The rotation concept explained here bases on a constant rotation velocity. The costs of this approach is that a point can not be fix connected to the mouse cursor. This can only be possible with low rotation velocity.

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 middle of the image

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

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

The rotation axis is thus:

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

Step 1

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

Step 2

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

Step 3

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

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

Problems: Adjustment of near and far plane necessary: yes

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

5.4 Testing

5.4.1 Cute Framework

After trying several unit test frameworks we decided to employ the Cute Framework [21], as we already have known Cute 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 this that there is no "Green Bar". On the contrary, as command line framework 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 on, as Cute is a "header library". It's only necessary to include the header files in the project.

The creation of a test suite is also 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 a class or struct. After, we register the test suite 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

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

glew [20] Cross-platform open-source C/C++ extension loading library. Run-time mechanisms for determining which OpenGL extensions are supported on the target platform

glm [15] Mathematics library for graphics software based on the OpenGL Shading Language specifications

jsoncpp [23] C++ library for `json` management

cute [21] C++ Testing Framework. Refer to 5.4 Testing

5.6 Code Statistics

TODO complete and update line codes with cloc

The code lines were counted with Cloc [24]. The lines quantity contains only self written code. External libraries were excluded.

Language	**Files**	**blank**	**comment**	**code**
C++	22	778	486	6264
C/C++ Header	29	140	145	672
GLSL	2	12	5	44
C#	14	92	316	1670
Total	67	1022	952	8650

Table 11: Code Statistics

5.7 Results

TODO complete

5.8 Conclusion

TODO complete

Appendices

Appendix A Project Planning

A.1 Development Strategy

TODO complete

- Scrum
- Sprints
- Sprints Duration
- How much Sprints
- Why Scrum

A.2 Risks

TODO complete

A.3 Effort

TODO complete
Worked Hours

- Expected
- Result

Appendix B Software Documentation

B.1 Installation

TODO complete

B.2 User Manual

TODO complete

Appendix C Field Report

C.1 Konrad Hoepli

TODO complete

C.2 Roberto Cuervo

TODO complete

Appendix D Legal

D.1 Declaration of Originality

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

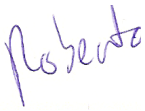
FHO Fachhochschule Ostschweiz

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.

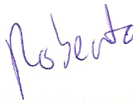
2. Copyright

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

1

D.3 Publication Consent Form

Consent form for the publication in eprints.hsr.ch

○ Term Project

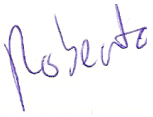
X Bachelor Thesis

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

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

$$f(s) = s'$$

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

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

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. 10, 12, 13, 15, 27, 31, 32, 52, 54

Figure 26: A Mesh, STL File or Anatomy Object

ARB

Architecture Review Board 16, *Glossary:* OpenGL Architecture Review Board

Calibration Data

Calibration data from the Zeiss Cameras ... TODO complete 12

Camera LEDs

The camera leds (Figure 27) are the graphical representation on screen of the real camera leds employed for the movement recording. 28

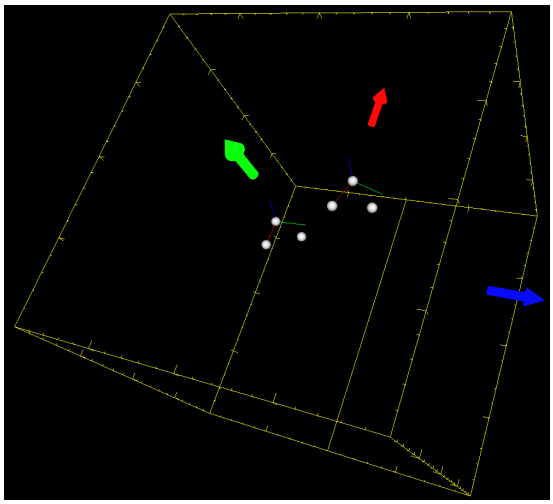


Figure 27: Camera Leds in Optis

Clip Space

Positions of Vertices after projection into a nonlinear homogeneous coordinate. [5] 16, 21

Fragment shader

A Shader that executes once per fragment and generally computes the final color of that fragment.[5] 17, 18, 25, 54

Frame

Almost equivalent to the definition of a film frame [25]. 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. 29

Frustum

A pyramid-shaped viewing volume that creates a perspective view. (Near objects are large; far objects are small). [5] 22

Geometry shader

A Shader that executes once per primitive, having access to all vertices making up that primitive.[5] 17, 18, 54

GLSL

OpenGL Shading Language 18, *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 [26]. 53

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. 15, 28, 34, 51, 54

Model Space

Positions relative to a local origin. Also sometimes known as *Object Space*. [5] 16, 21, 52

Motion Movement file

See 5.3.3.1 MVM files 10, 52

MVM

Motion Movement file (inaccurate) 10, 12, 15, 29–31, *Glossary:* Motion Movement file

NDC

Normalized Device Coordinates 20, 52, *Glossary:* Normalized Device Coordinates

Normalized Device Coordinates

The NDC [6] 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

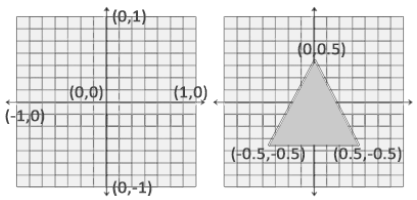


Figure 28: Normalized Device Coordinates [6]

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. 17, 20, 22, 52

Object Space

Positions relative to a local origin. Also sometimes known as *Model Space*. [5] 16, 21, 52

OpenGL

OpenGL [27] 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. 8, 11, 14–17, 20, 23, 25, 27, 28, 36

OpenGL Architecture Review Board

The committee body [5] 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. 16, 51

OpenGL Shading Language

A high-level C-like shading language. [5] 18, 36, 52

Optis

Proprietary software of the Clinic of Masticatory Disorders used to TODO COMPLETE THIS 29, 48, 51, 53

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 [5]" 25

Primitive

A group [5] 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. 16, 17, 54

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. 28

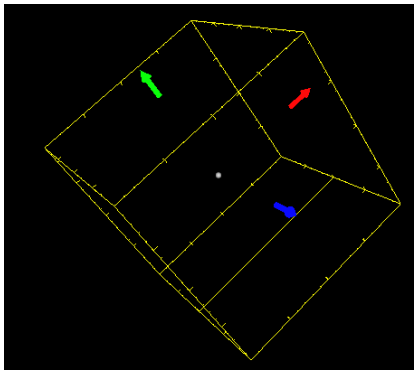


Figure 29: Reference Cube in Optis

Reference Sphere

Reference Sphere used for... TODO complete 12

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 [28] 28

Shader

A small program that is executed by the graphics hardware, often in parallel, to operate on individual Vertices or pixels [5]. There are Vertex shaders, Fragment shaders and Geometry shaders 16–18, 20, 32, 51, 52, 54

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 [18]. 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. 10, 27, 54

STL

Stereo Lithography Files 10, 12, 13, 15, 27, 51, *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. [5] 17, 54

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. [5] 17, 18

Vertex

A point in space is both a vertex and a vector [5]. Except when used for point and line primitives, it also defines the point at which two edges of a polygon meet. 15–17, 20, 21, 23, 27, 51–54

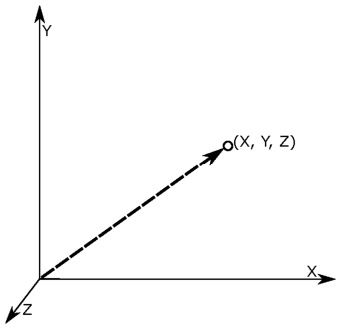


Figure 30: Vertex [5]

Vertex shader

A Shader that executes once per incoming vertex. [5] 17, 18, 54

View Space

Positions relative to the viewer. Also sometimes known as *camera* or *eye space*. [5] 16, 21, 24

Window Space

Or *Screen space*. Positions of Vertices in pixels, relative to the origin of the window. [5] 16

World Space

This is where coordinates are stored relative to a fixed, global origin. [5] 16, 21

References

[1] *Agile Software Development*. https://en.wikipedia.org/wiki/Agile_software_development Visited 03-06-2017.

[2] *Scrum*. https://en.wikipedia.org/wiki/Scrum_(software_development) Visited 03-05-2017.

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

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

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

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

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

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

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

[10] *3D Graphics with OpenGL By Examples*. https://www3.ntu.edu.sg/home/ehchua/programming/opengl/CG_Examples.html Visited 28-05-2017.

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

[12] *Virtual Camera System in video games*. https://en.wikipedia.org/wiki/Virtual_camera_system Visited 04-06-2017.

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

[14] *Gram-Schmidt Process*. https://en.wikipedia.org/wiki/Gram%E2%80%93Schmidt_process Visited 04-05-2017.

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

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

[17] *Normal Vector*. https://en.wikipedia.org/wiki/Normal_(geometry) Visited 05-06-2017.

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

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

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

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

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

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

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

[25] *Film Frame*. https://en.wikipedia.org/wiki/Film_frame Visited 12-03-2017.

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

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

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