FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Extension of Multiplatform Software for EEG Visualization
Student:	Bc. Marek Papinčák
Supervisor:	Ing. Petr Ježdík, Ph.D.
Study program:	Informatics
Branch / specialization:	Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2021/2022

Instructions

The aim of this thesis is to design and implement an extension of multi platform application which is capable to visualize EEG curves.

Functional requirements:

Visualization of EEG data using a topographical EEG scalp map.

Ability to perform a time-frequency analysis and visualize the results.

Other requirements:

Capability to visualize up to 256channels with sampling frequency 2048Hz.

Support of Microsoft Windows 7, 8, 10 and Linux Ubuntu 14 and 16.

Ability to run the application on a virtual machine.

The implemented expansion must be appropriately documented and tested.

Electronically approved by Ing. Michal Valenta, Ph.D. on 21 January 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Extension of Multiplatform Software for EEG Visualization

Bc. Marek Papinčák

Department of Software Engineering
Supervisor: Ing. Petr Ježdík, Ph.D.

June 27, 2021

Acknowledgements

I would like to thank my supervisor Mr. Ježdík for his guidance and my mom for her support in these strange pandemic times.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on June 27, 2021 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Papinčák, Marek. *Extension of Multiplatform Software for EEG Visualization.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

Táto diplomová práca popisuje rozšírenie programu na vizualizáciu a analýzu EEG dát. Rozšírenie zahrňuje vizualizáciu topografickej mapy povrchu skalpu použitím viac rozmernej interpolácie a časovo-frekvenčnú analýzu vykonanú pomocou krátkodobej Fourierovej transformácie vyzualizovanú pomocou spektrogramu. Obe vizualizácie používajú hardvérovú akceleráciu.

Klíčová slova EEG, hardvérová akcelerácia, OpenGL, OpenCL, biologické signály, spracovávanie digitálneho signálu, topografická mapa, spektrogram

Abstract

This thesis describes the extension of a program for the visualization and analysis of EEG data. The extension includes visualizing a topographic map of the scalp surface using spatial interpolation, and time-frequency analysis performed with short-time Fourier transform visualized as a spectrogram. Both, the spectrogram and topographic map, are visualized using HW acceleration.

Keywords EEG, hardware acceleration, OpenGL, OpenCL, biological signals, digital signal processing, topographic map, spectrogram

Contents

List of Figures

List of Tables

List of Listings

Introduction

ISARG[1] is a research group consisting of technically educated researchers from the Czech Technical University in Prague and medical doctors from Charles University in Prague. They analyze electrical signals produced by the nervous system. Their primary focus is studying the brains of epileptic patients.

Electroencephalography(EEG[2, p. 258]) is a measuring method used to record the electrical field on the scalp. A special cap with electrodes is placed on a patient's head that measures the electrical field and produces EEG signals. These continuous signals are then reduced into discrete signals and analyzed with various software tools. One of such tools used by the research groups is the in-house application Alenka.

Alenka is an EEG signal visualization software that was developed by students over the years. There are many complex commercial and open-source solutions on the market. Alenka strives to complement them, not replace them. The main point of the application is to be as simple and as fast as possible. The goal of this thesis is to expand the visualization capabilities of this student's project.

The main goal of this thesis is to design and implement an extension of the aforementioned EEG visualization application Alenka. The main two functional requirements of the extension, as specified in the thesis assignment, are:

- Topographic scalp map
- Time-frequency analysis

In the former requirement, the goal is to take irregularly distributed data points stored in a three-dimensional form in a shape of a cap. Then transform them into two-dimensional coordinates in such a way that the coordinates don't overlap and keep their distances. Then visualize a continuous stream of changing values at said points in a form of a 2D topographical map.

In the latter requirement, the goal is to process sets of data and visualize them in a form of a spectrogram.

There are also non-functional requirements:

- Ability to process files with 256 electrodes and 2048 Hz sampling frequency
- Compatibility with Windows 7, 8, 10, and Linux ubuntu 14 and 16 systems
- Documentation and testing

Alenka already provides support for the first two mentioned. This means that this goal translates to not introducing new elements into the application that could break this status. As this is my first contact with Alenka, the unmentioned goal is to familiarize myself with the project to provide the best solutions possible.

Chapter 1

Analysis

1.1 Alenka

Alenka takes sampled EEG data and visualizes them in a form of a graph, either in filtered or raw form. The application utilizes hardware acceleration to be able to quickly process and visualize massive amounts of data. Sampled EEG data are stored in files with various sampling frequencies and electrode formations with a different number of electrodes. These formations can go up to 256 electrodes with a 2048 Hz sampling frequency. Alenka accepts EDF, GDF, and MAT file types.

After loading, data are processed and filtered before being visualized. The researcher has the ability to use various basic filters including high pass, low pass, and notch filter. There is also a possibility to write a custom frequency multipliers. The processed data are visualized in the main window with the x-axis being time, the y-axis being the amplitudes, and each electrode having its own continuous wave. Only part of the file is visualized at a time, by moving the bottom scroll bar, the user is able to move through the file and observe all of the data. The amount of visualized data can be adjusted by changing the resolution, zooming in and out based on time and amplitudes. It is possible to interact with the signal waves, changing their amplitudes and creating special zones of interest. Picture of the main window and two side windows can be seen here 1.1.

Alenka provides various tools that amplify the analysis process. User is able to create custom montages by combining multiple electrodes. It is possible to perform a Fourier transform on a single electrode. There is an integrated video player that allows the user to watch a recording of a patient and observe his mental state. Alenka synchronizes the visualized data with the time step in the video. The application also provides a spike detector. This feature goes through the whole file and automatically detects zones of interest so that researcher doesn't have to. It is also possible to run multiple instances of the application on the same pc with a shared context, visualizing the same file.

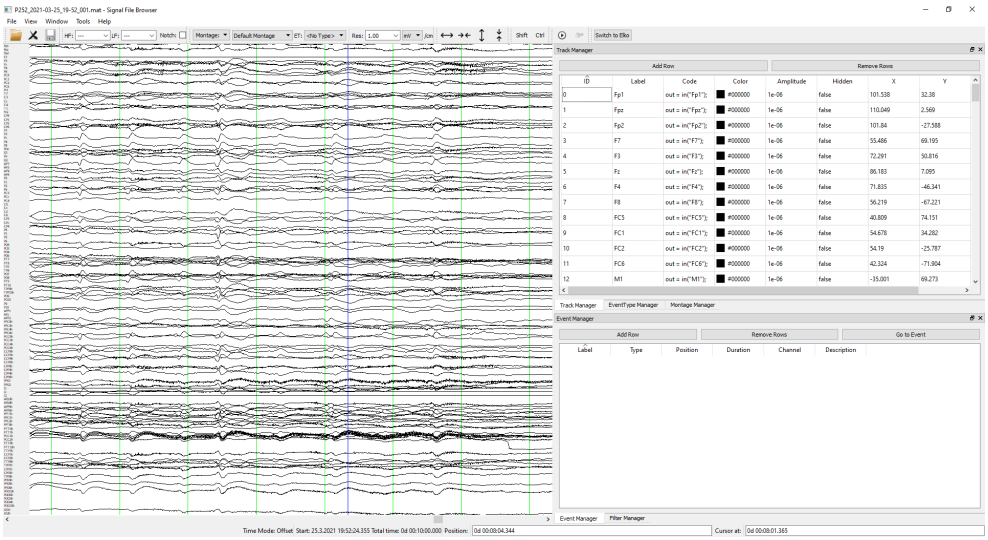

Figure 1.1: Alenka main window with two side windows.

You can read more about this and spike detector in the master thesis written by Martin Bárta[3]. Read more about introducing hardware acceleration into the project in his bachelor thesis[4].

Alenka[5] is a cross-platform application written in c++. It uses the Qt framework[20] as its main engine behind the graphical user interface(GUI) and to ensure compatibility across different systems, mainly Windows and Linux. Qt provides native-looking GUI based on the platform it is being run on. The application utilizes OpenCL and OpenGL for hardware acceleration. OpenCL is being used mainly during the processing and filtering of read sample data, while OpenGL is utilized in the visualization of the main window.

1.2 Topographic Scalp Map

The term topographic map is usually connected with the world of geology and its discipline topography. Topography generally studies terrain, the shape of a planet's surface, and its characteristics such as plains, mountains, rivers, and oceans. To best show the varying elevation of these characteristics, topography uses topographic maps, where areas with low elevation such as canyons are shown with a different color than high areas such as mountains. Since the color is based on the particular height of a coordinate on the map, a reader can quickly discern the steepness of hills, deepest and highest points, and the overall variability of the studied terrain. This idea also translates somewhat well into Electroencephalography(EEG), although the data measured in EEG do not provide complete information and are often misleading.

In Electroencephalography, several electrodes are put on top of the pa-

tient's head to monitor his brain activity. The brain contains billions of neurons that produce electric potential which can undergo an important change in milliseconds. The potential oscillates in time. This electric activity is usually recorded in the form of waveforms. A skilled technician or doctor can observe well-known patterns in these waveforms and tell what kind of activity was patient performing at the time. The observer can identify sleep stages, depth of anesthesia, seizures but also abnormalities[6, p. 3].

Another method of observing electric activity in the brain is a topographic map. There are 2D and 3D topographic maps. In this thesis, I focus on the former which is sometimes also called a topographic scalp map. The map shows one step of the electric potential oscillation in time. The method could be regarded as a "pseudo imaging method"[7, p. 7]. This can cause accuracy issues with the distribution of surface potential as we have a small number of spatial samples, but a relatively large area between them[8, p. 194]. The most exact topographic scalp map would be a set of points blinking in different colors. The interpolation adds substance and helps the observer. An example topographic scalp map can be seen in this picture 1.2

1.2.1 Spatial Interpolation

Spatial interpolation is a method where known points are used to estimate the values of unknown points in multi-dimensional space. There are often situations where resources and measuring points are limited. A typical example would be weather, there is a finite number of weather stations that do not cover the whole area. Various interpolation and approximation methods are then used to create complete rain or temperature maps[9]. Some of the methods are triangle interpolation[10] and inverse distance weighted.

Triangle interpolation generates spatially continuous surfaces using the triangulated irregular network(TIN[11]). Given irregularly distributed data points, a triangulation algorithm constructs nonoverlapping triangles such that every point from the grid is a vertex of at least one triangle. The interpolation of the given points is then simple, every point inside a triangle is a linear interpolation of the values observed at the triangle's vertices. A problem with this method is the assignment of the values to triangles. Also, the result inherits some artifacts from the triangular pattern, it is advised to use triangle interpolation at high observation densities to limit its influence[10].

Inverse distance weighted(IDW) is a simple spatial interpolation method where sample points are weighted based on the distance to the unknown point. This means that the further the sample point is from the unknown point, the less influence it has on its value. Not all points are used for a particular unknown point, either n closest points or points closer than a certain distance are selected. The result value is the weighted average of the considered points. The Disadvantage of the method is that it does not often produce the local

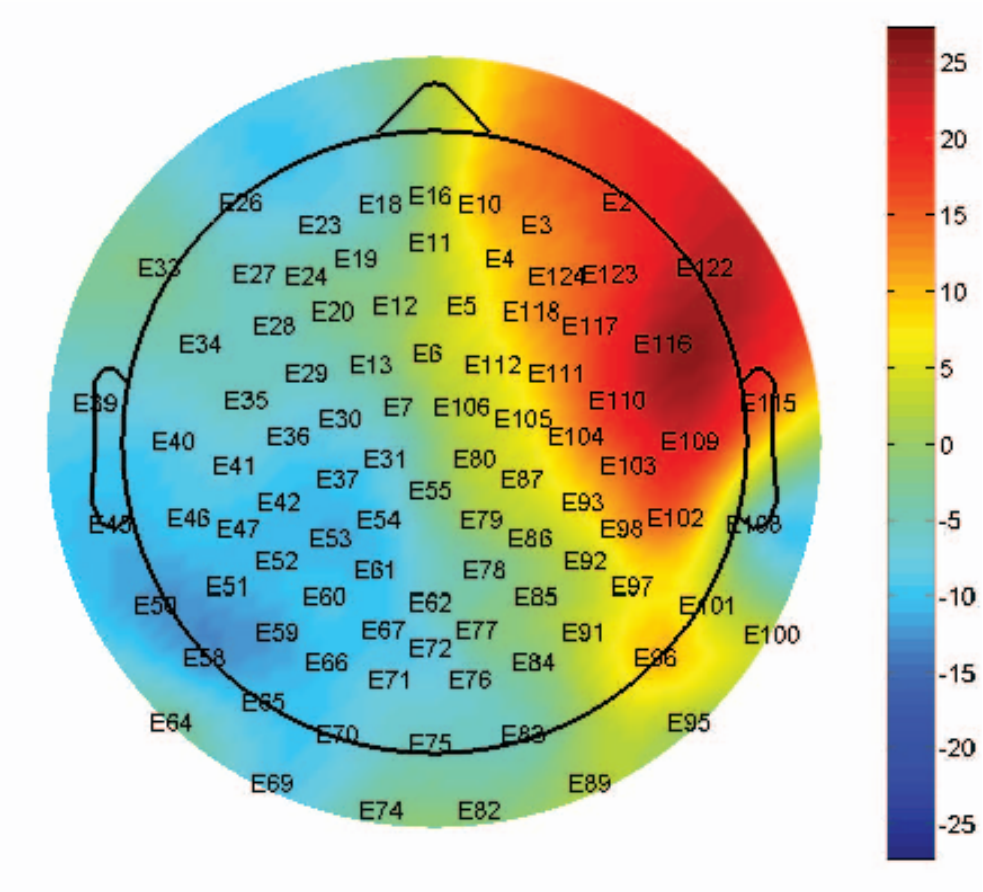

Figure 1.2: Example topographic scalp map plotted by EEGLAB

shape implied by data and produces local extrema at the sample points[9, p. 482].

1.2.2 OpenGL Interpolation

OpenGL[12] is a cross-platform API used to write hardware accelerated applications. OpenGL communicates with a graphics processing unit(GPU) to render complicated graphics objects in real-time. It is commonly used in computer games, computer-aided design, and scientific visualization. Modern GPUs have hundreds, or sometimes, thousands of cores, OpenGL takes advantage of this and runs parallel processes on all of the cores.

A graphics scene consists of multiple objects, each being made up of multiple primitives, each consisting of multiple vertices. One scene can contain thousands and even millions of vertices. To render such a scene, every object must first go through the graphics pipeline[13, p. 35]. Four major steps in the

pipeline are:

1. Vertex processing
2. Clipping and primitive assembly
3. Rasterization
4. Fragment processing

During the pipeline, each primitive is rasterized, turned into fragments, each containing interpolated values attached to the vertices of the primitive[14]. Every Fragment is then colored based on the fragment shader. For triangle primitives, OpenGL uses interpolation based on barycentric coordinates[13, p. 35]. If one of the parameters of the primitive vertex is color and we use a correct fragment shader, we get a simple and fast color interpolation inside the triangle.

1.2.3 Electrode Positions

Electrodes are used to measure the electric potential of the brain. They are placed on top of the patient's head according to internationally recognized systems. Systems used by ISARG[1] include 10-5, 10-20, and duke 256. These systems can have up to 256 electrodes. These electrodes sometimes referred to as channels, are placed on the patient's head based on well-known brain areas which are also reflected in their naming. Measurements done on different patients results in different electrode positions based on their skull shapes. Electrodes are supplied in a simple text file with each electrode having its three-dimensional cartesian coordinates.

1.2.4 Requirements Formalization

During the meetings with ISARG[1], several requirements on the scalp map were stated. The scalp map should be added as a new window in the windows section. The user should be able to pull it from the original position and move it around the screen, it should be easily resizable and turned off/on. It should visualize the same data as being shown in the main window of the application. This means that the data should have the same filters applied to it as selected in the main window toolbar. It should be possible to select between the local and customizable extrema. There should be electrode position indicators with their specific labels. Electrode positions should be customizable and there should be an option to load them from a file. There should be a customizable color gradient with customizable contrast and brightness.

1.3 Time-frequency Analysis

Time-frequency analysis(TFA) is one of the methods used in signal processing. It allows the researcher to study the signal in both the time and frequency domains simultaneously. Standard Fourier analysis assumes that signals are infinite in time or periodic. However, many types of signals in practice are rather short and significantly change over their lifespan. In TFA small time intervals, windows, of the signal are defined and separately transformed to the frequency domain. Results are then visualized in a form of a spectrogram, sometimes also called a sonogram. There are several different methods to perform TFA such as the short-time Fourier transform(STFT), the Wigner transform, the discrete wavelet(Haar) transform, and the continuous wavelet transform. In this thesis, I focus on the STFT.

Research has shown that the STFT can be used to analyze EEG signals successfully. Tzallas et.al.[15] compared STFT and various time-frequency distributions in the analysis of EEG recordings of epileptic patiens[15]. Hussin et.al. studied the EEG recording of normal and autistic children[16]. Zabidi et. al. used the STFT to compare the EEG recordings measured during relaxation to the recordings attained during writing.

1.3.1 The Short-Time Fourier Transform

Short-time Fourier transform is, simply put, the Fourier transform operating on small segments of a signal in the time domain. The results of the transforms are then stacked together and visualized in a form of a spectrogram.

In the discrete-time STFT, a sampled signal is divided into several, usually overlapping segments. Each segment is tapered with a window function(e.g. hamming) to avoid edge artifacts. The Discrete Fourier Transform(DFT) is then performed on the tapered segments. Often, the Fast Fourier Transform is used as a fast implementation of the DFT. The FFT is much faster for the powers of two. If this condition doesn't hold for the segment's size, it is padded with zeroes after it is tapered and before it enters the FFT.

Mathematically, the STFT would be described as follows[17]: Let $x[n]$ be a discrete signal and $x_l[m]$ be signal frames extracted at regular time intervals using a finite window function $w[m]$, expressed as

$$x_l[m] = x[m + lH]w[m], \tag{1.1}$$

where $m \in \{1, 2, ..., M\}$ is the local time index, $M \in N$ is the analysis window length, $l \in \{0, 1, ..., L - 1\}$ is the frame index, $L \in N$ is the total number of frames, $H \in N$ is the hop size (i. e., the time advance, expressed in samples, from one signal frame to next). Further, DFT is performed on every frame $x_l[m]$, given a localized two-sided spectrum

$$X[k, l] = \frac{1}{M} \sum_{m=1}^{K} X_l[m] e^{-j2\pi \frac{mk}{K}} \tag{1.2}$$

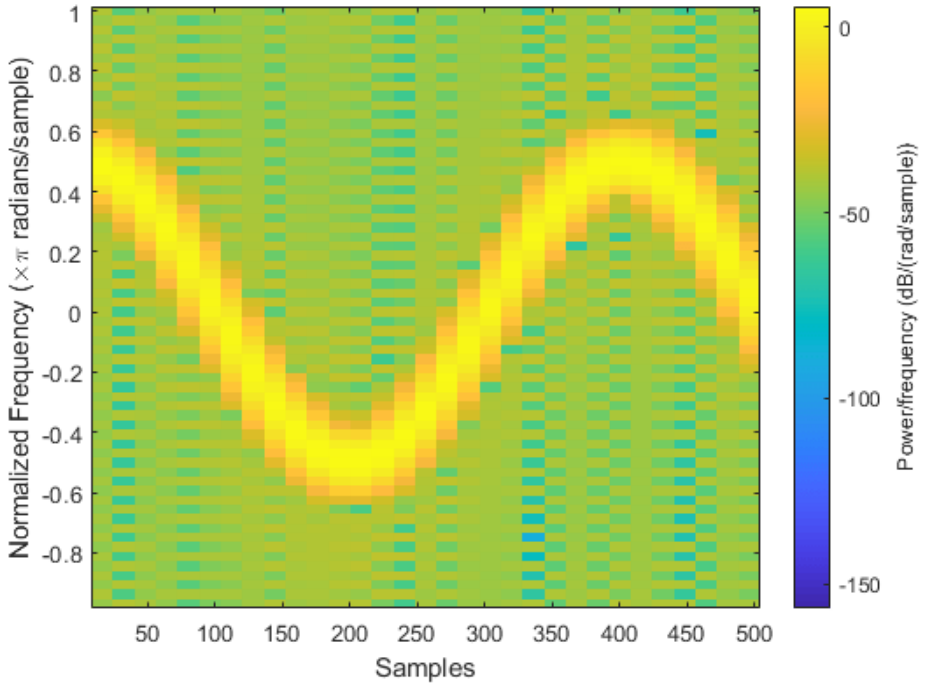

Figure 1.3: Example spectrogram plotted by MATLAB made by The MathWorks, Inc.

where $k \in 1, 2, ..., K$ is the frequency bin index and $K \in N$ is the DFT size. The term $X[k, l]$ is called STFT of $x[n]$ and corresponds to the local time-frequency behavior of the signal around the time index lH and the frequency bin k.

1.3.2 Spectrogram

The spectrogram can be defined as an intensity plot of the STFT magnitude. The spectrograms horizontal axis represents the time spectrum and the vertical axis shows the frequency spectrum. The third dimension is indicating the amplitude of a particular frequency at a particular time by the intensity or color of each pixel. What color represents which amplitude can be seen in the accompanying gradient. In STFT the frequency spectrum is divided into frequency bins and time is divided into frames. Each pixel of the spectrogram is a representation of the nth frequency bin and the mth time frame. An example spectrogram with clearly distinguishable pixels can be seen in this picture 1.3.

1.3.3 Requirements Formalization

During the meetings with ISARG[1], several requirements on the time-frequency analysis were stated. The time-frequency analysis should be added as a new side window. The TFA should be calculated from the samples currently observed in the main window. The middle of the TFA spectrogram's time axis should be the time step selected by the position in the main window. The frequency axis should be configurable, the researchers are not always interested in the whole frequency spectrum. There should be a $1/f$ and logarithm compensation.

Chapter 2

Design

2.1 Topographic Scalp Map

The scalp map is added as a new window in the windows section. It visualizes the signal amplitude at the current time step in the main window. This means that it has to be updated every time the position indicator in the main window moves. Thus, the update has to do the least amount of calculations required and be as fast as possible. The whole process required for the visualization of EEG data as a topographical scalp map could be summed into the following steps:

- Load electrode positions from electrode file
- Project positions from three-dimensional Cartesian coordinates to two dimensional
- Generate a mesh around electrode positions
- Obtain EEG sampled data to be visualized
- Visualize the EEG data

2.1.1 Loading Electrode Positions

Alenka manages electrodes through the track manager. The electrodes are loaded when a new file with samples is loaded. The track manager contains data about the electrodes that can be edited by the user. Edited data is saved after discarding the sample file and is restored upon opening it again. The track manager already contains prepared empty fields for three-dimensional Cartesian coordinates. That's why I added an option to load electrode positions here. The positions are loaded from a simple text ELC file. Each position has its label which is compared to already existing labels in the track manager table. The matching electrodes are updated with coordinates from

Listing 2.1: An example elc file with electrode positions and labels.

```
NumberPositions= 5
UnitPosition mm
Positions
FP1     :  101.538 32.38   40.252
FPZ     :  110.049 2.569   40.179
FP2     :  101.84  -27.588 41.343
AFP3H : 105.688 20.856  57.759
AFP4H : 105.472 -18.184 58.19
Labels
FP1   FPZ   FP2   AFP3H AFP4H
NumberHeadShapePoints= 5
UnitHeadShapePoints mm
HeadShapePoints
96.694  -2.084  1.205
98.834  -15.204 11.905
78.65 -53.17  13.188
63.894  -56.713 -16.333
87.749  -13.982 -12.454
```

the file. Example ELC file can be seen here 2.1. Alenka loads the information from the five lines after the *Positions* line, as the *NumberPositions* is 5.

2.1.2 Projection of Electrode Positions

Electrode positions are loaded in a form of three-dimensional Cartesian coordinates. To use them in this form would mean that electrodes that are on the side of the head would overlap and the information from these parts of the skull would be very hard to decipher. It is thus vital to project these coordinates onto a two-dimensional plane. For this, I use stereographic projection as presented in here[18]. The sphere equation in Cartesian coordinates is:

$$x^2 + y^2 + z^2 + ax + by + cz + d = 0. \tag{2.1}$$

The first step is to use the least-squares method to fit a sphere inside the electrode positions. The sphere has a center (x_c, y_c, z_c) and R expressed as:

$$(x_c, y_c, z_c) = (-a/2, -b/2, -c/2), R = \sqrt{(a^2 + b^2 + c^2)/4 - d}. \tag{2.2}$$

The electrode positions are then projected onto the sphere surface $P = (x_p, y_p, z_p)$ and mapped to a plane by stereographic projection:

$$(x', y') = (\frac{x_p}{R - z_p}, \frac{y_p}{R - z_p}). \tag{2.3}$$

Projecting of the electrodes happens only when new electrodes are loaded, or some electrode's coordinates are updated in the track manager. It doesn't interfere with the scalp map main update.

2.1.3 Mesh Generation

Graphical objects are constructed from primitives, such as points, lines, and triangles. To visualize the topography of the head, it first has to be broken down into these primitives, in this case into triangles. There were two options that I was considering:

- Two-dimensional square grid, where each square is represented by two triangles
- Two-dimensional triangle mesh created by triangulation of the area between the electrode positions

The former option, as described in the bachelor's thesis written by Petr Dobeš[19], involves creating a simple square grid and mapping the electrodes to it. This has the advantage that the grid doesn't have to be regenerated every time the electrode positions are changed. The disadvantage is that it has a square shape. It would either require to be cropped after the mapping for better readability or be overpainted by a sphere frame in every paint update.

The second option comprises of using Delaunay triangulation to create a triangle mesh that fits the electrode positions in such a way, that no position would be inside a triangle[18]. Triangles would then have to be split into smaller triangles based on distances between positions to avoid visual artifacts generated by triangle interpolation[10]. This process would have to be repeated every time that any electrode is changed. Although, slower algorithms performing Delaunay can have overall runtime of $O(n^2)$, electrode positions change doesn't happen that often and usually happens only once when adding a new electrode file.

Overall, both options are viable, but I decided to use the Delaunay triangulation as it didn't require the extra step of cropping and created a neat spherical mesh.

2.1.4 EEG Sample Data

EEG sample data are loaded from the sample file, processed, and visualized in the main window. After being processed they are stored in caches, so they don't have to be reloaded and processed every time the time step is changed in the main window. The data processing happens during the paint call in the main window. Visualization of the main window has to happen and can't be turned off in the current implementation of Alenka. Data visualized by the scalp should be already processed and filtered using the filters selected for the main window. I considered two options:

- Load raw data from files, process them, and visualize them
- Store data at the current time step during the visualization in the main window

Both options are not optimal, the first option does the job that was already done in the main window. The second option creates a new dependency on the main window. As there is already a dependency on the position indicator in the main window, I decided to use the latter option. The main window is an integral part of the application and can't be turned off so it will always be present besides the scalp map. The cleanest solution would be to refactor the main window in such a way that the visualization wouldn't be tied to the processing of the data. There would be a single processing unit from which all visualizers would receive the data.

2.1.5 Visualization of EEG Data

In the visualization step, the prepared mesh is colored based on the currently loaded EEG sample data. Scalp map mesh is created using Delaunay triangulation and then split into smaller triangles. The color of every pixel in each triangle is interpolated from the vertices of the triangle. By changing the time step in the main window, the amplitude at each electrode position is updated. This means that only the vertices located at the electrode positions have the current amplitude. Other vertices, from the smaller triangles, have to be calculated. This is done with the Inverse distance weighted(IDW) algorithm.

IDW is a simple spatial interpolation method. Given the N amplitudes $z_j, j = 1, ..., N$ measured at electrode points $r_j = (x, y), j = 1, ..., N$. Let m be the amount of nearest points, $\mathbf{r}$ be the unknown point, and p being a parameter that amplifies the influence of the closer points, then:

$$F(\mathbf{r}) = \sum_{i=1}^{m} w_i z(\mathbf{r}_i) = \frac{\sum_{i=1}^{m} z(\mathbf{r}_i)/|\mathbf{r} - \mathbf{r}_i|^p}{\sum_{j=1}^{m} 1/|\mathbf{r} - \mathbf{r}_j|^p} \tag{2.4}$$

is the estimator of the amplitude value at the unknown point $\mathbf{r}$[9]. Updated data are then visualized in the paint step. After that small circles can be visualized at the electrode points to indicate their position, and after that their labels.

2.1.6 The User Interface

The scalp map GUI is rather minimalistic. Most of the window controls can be found in the pop-up menu that gets triggered by the right-mouse click. The menu can be seen in its wireframe 2.1. Most of the options are resolved by a single click. Clicking on custom extrema triggers a new window where the user can set up desired values. The only feature that is controlled outside the pop-up is the brightness and contrast values. These can be altered by clicking on the gradient and dragging the mouse. I chose this approach as the window will be mostly viewed simultaneously with the main window. This means that

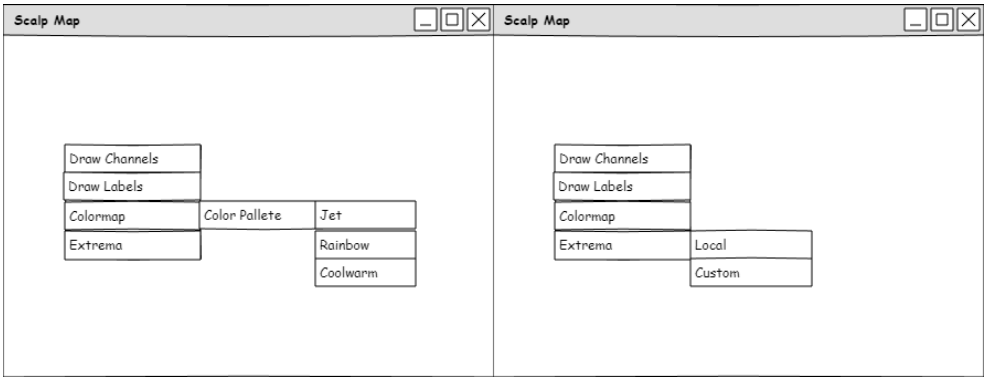

Figure 2.1: Proposed design of the Alenka scalp map pop-up menu

if the user doesn't have a second monitor, the scalp window will be small, and adding permanent toolbars would only make it smaller.

The electrode loading is added as an option into the pop-up menu in the track manager since it is changing the data in the track table. Also, the electrode positions might be used elsewhere in the future. Clicking on the load electrode position option brings up the standard file system browser tied to the used platform, where the user can select the ELC file from its location.

2.1.7 Visualization Design

The scalp map window mock-up can be seen here 2.2. Most of the window is filled with the topographic map.I chose a black background as the white and light colors vastly outnumber black color in the color interpolation. Also a large part of the gradient can be white if the contrast is increased. There is a small gradient with electrode voltage values on the right side, so the observer can understand which colors belong to what values.

2.1.8 Class Structure

Picture 2.3 shows the class structure that needed to be implemented or extended for the scalp map visualization.

Electrodes are loaded in the `TrackManager` using the `ElcFileReader` and stored inside the track table. `ScalpMap` is the base window class. It communicates with the global context through the `OpenDataFile` class.The `ScalpMap` class servers as a controller for the calculations that happen in the `ScalpModel` class and for the visualization that happens in the `ScalpCanvas`. `ScalpMap` receives impulses from the `SignalBrowserWindow` to generate electrode positions or to update the amplitudes at its positions. Both are then calculated in the `ScalpModel`. After that, they are sent to the `ScalpCanvas` class to be processed and visualized.

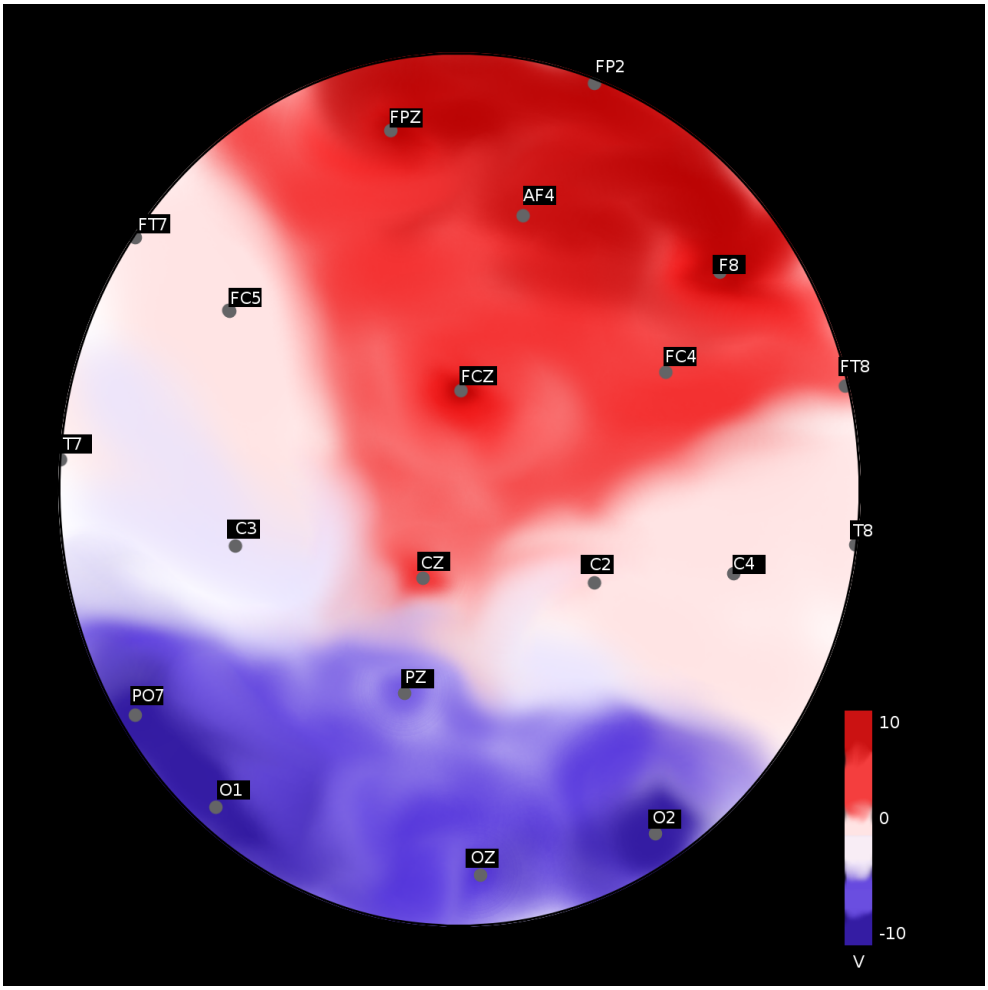

Figure 2.2: Proposed desing of the Alenka topographic scalp map

`ScalpCanvas` appropriates the data for visualization. It generates the scalp mesh when positions are updated and interpolates the amplitudes. It contains multiple graphics helper classes: `Colormap` that manages the color pallete used for visualization, `Gradient` for color pallete manipulation and `RectangleText` for rendering the gradient text and labels of electrode positions.

2.2 Time-frequency Analysis

The TFA is added as a new side window on the right side of the main window that visualizes a single electrode's EEG data in a form of a spectrogram. The data is processed with the STFT and turned into a square mesh where each vertex is assigned a specific color based on the STFT results. The whole

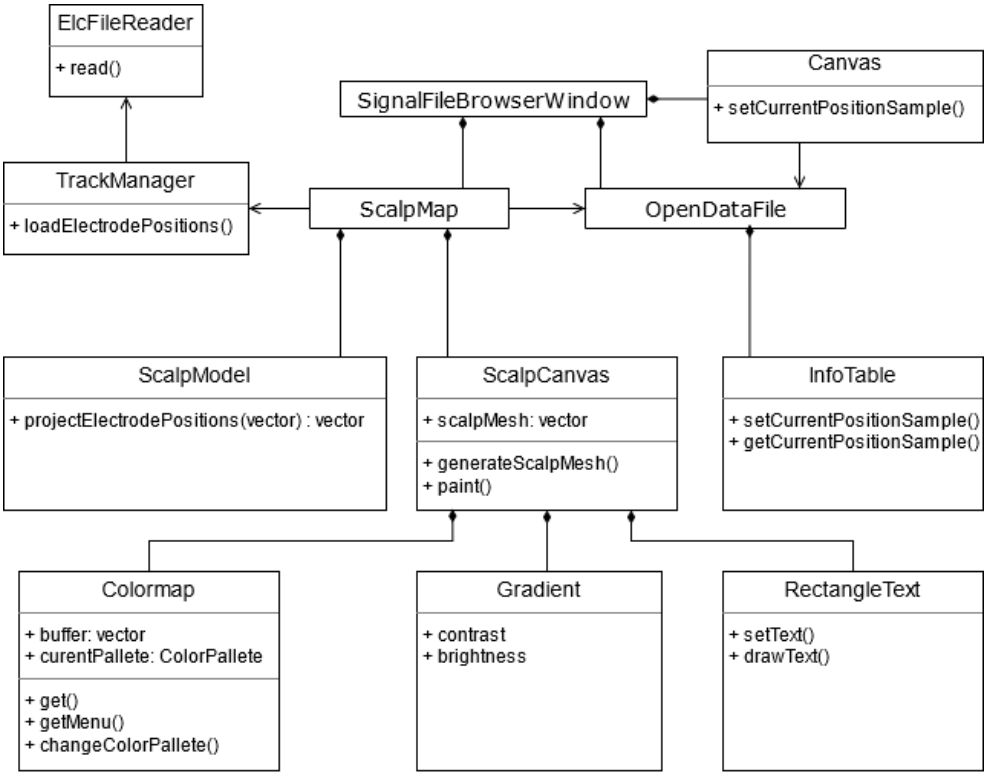

Figure 2.3: UML of Alenka scalp map class structure

process of computing and visualizing the STFT could be summarized into:

- Data acquirement
- Data processing
- Mesh generation
- Visualization

2.2.1 Data Acquirement

The TFA visualizes signal samples of a single electrode. The amount of samples is specified by the user in seconds and is independent of the amount shown in the main window. The data are loaded from the file as the STFT works with raw sample data. The loaded data are padded with zeroes if the required amount of samples is out of the range of samples stored in the file.

2.2.2 Data Processing

Loaded samples are processed with the STFT as described in the analysis section. Samples are split into the required amount of frames. This amount is calculated from values specified by the user, the frame size, hop size, and the time to be shown. First, a windowing function is applied to the frame. Then if the frame size is not a power of two, the frame is padded with zeroes. The FFT is applied to the frame and the result magnitudes are filtered. Only the frequency bins desired to be shown are selected. Finally, the $1/f$ compensation and *log* compensation is applied if required.

2.2.3 Mesh Generation

The STFT is visualized in form of a spectrogram. To visualize the spectrogram, a square grid mesh is generated. The mesh is regenerated every time the STFT result array is resized. This happens when the user changes the STFT parameters or sets the minimum and maximum frequencies to be shown.

2.2.4 Visualization

There are multiple elements that need to be visualized:

- Static graph elements - window frames, axis lines, and axis names
- Dynamic graph elements - axis numbers and gradient numbers
- Spectrogram mesh
- Gradient

Not all elements need to be redrawn all the time. The static graph elements should be repainted only when the window is resized. The Frequency axis numbers and the time axis numbers need to be repainted when they are updated by the user or a new file loaded. The gradient should be repainted when the color spectrum is changed or morphed. The gradient numbers and the spectrogram should be repainted every time the STFT data is updated.

The color of the spectrogram is determined by the STFT data. Each square in the grid mesh has a single color based on the magnitude in the STFT data array. The user can see what color corresponds to what magnitude in the gradient.

2.2.5 User Interface

Most of the TFA's GUI is concentrated in the top toolbar, where the user can regulate the STFT and resolution of the resulting spectrogram. Then there is the gradient, the user can change the brightness and the contrast of the colors by clicking on the gradient and dragging the mouse. There is also a

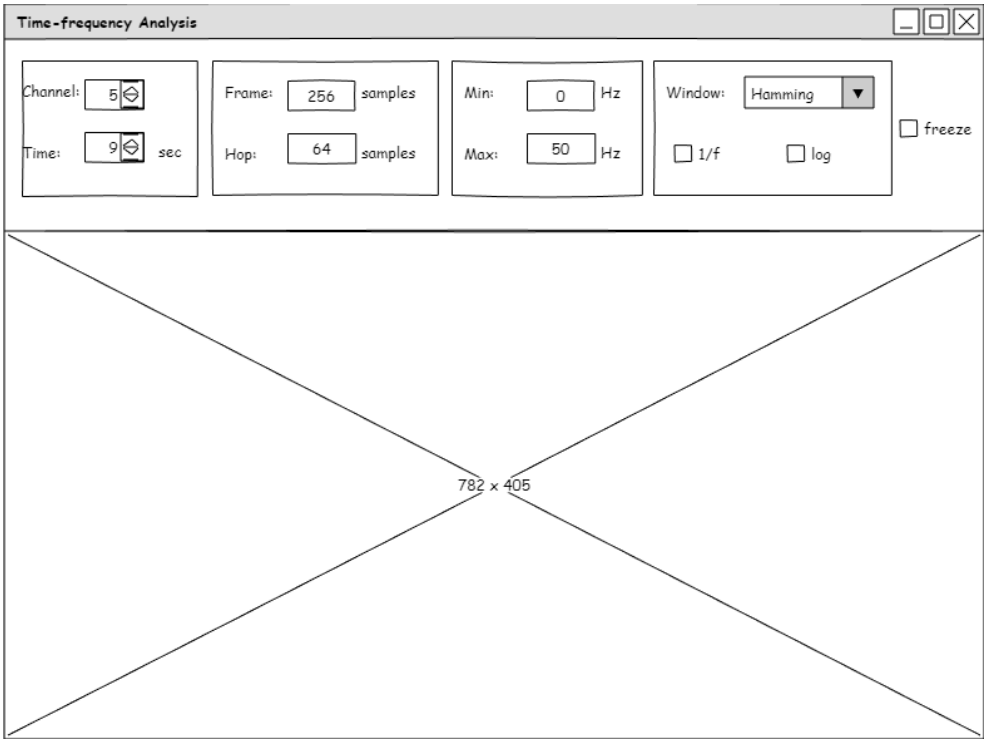

Figure 2.4: Wireframe of the TFA GUI

minimalistic popup menu where the user can change the colormap used in the spectrogram. Wireframe of the GUI can be seen in this picture 2.4.

The main idea behind the GUI is that the user should always be able to see the options that he has to regulate the STFT process. These options should be also easily and quickly changeable. Users shouldn't have to do multiple clicks to change the fundamental attributes.

2.2.6 Class Structure

The TFA's class structure is very similar to that of the topographic scalp map visualization. The `TfAnalyser` is a controller class that hosts the `TfModel` and the `TfVisualizer`. The `TfAnalyser` is a part of `SignalFileBrowserWindow` and communicates with the global context through `OpenDataFile`. The required samples are loaded and processed in the `TfModel`. It does that with the help of the `FftProcesser` that contains the calculations of the FFT. The `TfVisualizer` draws the processed data. It contains the graphics classes - `Colormap` and `GObject` with its derivates to divide the visualization process. UML diagram of the class structure can be seen in this picture 2.5.

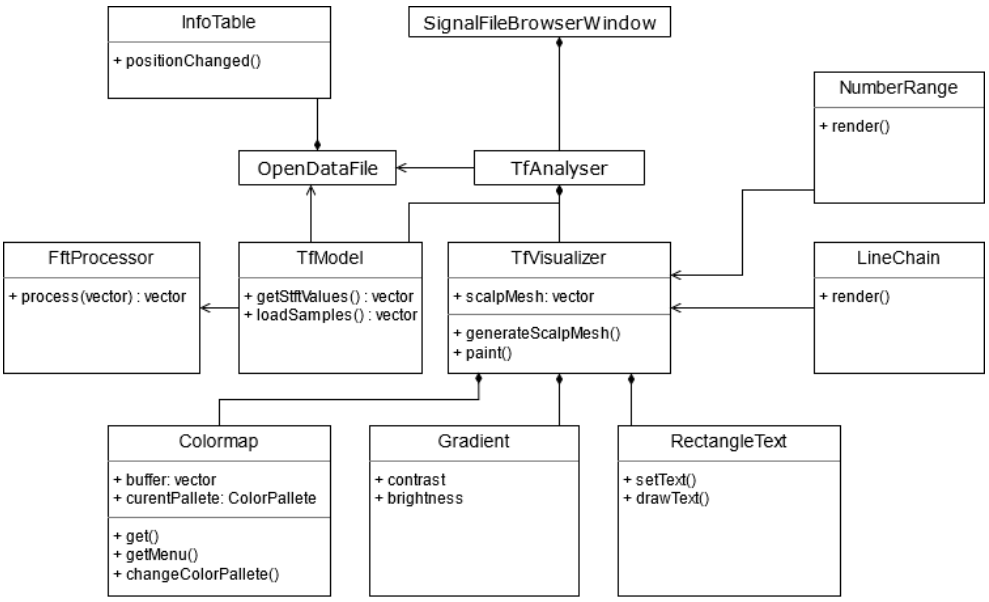

Figure 2.5: UML of Alenka TFA class structure

2.3 Graphics Objects

Both, the scalp map and the TFA use various common objects. Apart from the main colored meshes, the visualization could be deconstructed into multiple objects. I use a series of helper classes for the visualization of these elements as can be seen in this UML diagram 2.6. The `GObject` contains the coordinates for the area where the object is supposed to appear. The `Rectangle` renders a simple rectangle, used for frames. `Gradient` inherits the rectangle render but also takes part in the control of the color palette. `RectangleText` renders text inside of a rectangle area with the desired alignment and orientation. `RectangleChain` renders multiple rectangle objects in a row or a column. `NumberRange` renders a series of numbers from the specified minimum to maximum.

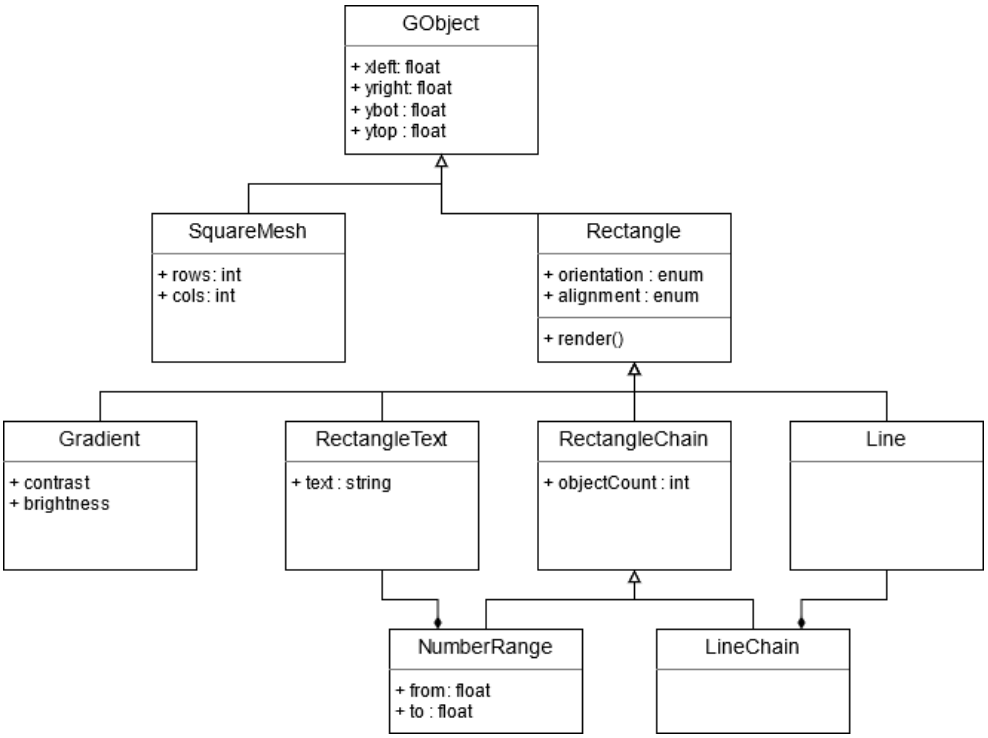

Figure 2.6: UML of Alenka graphics objects class structure

Chapter 3

Realization

3.1 Technologies

Alenka is a project developed by several people over the years. It is built on the Qt framework[20], which provides solutions for GUI development and OpenGL[12] integration. Martin Bárta reworked the project to utilize hardware acceleration through OpenGL and OpenCL[4]. I continue in this tradition and work with technologies already introduced into the project. I use OpenGL for the visualization of the main components in the added windows. I also use Qt's low-level painting solution QPainter[21] for minor parts such as text and lines used in graphs. I use clFFT library[22] that contains OpenCL functions for fast computing of a fast Fourier transform(FFT).

3.2 Topographic Scalp Map

I implemented the scalp map as explained in the design chapter. The `ScalpMap` is part of the `SignalFileBrowserWindow`. The QDockWidget is used to lock it in place or "dock" it in the side panel, similarly to other side windows. `ScalpMap` is implemented as an extension of the QWidget. This means that it can receive Qt signals emitted from the other widgets in the application. These signals trigger class-specific actions such as updating the electrode positions or amplitudes. `ScalpMap` can access the global context of the application through the `OpenDataFile`.

The visualization updates every time the position in the main window is changed. Whenever `ScalpMap` receives the time position update Qt signal, the amplitudes in `ScalpCanvas` are updated which is an operation with time complexity of $O(n)$. After that, the data are visualized with an OpenGL paint call as explained further in the text. If `ScalpMap` receives the electrode positions update, the mesh needs to be regenerated which is an operation with a time complexity of $O(n^2 log(n))$.

The final result can be seen here 3.1.

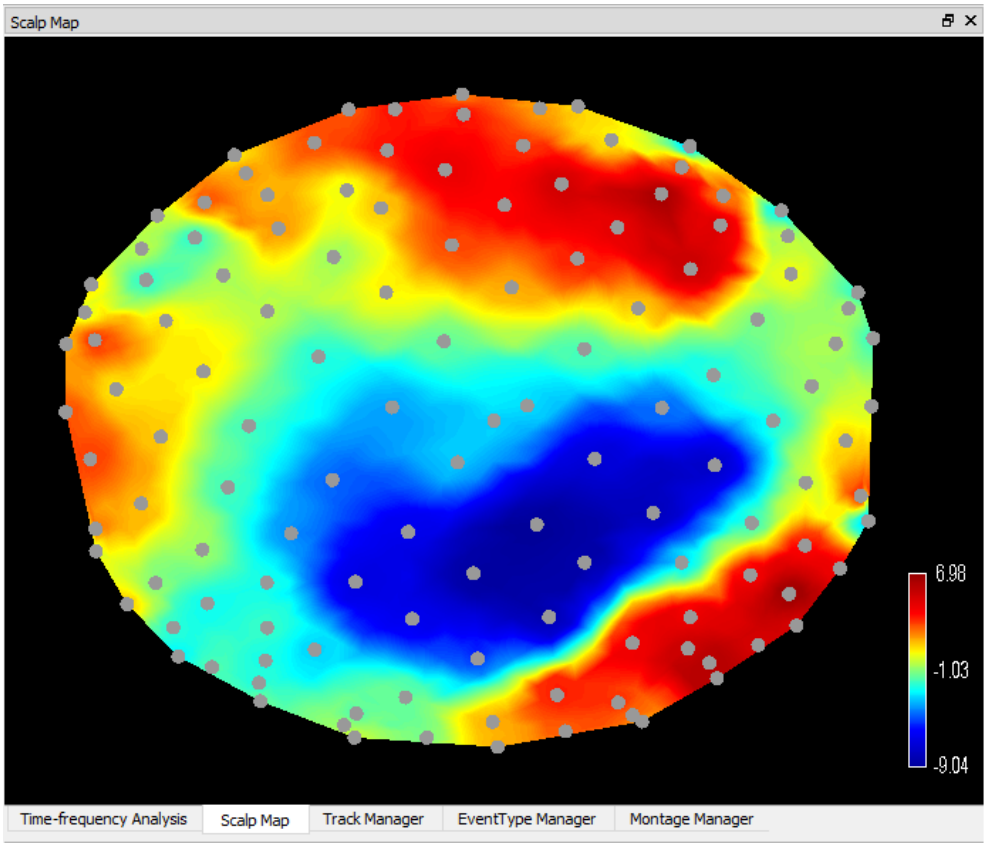

Figure 3.1: The final version of the topographic scalp map with electrode positions and jet color pallete

3.2.1 Electrode Positions

Loading of the electrode positions was implemented in the `TrackManager` and AlenkaFile. There are two ways to add new electrode positions:

- Editing the track table
- Loading from ELC file

The former was already partly present as there were editable coordinate columns in the track table. The latter was implemented as a `loadCoords()` method that is part of `TrackManager` that triggers a `QDialog` with file system explorer where the user can select the ELC file. The file is then read with the `ElcFileReader::read()` method. Read data are then inserted into the track table based on matching labes. There was a problem that each insert emits a Qt signal that triggers the whole electrode projecting and visualization process. I solved this by the `QObject::blockSignal()` method that suppresses the signal emitting and blocked all but the last of the inserts.

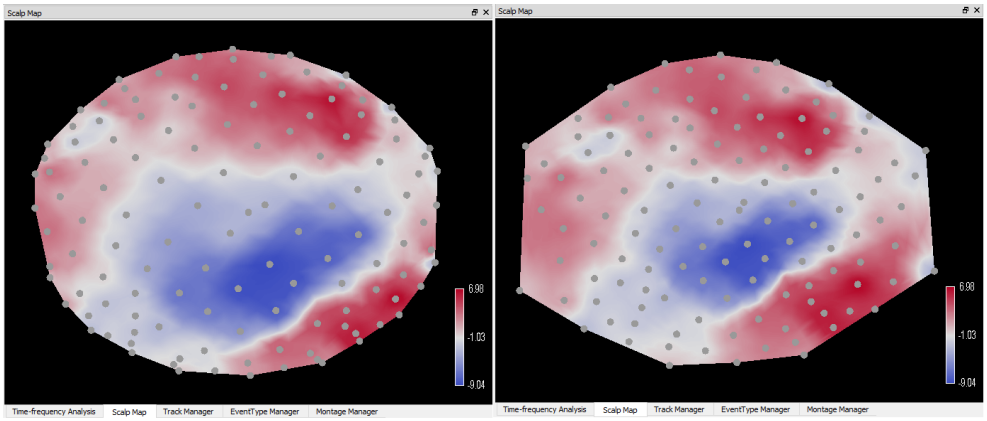

Figure 3.2: Comparison of the implemented projections. Left is the complete stereographic projection and right is fitting the points into a sphere.

3.2.2 Electrode Projection

The Electrode projection is implemented in the `ScalpModel`. The sphere fitting and the stereographic projection are used as presented in the design section. I reimplemented a sphere fitting least-squares method from pseudocode written by David Eberly[23]. During the implementation and testing, a version without the final step of stereographic projection was proposed. ISARG liked this version so I included both projections in the final window. The projection selection has been added to the pop-up window. This image compares the two types of projections 3.2. The implementation can be seen here 3.1.

The time complexity of the whole process is $O(n)$, where n is the number of electrodes. First, a sphere is fitted inside of the 3D coordinates with a complexity of

$$(O(3n) = O(n). \tag{3.1}$$

After that, the electrode positions are projected onto the sphere with a complexity of $O(n)$. Then, the positions are transformed from 3D to 2D space which is $O(n)$ as well. Finally, the 2D coordinates are normalized with a complexity:

$$O(2n) = O(n) \tag{3.2}$$

Together, the complexity of electrode projection is

$$O(4n) = O(n). \tag{3.3}$$

3.2.3 Mesh Generation

Projected coordinates are sent to `ScalpCanvas`, where a triangle mesh is generated using the Delaunay triangulation computed by the `Delaunator-cpp`[24].

Listing 3.1: Implementation of the electrode projection inside of `ScalpModel`

```
if (!fitSpehere(positions, sphereCenter, radius)) {
  return resultPositions;
}

QVector3D fittedSphere = sphereCenter / -2.0f;

float r = std::sqrt((sphereCenter.x() * sphereCenter.x() +
    sphereCenter.y() * sphereCenter.y() +
  sphereCenter.z() * sphereCenter.z()) / 4.0f - radius * 2);

std::vector<QVector3D> positionsProjectedOnSphere;
for (size_t i = 0; i < positions.size(); i++) {
  positionsProjectedOnSphere.push_back(projectPoint(positions[i],
      fittedSphere, r));
}

for (size_t i = 0; i < positions.size(); i++) {
  QVector2D newVec = { 0, 0 };

  if (useStereographicProjection) {
    newVec.setX(positionsProjectedOnSphere[i].x() / (r -
        positionsProjectedOnSphere[i].z()));
    newVec.setY(positionsProjectedOnSphere[i].y() / (r -
        positionsProjectedOnSphere[i].z()));
  }
  else {
    newVec.setX(positionsProjectedOnSphere[i].x());
    newVec.setY(positionsProjectedOnSphere[i].y());
  }

  resultPositions.push_back(newVec);
}

scaleProjected(resultPositions);
```

The library is a `c++` reimplementation of a robust `JavaScript` library[25]. The Delaunator is inspired by a sweep-hull algorithm which has a $O(nlog(n))$ time complexity[26]. The total time complexity of triangle grid generation is

$$O(2n + nlog(n)) = O(nlog(n)). \tag{3.4}$$

Let n be the number of vertices and b be the number of vertices on the convex hull. The maximum of triangles generated by the Delaunay triangulation is

$$t = 2n - 2 - b \tag{3.5}$$

The next step is splitting the triangles in the mesh. Each triangle is divided into 4 smaller triangles with the same area. This has a time complexity of $O(t)$. I divide the mesh twice as dividing more causes a massive strain on visualization computation time and the results are not much better.

Standard 10-20 system of electrode placement with 128 electrodes has 14 electrodes in the convex hull, this means that the resulting maximum amount of triangles would be

$$(2 * 128 - 2 - 14) * 4 * 4 = 3840. \tag{3.6}$$

The final step is a computation of the spatial coefficients used in spatial interpolation. Although this step is tied to the color interpolation and update of the amplitudes, I do it here as it only has to be done once when the coordinate positions change. The code can be seen here 3.2. The variable parameters are the nearest neighbor count and the p parameter, the amount of influence that distant positions should have on the color. 3 nearest neighbors and p of 1 produces the best results. Both adding more neighbours and using bigger p parameter creates artifacts in the resulting visualization. The time complexity of the computation is

$$O(t(n + nlog(n)) + 3) = O(4 * 4 * (2n - 2 - b) * (n + nlog(n))) \tag{3.7}$$

which can be simplified into

$$O(n^2 log(n)). \tag{3.8}$$

Together, the time complexity of the mesh generation is

$$O(nlog(n) + O(n) + O(n^2 log(n))) = O(n^2 log(n)). \tag{3.9}$$

3.2.4 Amplitude Update

Whenever the position indicator in the main window is changed, the `Canvas` stores the amplitudes in the current time step into the `InfoTable` which sends

Listing 3.2: Implementation of the spatial coefficients computation

```
for (int i = 0; i < triangulatedPoints.size(); i +=
    OPENGL_VERTEX_SIZE) {
  std::vector<std::pair<float, int>> distances;
  std::vector<PointSpatialCoefficient>
      singlePointSpatialCoefficients;

  for (int j = 0; j < originalPositions.size(); j++) {
    float distance = getDistance(triangulatedPoints[i],
        triangulatedPoints[i + 1], originalPositions[j].x,
        originalPositions[j].y);
    distance = (distance == 0) ? 0.0001f : distance;
    distances.push_back(std::make_pair(1.0f / (distance *
        distance), j));
  }

  std::sort(distances.begin(), distances.end());

  for (int j = distances.size() - 1; j > distances.size() -
      SPATIAL_NEAREST_NEIGHBOUR_COUNT - 1; j--) {
    float pDist = distances[j].first;
    for (int s = 1; s < SPATIAL_P; s++) {
      pDist = pDist * distances[j].first;
    }

    singlePointSpatialCoefficients.push_back(PointSpatialCoefficient(pDist,
        distances[j].second));
  }

  pointSpatialCoefficients.push_back(singlePointSpatialCoefficients);
}
```

a Qt signal received by the `ScalpMap`. This triggers an update of the amplitudes in the `ScalpCanvas`. This is a simple $O(n)$ operation, as everything needed is already precalculated. First, the amplitudes on the original electrode positions get updated. Then, the amplitudes at every mesh vertex get recalculated. `ScalpCanvas` has information about every vertex, amplitudes at its e nearest electrode positions A, and e spatial coefficients S that they should be multiplied with. The performed operation on every vertex is

$$a = \frac{\sum_{i=1}^{e} A[i]S[i]}{\sum_{i=1}^{e} S[i]} \tag{3.10}$$

where a is the new vertex amplitude. The next step is the OpenGL paint update which is explained further in the text.

3.3 Time-Frequency Analysis

The TFA is implemented according to the class structure described in the design chapter. `TfAnalyser` is implemented as an extension of the QWidget[27]. It is connected to the global Qt context and receives Qt signals that trigger the TFA process. `TfVisualizer` is an extension of the QOpenGLWidget.

Data processed in `TfModel` and sent to the `TfVisualizer` to be prepared before paint call. The processing in the `TfModel` happens every time the position is changed in the main window and has a complexity of

$$(O(f * n * log(n)) \tag{3.11}$$

for f frames with n samples per frame. The result is an array with

$$f * (n/2 + 1) \tag{3.12}$$

values that is then sent to the `TfVisualizer`. `TfVisualizer` either updates its already existing mesh which is an $O(f * n)$ operation or generates a new mesh if the resolution is different. The mesh generation has the same time complexity of $O(f * n)$. The mesh is then visualized in paint call explained further in the text.

3.3.1 Sample Processing

The sample processing happens in the `TfModel`. The implementation of the STFT can be seen here 3.3. First, data are prepared for the FFT. A window function with a time complexity of $O(n)$ is applied to the frame samples. Then, if the frame size is not a power of two, the samples are padded with zeroes. This has a time complexity:

$$O(n - 1) = O(n). \tag{3.13}$$

Together, the sample preparation has a complexity:

$$O(f * (n + n)) = O(f * n), \tag{3.14}$$

where f is the frame count and n is the frame size. The frame count is expressed as:

$$frameCount = \frac{totalSamples - frameSize}{hopSize + 1}, \tag{3.15}$$

$$totalSamples = secondsToDisplay * samplingFrequency. \tag{3.16}$$

The next step is the FFT, which is calculated using the `FftProcessor`. The FFT has a complexity of $O(n*log(n))$ since the zero-padded frame size is

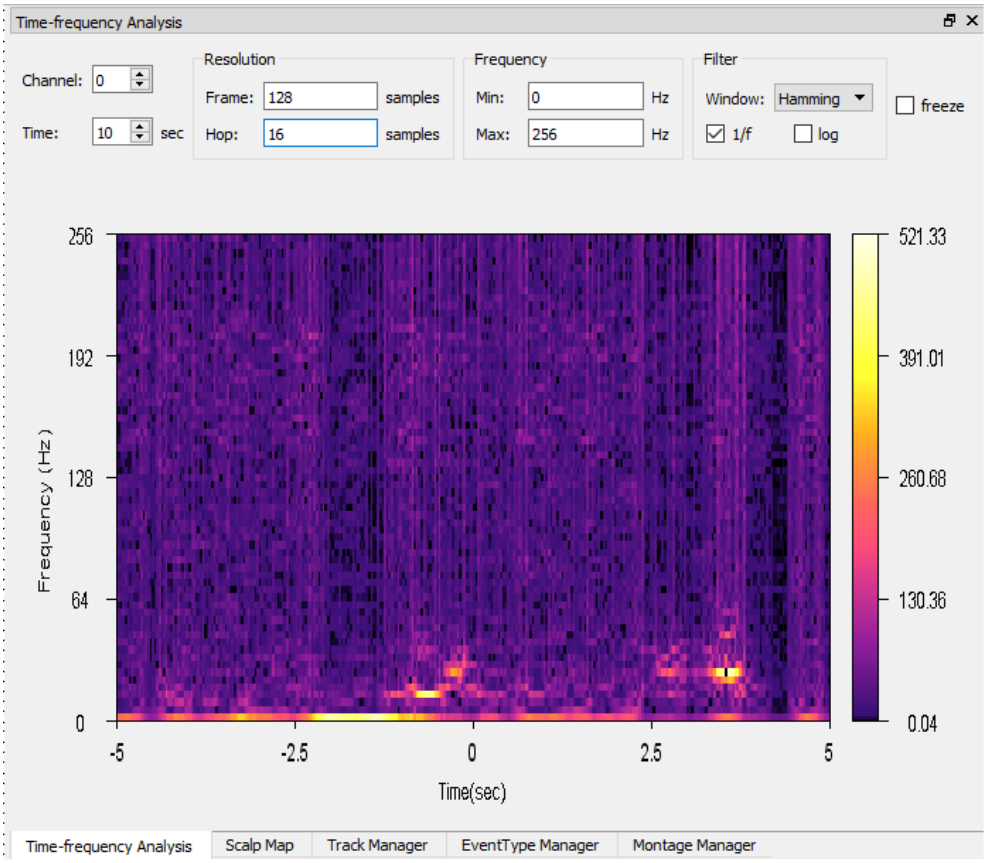

Figure 3.3: The final version of the TFA with inferno color pallete.

the power of two. The FFT is computed on f frames so the result complexity is

$$O(f * n * log(n)). \tag{3.17}$$

`FftProcessor` takes all of the frames in a bulk and returns

$$frameCount * (n/2 + 1) \tag{3.18}$$

complex numbers. $n/2 + 1$ is the count of the frequency bins.

The results are then filtered based on selected filers. This has a complexity:

$$O(f * (n/2 + 1)) = O(f * n). \tag{3.19}$$

The final complexity of the sample processing is

$$O(f * n + f * n * log(n) + f * n) = O(f * n * log(n)). \tag{3.20}$$

Listing 3.3: Implementation of the STFT inside of `TfModel`

```
std::vector<float> fftInput;
int zeroPaddedFrameSize = 0;
for (int f = 0; f < frameCount; f++) {
  auto begin = signal.begin() + f * hopSize;
  std::vector<float> frameSamples(begin, begin + frameSize);

  applyWindowFunction(frameSamples);

  //pad with zeroes
  zeroPaddedFrameSize = frameSize;
  while ((zeroPaddedFrameSize & (zeroPaddedFrameSize - 1)) != 0) {
    frameSamples.push_back(0.0f);
    zeroPaddedFrameSize++;
  }

  fftInput.insert(fftInput.end(),
      std::make_move_iterator(frameSamples.begin()),
    std::make_move_iterator(frameSamples.end()));
}

std::vector<std::complex<float>> spectrum =
    fftProcessor->process(fftInput, globalContext.get(),
  frameCount, zeroPaddedFrameSize);
```

3.3.2 Fast Fourier Transform Processor

The `FftProcessor` is implemented as a part of the `AlenkaFile` library. It's used as a part of the STFT implementation to calculate the hardware accelerated FTT. The `FftProcessor` utilizes the clFFT[22] library with OpenCL functions to compute the FFT. `FftProcessor` is connected to the global OpenCL context. To efficiently use the clFFT on multiple FFTs, it is required to send it the data in baches[28]. This is why the `FftProcessor::process` function accepts all of the frames required to be processed simultaneously.

First, the clFFT needs to be set up. An OpenCL queue is created in the global context. Next, a default plan for one dimensional FFT is created with layout set to `CLFFT_REAL` and `CLFFT_HERMITIAN_INTERLEAVED`. This means that the input buffer will be real numbers with no corresponding imaginary components and the output buffer will contain real and imaginary components without complex conjugates stored in the same array. The result location is set to be the same buffer as the input, the input values get overwritten. The plan needs to be updated every time the input batch size or the FFT size changes. Meaning that the frame count or the frame size input into the `FftProcessor::process` changes.

With the destruction of the `FftProcessor` object all plans, buffers, and queues have to be released.

3.3.3 Mesh Generation

The mesh generation process happens in the `TfVisualizer`. The processed data are received in a form of a `float` array with the frame count f, the x-axis, and the frequency bins count b, the y axis. The goal is to create a square grid with $f * b$ squares. First, a x-axis array with $f + 1$ and y-axis array with $b + 1$ equally distributed vectors is created. Both are then transformed and scaled into the desired mesh position. This has a time complexity:

$$(O(2(f + 1) + 2(b + 1))) = O(f + b) \tag{3.21}$$

The next step is generating the square grid from the x and y-axis arrays which has $O(f * b)$ time complexity.

3.4 User Interface

The GUI is implemented as stated in the design with a small addition of projection interpolation choice put into the scalp map pop-up menu. In both windows, the GUI in both windows is implemented with Qt framework[20] and various `QObjects` are utilized for the individual GUI elements. The side windows are extensions of the `QWidget` class that receives specific `QMouseEvents` whenever a user clicks inside of the side window. A `QMouseEvent` contains information about the clicked button and the clicked position. Depending on the implementation of the side window, the `QMouseEvent` triggers a specific action such as opening a pop-up menu.

The pop-up menu is created with the `QMenu` object which is connected to the global Qt context. Each menu item is implemented using the `QAction` object. Whenever the menu item gets clicked, the `QAction` gets triggered and a Qt signal is emitted to the global context.

The toolbar in `TfAnalyser` is implemented using various Qt layouts that contain text boxes, spin boxes, checkboxes, and their respective labels. The main menu layout is a horizontal `QHBoxLayout` that contains `QGridLayouts` and vertical `QVBoxLayout` with the toolbar items. Each menu item is a combination of `QLabel` and an interactive control element. The textboxes are implemented using `QLineEdit`, other elements use `QCheckBox`, `QSpinBox`, and `QComboBox`.

Each object has a specific way to ensure that the user can't enter a configuration that would break the application. `QLineEdit` uses `QIntValidator` to accept only integers in a certain range. Other objects have an inbuild solution to restrict the range of numbers.

3.5 Visualization

In the final step, the precalculated data is visualized on the screen. Both, `ScalpCanvas` and `TfVisualizer` are an extension of the `QOpenGLWidget`, which provides functionality for displaying OpenGL graphics integrated into a Qt application. `QOpenGLWidget` has three main graphics related methods, `initalizeGL()`, `paintGL()`, `resizeGL()`. The `initializeGL()` is called once before the `resizeGL()` and `paintGL()` is called. It sets up the OpenGL resources, such as programs and buffers, and state. The `resizeGL()` gets called when the window is resized. The `paintGL()` is the main paint method that gets call whenever the `ScalpCanvas` needs to be updated. I also use the `QPainter` to draw lines, text, and other simple 2D objects. Both, the `ScalpCanvas` and the `TfVisualizer` combine native OpenGL painting and the `QPainter`. Alenka offers an `OpenGLInterface` that's used for calling the OpenGL methods and error control. One of the things set up in the `initializeGL()` is the `Colormap` class which provides the color palette for the painting process.

3.5.1 Colormap

Colors are used as the main indicator of information in both implemented windows. ISARG uses color palettes with fewer colors for viewing topographic maps, mostly the red-white-blue colormap. In opposite to that, they use color palettes with more colors for spectrogram. I implemented the `Colormap` class as the main administrator of the color palettes used in Alenka. It hosts multiple color palettes that can be selected in the pop-up menu from the side windows.

A color palette is stored in the RGB format as an array of floats. Since there wasn't a requirement for high color variability, I hardcoded the color palettes into the `Colormap` class. Each color palette is stored as a set of a small number of major colors that get interpolated into the full-color palette. This solution could be later expanded by storing the palettes in `JSON` or with a completely customizable GUI solution.

The stored color palette is interpolated so that the final color array has a severalfold greater number of colors. This need has arisen during the implementation of the OpenGL visualization of the scalp map. Linear OpenGL interpolation leaves triangle artifacts that were partially overcome by using the nearest neighbor interpolation. This means that only the colors in the color array are used. A snippet of the colormap interpolation can be seen here 3.4.

The `Colormap` class also allows the color palette to be manipulated. The brightness and contrast can be changed. The brightness changes by moving the center of the color palette, the major colors are moved and a new color array is regenerated. Contrast is a value that multiplies the colors in the array.

Listing 3.4: Implementation of the color palette interpolation

```
int interpolationRegions = colorCnt - 1;
for (int i = 0; i < interpolationRegions; i++) {
  int firstColor = i * 3;
  int secondColor = firstColor + 3;

  float parts = colorPosition[i + 1] - colorPosition[i];
  for (int j = 0; j <= parts; j++) {
    float redC = (parts - j) / parts * colorTemplate[firstColor +
        red] + j / parts * colorTemplate[secondColor + red];
    float greenC = (parts - j) / parts * colorTemplate[firstColor +
        green] + j / parts * colorTemplate[secondColor + green];
    float blueC = (parts - j) / parts * colorTemplate[firstColor +
        blue] + j / parts * colorTemplate[secondColor + blue];

    int pos = colorPosition[i] * partitionSize + j * partitionSize;
    colormap[pos + red] = redC;
    colormap[pos + green] = greenC;
    colormap[pos + blue] = blueC;
  }
}
```

3.5.2 OpenGL Resources

To visualize the required data, OpenGL first needs to initialize the required resources. These resources include a vertex shader, a fragment shader, and data buffers. I also use a 1D texture for the color palette.

The shaders and texture are set up in the `initializeGL` method. I use Alenka's `OpenGLProgram` as a wrapper for setting up a shader program.

The texture is initialized by calling the `setupColormapTexture()` method. First, a texture buffer is generated, then the color palette array is loaden inside of the buffer with the `glTexImage1D()` method and parameters are set with `glTexParameteri`. The mag and min filters are set to `GL_NEAREST`. Finally, a sampler location is specified with the `glUniform1i()` method, so the shader program can use the texture.

There are two types of data buffers. One contains the processed EEG data in a form of triangles where each vertex is defined with its 2D coordinates and amplitude value. The second buffer contains indices, positions of the vertices inside of the vertex buffer. This is used because the triangle mesh contains many triangles that share vertices. The vertex buffer would contain multiple duplicates without the index buffer present. Since the amplitudes change constantly, data from both buffers are stored inside vectors and then loaded into OpenGL buffers during the `paintGL` call.

Index arrays are created during the mesh generation process. Each square

Listing 3.5: Triangle vertex shader used to draw spectrogram and scalp map

```
precision mediump float;

in vec2 currentPosition;
in float amplitude;

out float oAmplitude;

void main()
{
  gl_Position.xy = currentPosition;
  oAmplitude = amplitude;
}
```

inside of the spectrogram mesh is made up of two triangles and thus contains two duplicate vertices on the diagonal. The four non-duplicate vertices are stored as new vertices inside of the vertex array and four new indices and two duplicate indices are pushed inside of the index array. The scalp map is a little bit more complicated since many triangles share their edges. After the Delaunay triangulation, the result triangles are filtered. The same needs to be done for the newly created vertices in the middle of the edges during the triangle division process.

3.5.3 OpenGL Visualization

The OpenGL visualization happens in the `paintGL` method. First, the channel program containing both the vertex and the fragment shader is selected to be used. Then the vertex and index buffers are generated and bind. After that, the vertex array gets loaded into the vertex buffer. A vertex array is set up to tell the OpenGL that the vertex buffer is split into sets of threes, where the first two are the 2D coordinates and the last one is the amplitude. Then the index array gets loaded into the array buffer and a draw call is made with the `GL_TRIANGLES` parameter. First, the vertex shader is used to define the primitives and transform the input vertex into an output vertex. The interpolated output vertex is then received by the fragment shader which gets called on each fragment inside of the area defined by the vertex shader primitive. Both are rather simple, the vertex shader 3.5 splits the input vertex into coordinates and amplitude, sets up the position of primitive, and passes on the amplitude to the fragments inside of the primitive. The fragment shader 3.6 takes an interpolated amplitude value and finds a texture color that is the nearest to it. Lastly, the fragment shader sets the fragment color as the attributed texture color. After this, some of the resources are freed, namely, the buffers and vertex attribute array is disabled.

Listing 3.6: Triangle fragment shader used to draw spectrogram and scalp map

```
uniform sampler1D colormap;

in float oAmplitude;

void main() {
  vec4 color = vec4(texture(colormap, oAmplitude).rgb, 1.0f);
  outColor = color;
}
```

Listing 3.7: Vertex shader used to draw electrode positions in the scalp map.

```
in vec2 pointPos;

void main()
{
  vec2 st = gl_PointCoord.xy;

  vec3 color = vec3(0.6, 0.6, 0.6);

  float dist = distance(pointPos, st - 0.5);

  //dont touch pixels beyond radius of the circle
  if (dist > 0.5)
    discard;

  outColor = vec4(color, 1.0);
}
```

The drawing of the electrode position indicators in `ScalpCanvas` is quite similar. The difference is that instead of the whole mesh, only the original electrode positions are loaded into the vertex buffer, there is no index buffer, and `glDrawArrays` is called with the `GL_POINTS`. The used shaders are also different. The vertex shader is very simple. It is the same shader as used in the main window `Canvas` paint. All it does is setting the `gl_Position` and passing the coordinates onto the fragment shader. The fragment shader can be seen here 3.7. The fragment shader uses the received coordinates to paint a point which is then cropped based on the distance from its center to create a small circle.

3.5.4 Graphics Objects

Some of the elements in the windows are visualized with the use of `QPainter`. I created several helper classes that visualize the individual elements. One of the problems I ran into was that Qt uses a different coordinate system than OpenGL. OpenGL works in the -1 to 1 range on both x and y axes. A lot of Qt graphics objects take a point on the 2D axis and then its height and width. Qt objects also generally work with the real widget coordinates. I added a conversion from OpenGL coordinates to Qt into the implemented graphics objects so that the developer can think in a single coordinate system.

Objects drawn with `QPainter` include lines, window frames, and text. One of the main reasons I decided to use the `QPainter`is that drawing text is very difficult in OpenGL and would require another external library. `QPainter` is a fast solution that also uses OpenGL calls to draw objects and was already present in Alenka. The text is drawn using the `QPainter` method `drawText()`. Some of the text is rotated with the `traslate()` and `rotate()` functions. The lines are drawn with the `drawLine()` function and window frames with the `drawRect()` function.

3.6 Maintenance

3.6.1 Combining the Previous versions

One of the first things I did was combining two previous versions of Alenka into a single one[29]. The first version was from Martin Bárta. He developed the application further after finishing his thesis[5]. I had to fix an error that caused Alenka to lag every time step. The error was in the newly added `VideoPlayer`.

The second version was done by Lucas Morona[30]. I had to finish the `AQuestionDialog` as it was causing the application to crash.

3.6.2 Deployment

Deployment of Alenka was always a tedious task. To successfully run Alenka, the user has to have OpenGL and OpenCL drivers installed. Then either required dynamic Qt libraries or have the Qt installed. Alenka is distributed inside of an archive with the compiled program and required dynamic libraries. I wasn't able to run Alenka compiled on Ubuntu 18 on Ubuntu 20. I solved this issue with *linuxdeployqt*[31]. The deployer creates a single image with all of the required dependencies collected. Such an image created on a Ubuntu distribution can be then launched on other upstream distributions.

3.6.3 Documentation

I expanded the Doxygen[32] documentation already included in Alenka. I also created a wiki page on the projects github[29].

Testing

This chapter contains several testing methods that were applied at the end of the development process. I tested the application on several platforms. Then performed functional testing according to a premade scenario. In the end, I show three benchmark tests performed on dedicated graphics cars and using AMD emulation of OpenGL running on CPU on a virtual machine. The used graphics cards are *Nvidia GTX 1070*[33] and an older *Nvidia GTX 860m*. The virtual machine is running on a pc with *AMD 5 5600x*[34] processor. The full specification of the computers can be seen further in the text.

Cross-platform Testing

Two of the non-functional requirements are:

- Support of Microsoft Windows 7, 8, 10 and Linux Ubuntu 14 and 16
- Ability to run the application on a virtual machine

Table 3.1 shows various testing environments that Alenka was tested on.

Table 3.1: Testing Environments

	Nvidia 1070	Nvidia 860M	Virtual machine
Ubuntu 20.04		x	
Ubuntu 18.04			x
Ubuntu 16.04			x
Ubuntu 14.04			x
Windows 10 x64	x		
Windows 8 x64		x	
Windows 7 x64		x	

GUI and Functionality Testing

The following testing scenarios were performed to test the individual features and the GUI.

1. Scalp map window functionality test
    a) Open a new file
    b) Load electrodes into the track table
    c) Show scalp map (window → scalp map)
    d) Test the pop-up menu
        i. Show channels
        ii. Show labels
        iii. Change the colormap to jet (right mouse click inside of the window → color palette → jet)
        iv. Use different projection
        v. Set custom extrema to a different value
    e) Move the gradient from top to bottom and from bottom to top as much as possible (mouse left-click on the gradient and drag)
    f) Move the gradient from left to right and from bottom to top as much as possible (mouse left-click on the gradient and drag)
    g) Undock the window (move the window out of the original position)
    h) Dock the window (move the window to the original position)
    i) Dock the window to lower positon such that there will be two side windows at the same time
2. Scalp map window interaction with main window test
    a) Open a new file
    b) Show scalp map (window → scalp map)
    c) Load electrodes into the track table
    d) Test interactions with the main window
        i. Move the position scroll bar
        ii. Move the blue position indicator (move the mouse to the desired position and press the `T` key)
        iii. Change the high pass filter value
        iv. Change the low pass filter value
        v. Change the notch filter value
3. Scalp map window interaction with the track table test

a) Open a new file
b) Load electrode positions into the track table
c) Set the side windows in such a way that there are two side windows, first being the track manager and the second being the scalp map
d) Show channels and labels
e) Change the values in the track table
    i. Change the first two labels coordinates to zero
    ii. Load electrode positions into the track table

4. Time-frequency analysis window functionality test

a) Open a new file
b) Show time-frequency analysis (window → time-frequency analysis)
c) Change the colormap in the pop-up menu to inferno (right mouse click inside of the window → color palette → inferno)
d) Test the values in the top toolbar of the TFA window
    i. Change the channel to 5
    ii. Change the time to 6
    iii. Change the frame to 256 and the hop to 64
    iv. Change the hop to 33
    v. Set maximum frequency to half of the current maximum.
    vi. Set minimum frequency to half of the current maximum.
    vii. Change the filter window to Blackman.
    viii. Check and uncheck the $1/f$ compensation.
    ix. Check and uncheck the *log* compensation.
    x. Uncheck the freeze option.
    xi. Move the position indicator in the main window (move the mouse to the desired position and press the `T` key)
    xii. Move the position scroll bar
e) Test the bounds of the toolbar entry fields
    i. Try to set the time to -1
    ii. Try to set the channel to -1 and to a bigger value than the maximum in the file
    iii. Try to set the frame to -1
    iv. Try to set the minimum frequency to below 0
    v. Try to set the maximum frequency to a bigger number than the maximum possible
    vi. Set minimum frequency to 30 and try to set maximum frequency to 20

vii. Set set the maximum frequency to 50 and try to set the minimum frequency to 60

f) Move the gradient from top to bottom and from bottom to top as much as possible (mouse left-click on the gradient and drag)

g) Move the gradient from left to right and from bottom to top as much as possible (mouse left-click on the gradient and drag)

h) Undock the window (move the window out of the original position)

i) Move the position scroll bar

j) Dock the window (move the window to the original position)

k) Dock the window to lower positon such that there will be two side windows at the same time

Discovered Problems

Some problems were discovered during the functional testing:

1. Last image produced in scalp map gets left in the window and doesn't get deleted when there are no longer valid data to draw.

2. Changing the contrast and brightness of the colormap can cause a single black row to appear in the upper part for some color palettes. This then propagates to the triangle with a vertex that has maximum amplitude.

3. Setting custom extrema to low range around 0 causes the scalp map to not draw properly. Some of the triangles that should represent the maximum color are instead painted as minimum color.

4. There is no limit for channels in the TFA toolbar. User can set an invalid channel and kill the application.

5. Sometimes, wrong number values entered into text fields don't get fixed back to the old good value and the user has to reenter a good value.

I will try to fix these issues as soon as possible in a future patch.

Benchmarks

I performed several performance benchmarks of the Alenka application with the newly added features. One of the non-functional requirements is on the performance:

- Capability to visualize up to 256 channels with sampling frequency 2048 Hz.

Table 3.2: The TFA configuration used in all tests

Time	Frame	Hop	Min f	Max f	Window	1/f	log
10 s	128	16	0	max	hamming	true	false

Table 3.3: The scalp map configuration used in all tests.

Labels	Channels	Projection	Extrema
off	off	stereoprahic	local

Table 3.4: Configuration of the high-end pc.

Operating system	Windows 10 64-bit
Processor	AMD Ryzen 5 5600X; 6-Core; 3.7 GHz
RAM	16 GB DDR4; 3600 MHz
GPU	NVIDIA GeForce GTX 1070; 8 GB GDDR5
Secondary Memory	SSD 540MB/s

This is an upper limit requirement, standard EEG data files used by ISARG have 500 to 1000 Hz sampling frequency and 128 channels or less. I performed several benchmarks on different platforms. I used a standard file *standard_512Hz_128ch.mat* with real EEG data that has 512 Hz sampling rate and 128 channels. Another file *big_2kHz_256ch.mat* was generated for the purpose of this test and has 2048 Hz sampling rate and 256 channels.

Alenka has a `--printTiming` toggle that enables the benchmarking and prints the time interval between the start and the end of the main window data processing and single paint call. I expanded the benchmarking to include the methods that get called when the position in the main window updates. These are the amplitude update in the scalp map, the data update in the TFA, the loading of the data, and the STFT. Both paint calls, the one in `ScalpCanvas` and the one in `TfVisualizer` are included as well.

I set the main window to show 10 seconds of the signal and turned off all filters as Martin Bárta did in his benchmark[3]. For the TFA, I use the configuration that can be seen in this table 3.2. Ramos et al. experimented with the STFT parametrization for epileptic seizure detection in EEG signals[35]. They use window sizes of 64 and 128 and different hop sizes. A similar configuration was also used during the meetings with ISARG. Configuration for the scalp canvas can be seen here 3.3. The electrode positions are already stored in the track table for the test.

I used three different environments to test the application. A high-end Windows 10 desktop pc 3.4. A six years old notebook with Ubuntu that represents the mid-end to low-end pc 3.5. I also test on a virtual machine with Ubuntu 18.04 running on the high-end Windows pc.

I repurposed the testing scenario done by Martin Bárta in his benchmark

Table 3.5: Configuration of the mid-end pc.

Operating system	Ubuntu 20.04 64-bit
Processor	Intel Core i7-4710HQ; 4-Core; 2.5 GHz
RAM	8 GB DDR3; 1600 MHz
GPU	NVIDIA GeForce GTX 860M; 4 GB GDDR5
Secondary Memory	SSD 540MB/s

on the last version of Alenka[3] and use it in all tests.

Testing scenario:

1. Launch already preconfigured Alenka and open a new file.
2. Perform 10 movements to the right (press *PageDown*)
3. Perform 10 movements to the left (press *PageUp*)
4. Turn on the Notch filter
5. Perform 10 movements to the right (press *PageDown*)
6. Turn off the application

Window Benchmark

In the first benchmark, I test the whole application on the Windows 10 desktop pc. I ran the testing scenario four times, first with only the main window active, then adding the scalp map, only the main window, and the TFA, and finally all windows together.

The results can be seen in this graph 3.4. The graph shows the computation time in seconds of the individual steps in the testing scenario. The first tick for TFA measurements includes the TFA window setup, STFT, and first paint call. After that, there is a second tallest tick that represents the resources being initialized in the main window. Then the rest can be split into the following parts:

1. Moving to the right, data has to be read and allocated into the GPU memory in the main window.
2. Moving back, data is allocated and only has to be drawn.
3. Notch filter is turned on, this doesn't affect the TFA so there is no paint update.
4. Again, moving to the right, data has to be read again and processed with an activated filter.

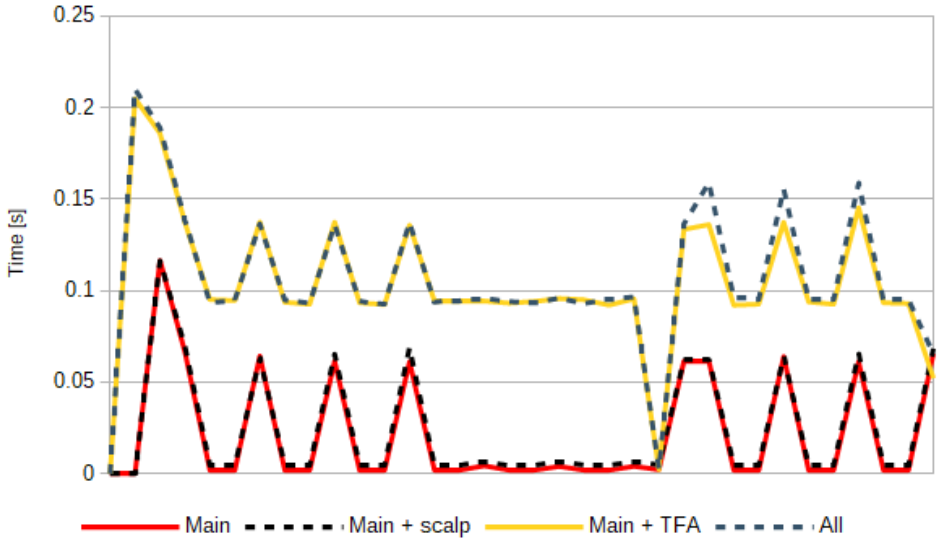

Figure 3.4: Graph of the window benchmark. The x-axis represents the individual steps in the test scenario.

Drawing the scalp map is very fast which can be seen on the graph, as the test with activated scalp map mostly copies the test with only the main window running. The TFA has its own data, it is separated from the main window calculations and increases the computation time linearly.

Platform Benchmark

The second benchmark focuses on the performance of the application on different platforms. I performed the test with the main window and the scalp map activated. I used the standard file and the test scenario from the previous benchmark. I tested the application in all three environments, the Windows pc, the Ubuntu notebook, and the virtual machine running on the Windows pc. Results of the platform benchmark can be seen here 3.5.

Both the notebook and the desktop have very similar results as they have a dedicated GPU. The virtual machine emulates the OpenGL on the CPU and is substantially slower. It's 4 times slower during the memory allocation times and 25 times slower when tracking back and only visualizing the precached data.

The delay between frames never goes beyond 100 ms on dedicated GPUs, which is a good result for a graphic application and feels rather smooth. The delay on a virtual machine isn't that much bigger and feels alright as well. I also tested the TFA on all platforms and the results confirmed the delay ratios from the window benchmark.

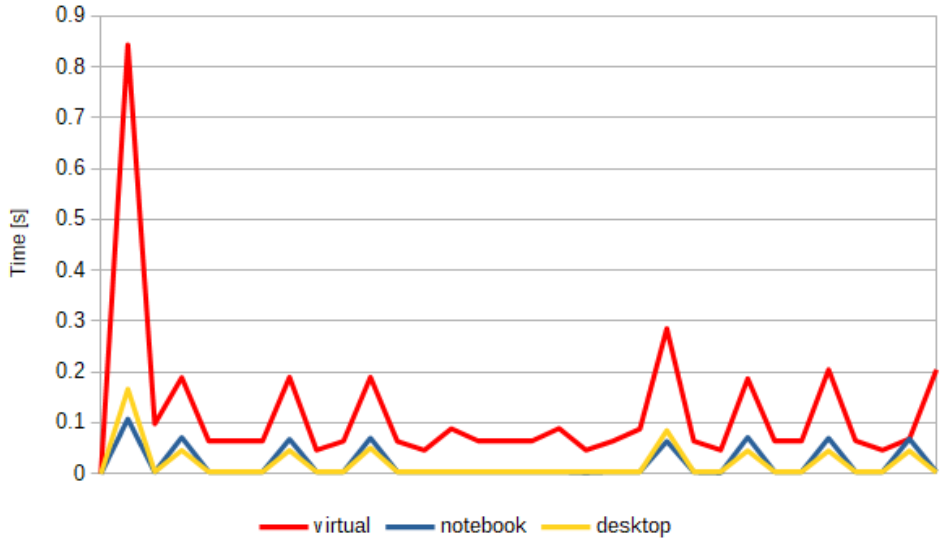

Figure 3.5: Graph of the window benchmark. The x-axis represents the individual steps in the test scenario.

File Benchmark

The last benchmark concerns the upper limit file requirement. I compared the performance of the standard 512 Hz and 128 channel file to the upper limit test file with 2kHz and 256 channels. I performed the test with the main window and the scalp map activated. The result can be seen in this graph 3.6

The difference between performance with the two files is minimal in the backtracking steps, where the data is only visualized. However, the difference when the data has to be loaded into the GPU memory is substantial. The amount of data that needs to be processed in the upper limit file is 8 times bigger, this means that the main window has to process $10 * 256 * 2000 * 4 \approx 20MB$ to visualize a single frame. However, the upper limit file takes 215 times longer to process in the main window. I also managed to load, process, and visualize the upper limit file in a virtual machine.

Although the delay between frames is substantial, I still conclude that the data upper limit was fulfilled as the application can run, process, and visualize the files successfully.

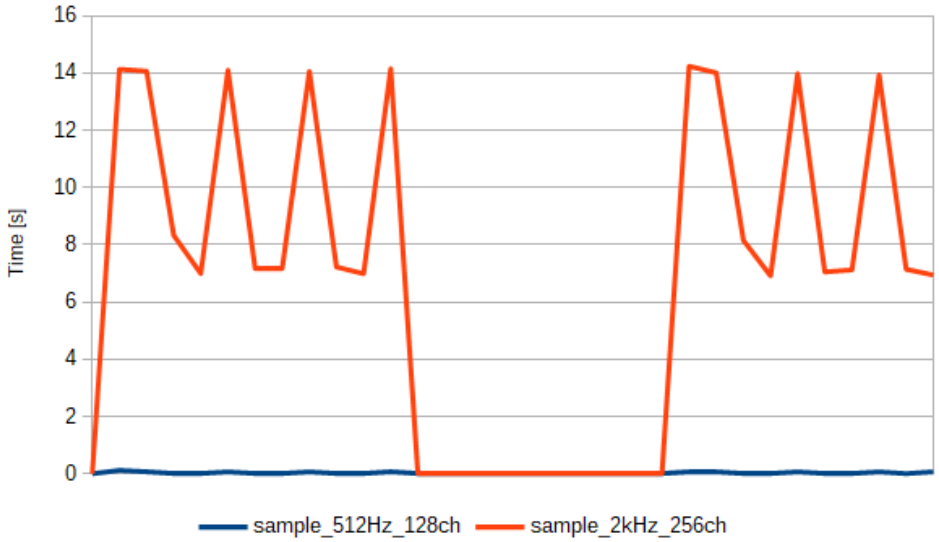

Figure 3.6: Graph of the file benchmark. The x-axis represents the individual steps in the test scenario.

Conclusion

I researched, designed, and implemented an extension of a multi-platform application that is capable to visualize EEG curves. The extension includes a topographical scalp map and time-frequency analysis(TFA) performed with the STFT and visualized with a spectrogram. I documented and tested both of the additions. The final application is capable to visualize up to 256 channels with a sampling frequency of 2048Hz. The application supports Microsoft Windows 7, 8, and Linux Ubuntu 14, 16, 18, and 20. It is also possible to run the application on a virtual machine.

The application is able to load three-dimensional electrode positions from a file, and transform them into two-dimensional coordinates using the stereographic projection. To visualize the electrical activity measured by the electrodes, the application first creates a triangle mesh with a help of the Delaunay triangulation. The mesh is then split into smaller triangles and the amplitudes at the newly created positions are interpolated using spatial interpolation. The final mesh is then visualized with OpenGL and its native interpolation to color the areas inside of the triangles.

The STFT is performed with clFFT library and its OpenCL functions that use hardware acceleration for parallel computation. The processed data from the STFT is transformed into a square grid and visualized as a spectrogram with OpenGL.

The implemented extension could be expanded in multiple ways. Different color interpolations could be used in the topographic map, especially interpolations that are highly customizable and take into account the specific criteria of different patients, such as different head shapes. The electrode positions could be also used in a three-dimensional visualization of the head with three-dimensional color interpolation. The TFA could be expanded with more advanced methods, such as the Wigner transform or the wavelet transform.

Bibliography

[1] ISARG: Intracranial Signal Analysis Research Group. `https://isarg.fel.cvut.cz/`, accessed: 2021-06-09.

[2] Proekt, A. *Brief Introduction to Electroencephalography*, volume 603. 01 2018, pp. 257–277, doi:10.1016/bs.mie.2018.02.009.

[3] Bárta, M. *Multiplatformní zobrazovací software pro elektroencefalografii*. Master's thesis, CTU, Prague, 2018.

[4] Bárta, M. Specializovaný systém pro zobrazování biologických signálů pacientů zařazených do epilepto-chirurgického programu: rozšiřovací moduly. Bachelor's thesis, CTU, Prague, 2015.

[5] Bárta, M. A Visualisation System for Biosignals. GitHub [Online]. 2018 [cit. 2021-6-10]. Available from: `https://github.com/machta/Alenka`

[6] Nunez, P. L.; Srinivasan, R. *Electric Fields of the Brain: The Neurophysics of EEG*. New York, NY: Oxford University Press, second edition, 2006, ISBN 0-19-505038-X, doi:10.1093/acprof:oso/9780195050387.001.0001.

[7] Maurer, K.; Dierks, T. *Atlas of Brain Mapping: Topographic Mapping of EEG and Evoked Potentials*. Berlin-Heidelberg-New York-London-Paris-Tokyo-Hong Kong-Barcelona-Budapest: Springer-Verlag, 1991, ISBN 978-3-642-76045-7, doi:10.1007/978-3-642-76043-3.

[8] Law, S. K.; Nunez, P. L.; et al. Topographical Mapping of Brain Electrical Activity. VIS '91, Washington, DC, USA: IEEE Computer Society Press, 1991, ISBN 0818622458, p. 194–201, doi:10.5555/949607.949639.

[9] Mitas, L.; Mitasova, H. Spatial Interpolation. In *Geographical Information Systems: Principles, Techniques, Management and Applications*, volume 1, edited by P.Longley, M.F. Goodchild, D.J. Maguire, D.W.Rhind, Wiley, 1999, ISBN 9780471321828, pp. 481–492.

[10] Böhner, J.; Bechtel, B. 2.10 - GIS in Climatology and Meteorology. In *Comprehensive Geographic Information Systems*, edited by B. Huang, Oxford: Elsevier, 2018, ISBN 978-0-12-804793-4, pp. 196–235, doi: 10.1016/B978-0-12-409548-9.09633-0.

[11] Chapter 2 - Geometric processing and positioning techniques. In *Advanced Remote Sensing (Second Edition)*, edited by S. Liang; J. Wang, Academic Press, second edition edition, 2020, ISBN 978-0-12-815826-5, pp. 59–105, doi:10.1016/B978-0-12-815826-5.00002-7.

[12] KHRONOS GROUP. OpenGL [Online]. c1997-2021 [cit. 2021-6-9]. Available from: `https://www.opengl.org/`

[13] Angel, E.; Shreiner, D. *Interactive Computer Graphics: A Top-down Approach with OpenGL*. Boston, Massachusetts, USA: Addison Wesley, 2012, ISBN 0-13-254523-3.

[14] KHRONOS GROUP. Rendering Pipeline Overview [Online]. 2012 [cit. 2021-6-9]. Available from: `https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview`

[15] Tzallas, A. T.; Tsipouras, M. G.; et al. The Use of Time-Frequency Distributions for Epileptic Seizure Detection in EEG Recordings. In *2007 29th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, 2007, pp. 3–6, doi:10.1109/IEMBS.2007.4352208.

[16] Hussin, S.; Sudirman, R. EEG Interpretation through Short Time Fourier Transform for Sensory Response Among Children. *1991-8178*, 04 2014: pp. 417–422.

[17] Zhivomirov, H. On the Development of STFT-analysis and ISTFT-synthesis Routines and their Practical Implementation. *TEM Journal*, volume 8, 2019: pp. 56–64, doi:10.18421/TEM81-07.

[18] Žiga Špiclin; Likar, B.; et al. Registration of EEG electrode positions to PET and fMRI images. In *Medical Imaging 2009: Image Processing*, volume 7259, edited by J. P. W. Pluim; B. M. Dawant, International Society for Optics and Photonics, SPIE, 2009, pp. 834 – 844, doi:10.1117/12.811892.

[19] Dobeš, P. Topografické mapování elektrické aktivity mozku. Bachelor's thesis, BUT, Brno, 2014.

[20] THE QT COMPANY. Qt [Online]. c2020 [cit. 2021-6-10]. Available from: `https://www.qt.io/`

[21] THE QT COMPANY. QPainter class documentation [Online]. c2021 [cit. 2021-6-15]. Available from: `https://doc.qt.io/qt-5/qpainter.html`

[22] ClMathLibraries: clFFT. GitHub [Online]. 2021 [cit. 2021-6-15]. Available from: https://github.com/clMathLibraries/clFFT

[23] Eberly, D. Least Squares Fitting of Data by Linear or Quadratic Structures: 5 Fitting a Hypersphere to Points. [online]. 2021 [cit. 2021-6-20]. Available from: https://www.geometrictools.com/Documentation/LeastSquaresFitting.pdf

[24] Delaunator-cpp. GitHub [Online]. 2021 [cit. 2021-6-20]. Available from: https://github.com/delfrrr/delaunator-cpp

[25] Delaunator. GitHub [Online]. 2021 [cit. 2021-6-20]. Available from: https://github.com/mapbox/delaunator

[26] Sinclair, D. S-hull: a fast radial sweep-hull routine for Delaunay triangulation. [online]. 2016 [cit. 2021-6-20]. Available from: https://arxiv.org/abs/1604.01428

[27] THE QT COMPANY. QWidget class documentation [Online]. c2021 [cit. 2021-6-17]. Available from: https://doc.qt.io/qt-5/qwidget.html

[28] ClMathLibraries: clFFT documentation. GitHub [Online]. 2021 [cit. 2021-6-15]. Available from: https://clmathlibraries.github.io/clFFT/

[29] Papinčák, M. A Visualisation System for Biosignals. GitHub [Online]. 2020 [cit. 2021-6-10]. Available from: https://github.com/papincakm/Alenka

[30] Morona, L. A Visualisation System for Biosignals. GitHub [Online]. 2020 [cit. 2021-6-10]. Available from: https://github.com/MoronaCzech1991/Alenka-master

[31] AppImage. linuxdeployqt. GitHub [Online]. 2020 [cit. 2021-6-10]. Available from: https://github.com/probonopd/linuxdeployqt

[32] van Heesch, D. Doxygen [Online]. 2021 [cit. 2021-6-15]. Available from: https://www.doxygen.nl/index.html

[33] Asus. ASUS Dual series GeForce GTX 1070 OC edition [Online]. 2021 [cit. 2021-6-23]. Available from: https://www.asus.com/Motherboards-Components/Graphics-Cards/Dual/DUAL-GTX1070-O8G/

[34] AMD. Ryzen 5 5600X [Online]. 2021 [cit. 2021-6-13]. Available from: https://www.amd.com/en/products/cpu/amd-ryzen-5-5600xl

[35] Ramos, R.; Olvera-López, J.; et al. Parameter Experimentation for Epileptic Seizure Detection in EEG Signals using Short-Time Fourier Transform. *Research in Computing Science*, volume 148, 10 2019: pp. 90–95, doi:10.13053/rcs-148-9-7.

[36] Fadzal, C.; Mansor, W.; et al. Short-time Fourier Transform analysis of EEG signal from writing. *Proceedings - 2012 IEEE 8th International Colloquium on Signal Processing and Its Applications, CSPA 2012*, 03 2012, doi:10.1109/CSPA.2012.6194785.

[37] KHRONOS GROUP. The open standard for parallel programming of heterogeneous systems [Online]. c2021 [cit. 2021-6-15]. Available from: `https://www.khronos.org/opencl/`

Appendix A

Acronyms

CPU Central processing unit

DTF Discrete-time Fourier transform

EDF European Data Format

FFT Fast Fourier transform

GDF General data format

GPU Graphics processing unit

GUI Graphical user interface

IDW Inverse distance weighted

ISARG Intracranial Signal Analysis Research Group

STFT Short-time Fourier transform

TFA Time-frequency analysis

TIN Triangulated interpolation network

HW Hardware

Appendix B

Controls

Controls tooltip can be displayed with the `help` application argument (see appendix C) during the application launch. The added windows can be activated in the top bar *window* section. They will then appear in the right-side window panel.

B.1 Main Window Controls

`MouseWheel` time step move

`PageUp` faster time step move left

`PageDown` faster time step move right

`Left and right arrow keys` slow time step move

`Ctrl + MouseWheel` change amplitude of the one selected channel

`Shift + MouseWheel` change amplitude of all channels

`Alt + MouseWheel` zoom

`Ctrl + LeftMouse` one-channel annotation

`Shift + LeftMouse` all-channel annotation

`C` cross off/on

`T` move position indicator to cursor position in the main window

`Ctrl + Z` undo

`Ctrl + Shift + Z` redo

B.2 Manager Side Windows Controls

`Ctrl + C` copy selected cells to the clipboard

`Ctrl + V` insert the clipboard inside the cells

`Delete` delete selected rows

`G` move main window visualization to the start of selected annotation

B.3 Gradient Controls

`LeftMouse + drag` inside of gradient to change the colormap

`MiddleMouse` inside of gradient to reset the colormap

B.4 ScalpMap Controls

`RightMouse` show pop-up menu

B.5 TFA Controls

`RightMouse` show pop-up menu

Appendix C

Alenka Help

Display help using the `./Alenka help` argument.

```
Usage:
Alenka [OPTION]... [FILE]...
Alenka --spikedet OUTPUT_FILE [SPIKEDET_SETTINGS]... FILE [FILE]...
Alenka --help|--clInfo|--version
Command line options:
--help help message
--config path override default config file path
--spikedet OUTPUT_FILE Spikedet only mode
--clInfo print OpenCL platform and device info
--glInfo print OpenGL info
--version print version number
--printTiming print the time it took to redraw Canvas
Configuration:
--mode val (=desktop) desktop|tablet|tablet-full
--locale lang (=en_us) mostly controls decimal number format
--uncalibratedGDF bool (=0) assume uncalibrated data in GDF
--autosave seconds (=120) interval between saves; 0 to disable
--kernelCacheSize count (=0) if 0, the existing file is removed
--kernelCacheDir path default is install dir
--gl20 bool (=0) use OpenGL 2.0 instead of 3.0
--gl43 bool (=0) use OpenGL 4.3 instead of 3.0; disabled
--cl11 bool (=0) use OpenCL 1.1 instead of 1.2
--glSharing bool (=1) use cl_khr_gl_sharing extension
--clPlatform ID (=0) select OpenCL platform
--clDevice ID (=0) OpenCL device
--blockSize val (=32768) samples per channel per block
--gpuMemorySize MB (=0) allowed GPU memory; 0 means no limit
--parProc val (=2) parallel signal processor queue count
--fileCacheSize MB (=0) allowed RAM for caching signal files
--notchFrequency f (=50) power interference filter
--resOptions list (=1 2 ...) resolution combo options
--screenPath path screenshot output dir path
--screenType type (=jpg) screenshot file type; jpg, png, or bmp
--matData val... data var names for MAT files; default is 'd'
--matFs val (=fs) sample rate var name
```

```
--matMults val (=mults) channel multipliers var name
--matDate val (=tabs) start date var name
--matLabel val (=header.label) labels var name
--matEvtPos val (=out.pos) event position var name in seconds
--matEvtDur val (=out.dur) event duration var name in seconds
--matEvtChan val (=out.chan) one-based event channel index var name
Spikedet settings:
--fl f (=10) lowpass filter frequency
--fh f (=60) highpass filter frequency
--k1 val (=3.65) K1
--k2 val (=3.65) K2
--k3 val (=0) K3
--w val (=5) winsize
--n val (=4) noverlap
--buf seconds (=300) buffering
--h f (=50) main hum. freq.
--dt val (=0.005) discharge tol.
--pt val (=0.12) polyspike union time
--dec f (=200) decimation
--sed seconds (=0.1) spike event duration
--osd bool (=1) use orginal Spikedet implementation
```

Appendix D

Build Instructions

Instruction have been generated from GitHub[29]. Written by Martin Bárta.

This page describes steps needed to build Alenka from source. The command-line examples can be run using bash (or git-bash on Windows which is included in git's installer).

Install Qt via the installer on their website. Select the `Qt 5.8 msvc2015 64/32-bit` package for Windows, or `Desktop gcc 64/32-bit` for Linux and Mac. Also select the `QtCharts` module.

Then download the third-party libraries using the preprepared script:

```
cd libraries
./download-libraries.sh
cd ..
```

You can use cmake-gui in place of cmake to change some of the following options to customize the build configuration:

- set `CMAKE_PREFIX_PATH` to specify Qt's install directory if needed (for example `/opt/Qt/5.8/gcc_64` on Linux, `C:/Qt/5.8/msvc2015_64` on Windows)
- on some systems you need to add to `CMAKE_PREFIX_PATH` the location of AMD APP SDK (e.g. on Linux set it to `/opt/Qt/5.8/gcc_64; /opt/AMDAPPSDK-3.0/lib/x86_64/sdk`)
- set `CMAKE_BUILD_TYPE` to switch between debug and release
- check ALENKA_STATIC_LINK to make a standalone Linux binary that works on both Ubuntu 14 and 16
- check ALENKA_BUILD_TESTS to build unit-tests

The rest of the instructions are OS specific.

D.1 Linux

First install the required tools and matio library. On some Linux distributions you need to also install OpenGL headers. On a Debian-like system you can do this by running:

```
sudo apt install git cmake-gui build-essential libmatio-dev curl
libgl1-mesa-dev
```

Then use the usual cmake/make combination to make an out-of-source build. From the repository's root directory use:

```
mkdir build-Release && cd build-Release
cmake -DCMAKE_BUILD_TYPE=Release ..
make
```

That should be all you need to do to build Alenka. Additionally you can use Qt Creator (or some other IDE) to open CMakeLists.txt file as a project. During project configuration redirect to the build directory we have just created. Then you can rebuild, run and debug the program directly from the IDE.

D.2 Windows

MSVC++ compiler can be acquired by installing Visual C++ Build Tools 2015. Choose `Custom Installation`, and uncheck all options but `Windows 8.1 SDK`. If you already have Visual Studio 2015, you probably don't need to install this.

Now you have two options: you can use Qt Creator to generate a makefile-based project, or use cmake to generate a Visual Studio solution. The first approach can use only a single thread for compilation which can lead to some annoying build times. By choosing the second option you lose some of Qt Creator's functionality designed to work specifically with Qt.

D.2.1 Qt Creator Project

Open CMakeLists.txt via Qt Creator and fill the configuration form that appears. Qt Creator with its default settings should take care of everything.

D.2.2 Visual Studio Solution

These commands generate a MS Visual Studio solution and then build Alenka.

```
mkdir build-Release && cd build-Release
cmake -G "Visual Studio 14 2015 Win64" ..
cmake --build . --config Release
```

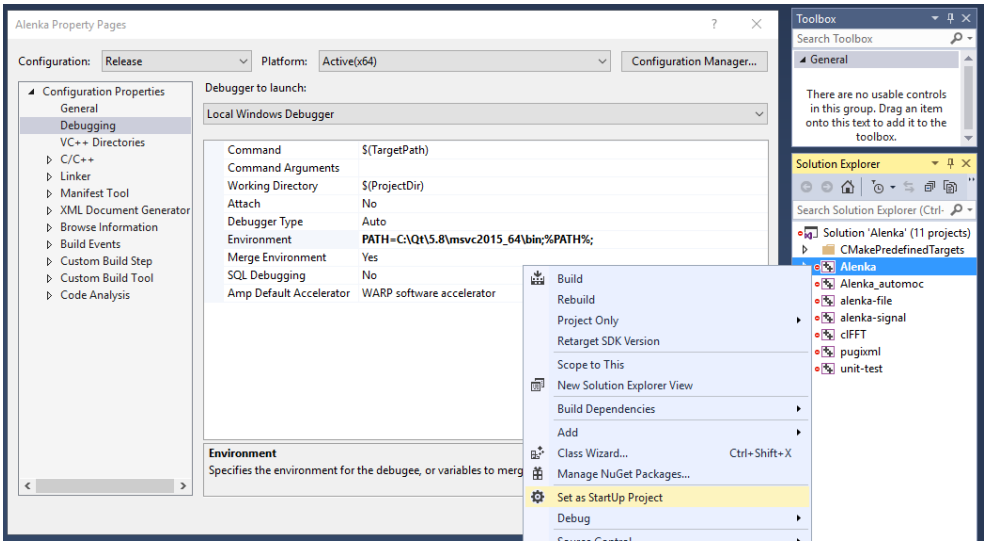

Figure D.1: Setup project in Virtual Studio. Original picture made by Martin Bárta.

Or you can skip the cmake build step and open Alenka.sln located in the build directory. Then select Alenka as the StartUp project, and set PATH to contain Qt's library installation directory. (See the picture D.1 for details. You can also set PATH system wide as an environment variable). Now you can run Alenka via the debugger button or F5.

Appendix E

Contents of enclosed DVD

- `DP_Marek_Papincak_2021.pdf` ..... thesis text as a PDF document
- `Ubu18-Alenka.zip` .. Image of virtual machine with preinstalled Alenka
- `doc`
  - `index.html` ..... link to documentation.
- `exe` ..... distribution packadges for Linux and Ubuntu
- `samples` ..... test sample files with electrode elc files
- `src`
  - `impl` ..... implementation source
  - `thesis` ..... thesis text source in LaTeX