ASSIGNMENT OF MASTER’S THESIS

Title: Web based visualization of latest generation hybrid active pixel detector data
Student: Bc. Jiří Vycpálek
Supervisor: MSc. Benedikt Ludwig Bergmann
Study Programme: Informatics
Study Branch: Web and Software Engineering
Department: Department of Software Engineering
Validity: Until the end of winter semester 2017/18

Instructions

The goal of the work is to develop an interactive visualization tool for Timepix3 data. The Timepix3 is a latest generation pixel detector developed at CERN. Unlike its predecessor’s frame-based approach, the Timepix3’s event-driven read-out scheme significantly increases the amount of data. Therefore, an efficient storage structure is required. Without the frame as a natural unit of visualization, a new approach for data presentation is needed.

Design an efficient database system for I/O operations capable of handling huge datasets (compression vs. speed).
Define an application interface for versatile query structure and integral data retrieval based on a customized web service similar to REST featuring JSON.
Develop a server application for data processing and a platform-independent client software tool (based on HTML) for visualizing the data in interactive 1D and 2D charts locally or remotely allowing client side filtering of interesting events.

References

Will be provided by the supervisor.

L.S.

Head of Department Dean

Prague February 24, 2016
Master’s thesis

Web based visualization of latest generation hybrid active pixel detector data

Bc. Jiří Vycpálek

Supervisor: MSc. Benedikt Ludwig Bergmann

30th June 2016
Acknowledgements

I would like to thank my supervisor Benedikt Bergmann for providing a thorough guidance and useful feedback throughout the development of the application, as well as writing this thesis. Also for providing relevant materials, explaining some important concepts and teaching me some related physics knowledge.

I would also like to thank everyone who supported me while I was writing the thesis, my friends, my girlfriend, my family. Thanks to them for help, for patience and for understanding, when I was focused on the thesis and not being exactly the most attentive, considerate and fun person to be around.

Special thanks also goes to my colleagues at IEAP and the whole institute itself, for providing very nice and welcoming working environment and the opportunity to work on this project.
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 particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 30th June 2016
Abstract

The aim of the thesis was to create an interactive visualization tool for data from the Timepix3, latest generation pixel detector developed at CERN. It describes the analysis, design and implementation of an application processing the data, storing them in an efficient database-like system, and displaying them in a web-based visualization tool with various useful tools for examining the data.

Keywords Visualization, Detectors, Timepix, Timepix3, Data processing
Contents

Citation of this thesis .............................................. viii

Introduction ................................................................ 1
  Timepix detectors and their applications .................. 1
  Frame based scheme .............................................. 2
  Pixelman .......................................................... 2
  Thesis structure .................................................. 2

1 Timepix 3 ............................................................... 5
  1.1 Data read-out ................................................ 5
  1.2 Event driven read-out scheme ......................... 6
  1.3 Detector data storage ..................................... 7
  1.4 Faults and data corruption ............................. 7

2 Requirements ......................................................... 9
  2.1 Functional requirements ................................ 9
  2.2 Non-functional requirements .......................... 11

3 Analysis and design ................................................ 13
  3.1 Overall application design .............................. 13
  3.2 Languages and technologies .......................... 14
    3.2.1 C++ ..................................................... 14
    3.2.2 ROOT ................................................... 15
    3.2.3 Node.js ............................................... 15
    3.2.4 HTML5 ............................................... 15
    3.2.5 TypeScript ........................................... 16
    3.2.6 Sass .................................................... 16
  3.3 Application architecture ................................. 17
  3.4 Back-end application ..................................... 17
  3.5 Data storage & database ............................... 17
3.6 Server and API .................................................. 19
3.7 UI & UX design .................................................. 20
  3.7.1 Target user group ........................................... 20
  3.7.2 Usability ..................................................... 21
  3.7.3 UI layout & wireframing ................................... 21
  3.7.4 Responsivity ............................................... 22
  3.7.5 Colors and graphic design ................................ 23
3.8 Front-end application ......................................... 23

4 Data processing – backend ....................................... 27
  4.1 Raw data conversion .......................................... 27
    4.1.1 Data sorting ............................................. 28
    4.1.2 Trigger-based approach ................................ 28
    4.1.3 Error detection & repair ................................ 29
  4.2 Data storage & database ...................................... 31
    4.2.1 ROOT files .............................................. 31
    4.2.2 Data compression ....................................... 32
    4.2.3 Tree structure .......................................... 32
    4.2.4 ROOT database ......................................... 33
    4.2.5 Indexing ................................................ 33
  4.3 Processed data retrieval ...................................... 35
    4.3.1 Fast interval retrieval .................................. 35
    4.3.2 Data serialization ...................................... 36

5 Data visualization - frontend ..................................... 37
  5.1 Application structure ........................................ 37
  5.2 Data retrieval ............................................... 38
    5.2.1 Time interval querying ................................ 39
    5.2.2 Animation ............................................... 40
  5.3 Data filtering ............................................... 40
    5.3.1 Pixel property filtering ................................ 41
    5.3.2 Time filtering .......................................... 41
    5.3.3 Custom user filtering .................................. 41
  5.4 1D Visualization ............................................. 42
    5.4.1 Binning .................................................. 42
    5.4.2 Trigger detection & display ............................. 43
    5.4.3 Data drawing ............................................ 43
    5.4.4 Timeline scale ......................................... 44
    5.4.5 Timeline interactivity ................................... 44
    5.4.6 Data updating .......................................... 45
    5.4.7 Filtered data display ................................... 46
  5.5 2D Visualization ............................................. 46
    5.5.1 Data management ........................................ 48
    5.5.2 Data display ............................................ 48
List of Figures

1.1 Operation of Timepix3 pixel in ToA/ToT mode . . . . . . . . . . . 6
3.1 Architecture diagram . . . . . . . . . . . . . . . . . . . . . . . . . 18
3.2 Class diagram of data processing part of the application . . . . . 19
3.3 Wireframe of the application . . . . . . . . . . . . . . . . . . . . . 22
3.4 Wireframe of the application – mobile version . . . . . . . . . . . 24
3.5 Application color scheme . . . . . . . . . . . . . . . . . . . . . . 25
4.1 Error detection performance – small instances . . . . . . . . . . . 30
4.2 Error detection performance – large instances . . . . . . . . . . . 31
4.3 Scheme of the ROOT database storage with indexing . . . . . . 34
5.1 Class diagram of front-end application . . . . . . . . . . . . . . . 38
5.2 Display of the filtered data on the timeline . . . . . . . . . . . . . 46
5.3 2D sensor histograms visualization . . . . . . . . . . . . . . . . 47
5.4 Zooming process . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
5.5 Screenshot of final application . . . . . . . . . . . . . . . . . . . . 54
7.1 Multi-sensor practical application . . . . . . . . . . . . . . . . . . . 63
7.2 Wireframe of multi-sensor view . . . . . . . . . . . . . . . . . . . . 64
List of Tables

4.1 Compression factors ........................................... 32
Introduction

The topic of this thesis is the creation of an interactive visualization for Timepix3 data. The Timepix3 is the latest generation of pixel detectors and the successor of the Timepix.

Timepix detectors and their applications

The Timepix is a hybrid active pixel detector developed within the Medipix collaboration at CERN (European Organization for Nuclear Research) as the successor of the Medipix2 chip [1]. Its 1.96 cm² of sensitive area are flip-chip bump bonded to the readout ASIC (Application Specific Integrated Circuit). It is this ASIC, which segments the active area into a matrix of 256x256 square pixels of 55 µm width. Each pixel can be configured independently in either one of the three different modes: counting mode, ToT mode and arrival time mode.

In counting mode, a counter is incremented once for each received signal above certain threshold, in ToT mode it is incremented continuously as long as signal stays above a threshold and in arrival time mode, the counter is incremented continuously from the time the first hit arrives until the end of the frame acquisition [2].

Timepix, Timepix3 and hybrid pixel detectors in general have wide range of applications in various fields. Timepix detectors are being used for example in the ATLAS experiment at CERN, where they replaced the Medipix pixel detectors. There, they provide real-time information about the radiation fields at several positions, the activation of surrounding material, and the luminosity [3, 4]. They are also used in X-ray radiography [5], ION beam therapy [6] or for example together with MRI scans for tumor or brain damage diagnostics [7]. In all these applications, Timepix3 detectors will replace the current technology due to their superior performance.
Introduction

Frame based scheme

The Timepix detector utilizes a frame-based readout. That means, data from every pixel placed in the matrix are read out at once as a single unit – the frame – representing the state of the whole sensor (all pixels) in a given time. These frames are written to disk, from where they can be loaded into the visualization and analysis software.

Pixelman

The state-of-the-art data acquisition and processing software is Pixelman, developed by the Institute of Experimental and Applied Physics, CTU in Prague [8]. It is a desktop application with Java GUI, built on highly modular architecture supporting C++, as well as Java plugins [9]. It supports all the older Medipix based devices (Medipix 2.1, Medipix2 MXR, Timepix, Medipix 2.1 Quad, Medipix2 MXR Quad, Timepix Quad) [10].

However, it can only handle the frame based data acquisition scheme and not the data driven acquisition, which is used in Timepix3 and will be described in the following chapters.

Thesis structure

The thesis describes how an interactive web application was developed for displaying a continuous stream of data, without a predefined unit of visualization such as the frame.

In the first chapter the characteristics of the Timepix3 detector and the nature of the data it produces are introduced and described. In the second chapter, a detailed list of requirements is given. The third chapter is focused on the analysis of tools used for development and on design of the application. The implementation itself is divided into three parts:

- Implementation of the back-end part of the application (in chapter 4) is focused on creating data processing tool for converting raw data from detectors into efficient database-like storage suited for fast data retrieval.

- Implementation of the frontend (chapter 5) describes providing comprehensive toolset for the client part of the application, exposing a wide range of useful tools for physicists examining the particle data.

- Implementation of the server and communication (chapter 6) is focused on connecting all the parts of the application together, creating a native Node.js add-on built on Chrome’s V8 engine, wrapping all the written C++ code with its external scientific libraries’ functionality and bringing it to web.
In the end, the future of the application and possible further improvements are outlined and the whole work is summarized and concluded.
Chapter 1

Timepix 3

Timepix 3 is a hybrid active pixel detector readout chip, which uses a data driven readout scheme. The functionality is similar to the Timepix device described above, pixels are also arranged in a 256x256 matrix of 55 µm x 55 µm. In contrast to the Timepix, it allows measuring of the ToA and ToT simultaneously in each pixel.

ToA (Time of Arrival) is a time stamp of an event happening and its range is 14 bits, ToT (Time over Threshold) is a charge measurement of the pixel and is stored in 10 bits. The output data packet is always 48 bit long for each detected hit.

For the purpose of this thesis, the processed sensor data are of ToA/ToT-type, which means in the data, ToA as well as ToT values will always be available. In other modes the missing values are replaced by dummy values [11], so concerning the visualization, the user can ignore not selected values.

1.1 Data read-out

The way how the ToA and ToT values are obtained is done analogue to the Timepix. Whenever a hit occurs, there is a separate synchronized pulse, which rises with first rising clock edge. The ToA value is latched into a register in time of rising edge of this synchronized pulse and the ToT is the number of clock counts happening while this synchronized pulse is high. The clock has a frequency of 40 MHz, therefore the time resolution is 1/40 MHz = 25 ns.

Additionally, Timepix 3 provides readout in fast timing mode, which allows ToA measurements to be more precise (resolution of 1.56 ns [11]). It means that there is a second, faster clock, which starts at the exact arrival of the hit, ends at the following rising edge of the pixel clock (where the normal ToA is calculated) and allows to store this time difference in the fToA value (fast Time of Arrival). The frequency of this second faster clock is 640 MHz,
1. **Timepix 3**

![Figure 1.1: Operation of Timepix3 pixel in ToA/ToT mode](image)

therefore the resulting time resolution is $1/640 \text{ MHz} = 1.56 \text{ ns}$.

It is possible to see the described hit acquisition process in figure 1.1. The first line in the figure is “shutter”, which is essentially an on/off switch, deciding whether acquisitions are made or not. The second line is the standard 40 MHz clock, which counts the ToT and ToA values. The orange “discr” line is a representation of the individual pixel hits and it is possible to see the described synchronized signal “discr sync”. The 640 MHz clock for measuring the fToA value in fast timing mode is marked “fToA clk” and it is possible to see, that it starts at the arrival of a pixel hit and measures the time until the ToA value is latched.

### 1.2 Event driven read-out scheme

The most important difference between Timepix and Timepix3 is the data driven readout mode – it means, that the readout chip sends packets of data with each pixel’s coordinate, ToT and ToA immediately after the hit is processed by the pixel [12]. Therefore, there isn’t a natural unit of visualization in the form of a frame, as it was with Timepix devices.

The data are instead sent as a continuous stream, each single event individually with its properties. Thus, a new approach to process and visualize such data is needed.
1.3 Detector data storage

Currently, data received from sensor are stored in ASCII format in a single large file. Each line in the file represents one event on the sensor and consists of the following values:

- **Index:**
  Arbitrary index, marking the order at which the events were written to the file.

- **Matrix Index:**
  Pixel index in the matrix. The matrix is an array of 256x256 entries. It is both pixel X and Y coordinates encoded in one single number in following way: $pixelX \cdot 256 + pixelY$.

- **ToA:**
  Value stored in clock counts.

- **ToT:**
  Value stored in clock counts.

- **fToA:**
  Value stored in fast clock counts.

Processing data from ASCII format is very inefficient and the size of the files is a problem, as it ensures slow access to the data and sometimes, due to its size, it is not even possible to open and browse the data by any standard means. The data also have to be processed (some calculations have to be made) to get the real accurate values for each event. In the data, there can also be empty lines (all values zero), which indicate that a so-called trigger was received by the sensor – ToA registers are reset, so that the time measurements from this point onward starts again from zero. Therefore, the real ToA value has to be calculated using ToA and fToA, taking the trigger signals into account.

1.4 Faults and data corruption

Several possible errors can occur in the data. Firstly, data in the file are not sorted chronologically, because some acquired hits may be stored in the pixel longer than expected, waiting for readout, and their values can be written to file later than data of pixel hits with higher ToA. This delay in write-down can be especially problematic, if it occurs at the edge of a trigger period, as it prevents a simple sorting of the data trigger-by-trigger from solving the unorderness problem.
1. Timepix 3

Secondly, there can be an error in the readout and the pixel values are wrong – resulting for example in a negative ToA value.
Requirements

2.1 Functional requirements

The requirements for the program which is the subject of this thesis are as follows:

- Database system:
  Design an efficient database system capable of handling huge datasets with considering the competition between compression and speed.

- Process the raw data from sensors:
  Find and implement a way how to process the raw data produced by the detectors, which are now stored in large ASCII files, and convert them into the designed database format.

- Application interface:
  Define an application interface for versatile querying of data, allowing a fast and efficient retrieval of the data selected by the specified time interval.

- Server:
  Develop a server application, handling the above mentioned data processing functionality. It should be provided as an online service.

- Client tool:
  Develop a platform-independent client software tool (based on HTML) for visualizing the data.

- 1D Visualization:
  Create a timeline, visualizing the data received from the sensor in time – should be implemented as a bin chart, each bin containing number of pixel hits within certain short time.
2. Requirements

- Time interval & bin size selection:
  Let the user select the size of the time interval currently displayed (its start time during the measurement, as well as its size, both in seconds) and also the size of bins displayed on the timeline. It should not be possible to set bins smaller than a minimum given by the screen resolution.

- Zooming in certain areas:
  Create two interactive handles on both sides of the timeline, which the user can drag within the timeline to limit displayed data only to certain selected region.

- Show next/previous period:
  Add buttons to query the server and receive the data from time interval of the same size as the current one, only immediately following or preceding, respectively.

- Animation mode:
  Implement an animation mode, which will automatically, periodically, query the server and display the data from next time interval of the same size. The visualization will update, so that the user can see its progress in time.

- 2D Visualization:
  Create a 2D square histogram visualization, representing the sensor and its matrix of 256x256 pixels and display the pixel hits and their values on it.

- Show ToA/ToT/#hits:
  Separate the 2D visualization into 3 different squares, one for each mentioned properties. The displayed ToA is the latest ToA value of the current pixel (in ns), the displayed ToT is sum of particle charges (in clock counts) of all hits on the current pixel and the displayed #hits is the number, how many times a pixel was hit during selected time interval. Display these values plus real X and Y sensor coordinates of the pixel while the user hovers over them.

- Zooming:
  Allow the user to zoom freely into any area of the histogram by click-dragging.

- Display the histogram values:
  Display the minimum and maximum values of the displayed data and indicate the pixel values by their colors for each screen.
2.2. Non-functional requirements

- **Color maps:**
  Provide the user with the possibility of changing colormaps of the visualization (use standard ones – Greyscale, Jet, Hot), with the possibility of setting white background in “Jet” colormap.

- **Linear/Logarithmic scale:**
  Allow the user to change the displayed color values of each pixel to be scaled according to linear or logarithmic scale.

- **Filtering based on values:**
  Implement client-side filtering of interesting events – display only such pixel values, which match user-selected criteria (min and max values for ToA, ToT, Trigger number, fToA and sToA).

- **User specified filter:**
  Allow the user to manually define simple expression for data filtering in C++ syntax. For example “fToA % 2 == 0”, which would filter all data to only display those with even fToA value.

- **Show statistics of data:**
  Display on screen total number of pixels currently shown, their summed energy value and the number of trigger periods they contain.

- **Show warnings and errors:**
  In any case of error or anomaly in the received data (for example negative values), display such information to the user in the form of warning, or error.

2.2 Non-functional requirements

- **Data converting part:**
  The data converting part will be available as an application ran from command line on Linux/Unix based systems.

- **Server part:**
  The server application should be also compatible and deployable on Linux servers.

- **Client part:**
  The visualization part of the application will be platform-independent. It will be possible to run it on Unix based system, as well as Windows.
Analysis and design

With all the requirements known, it is necessary to analyze and decide how the application will be built to meet those requirements. In this chapter, specification and outline how the application is designed to work, to be structured and look, are be presented.

3.1 Overall application design

As it is already clear from the problem specification and functional requirements, the application will be separated into several independent parts. It is advantageous to promote modularity and allow for example future replacement, or easier changes to only one of the parts of the application.

Therefore, the application will consist of an independent client part, where the data will be visualized and where all the data filtering will take place. The client will retrieve data from the second part of the application – the server. The server will provide the data in a format easily processable by the client and will be able to send them based on simple queries, typically time specifications. The amount of data will be usually huge, and storing them on server can offer much more storage space and, at the same time, access for several clients. The users will be able to see only a fraction of the data, without loading the whole files, and choose to filter only some of them, optionally downloading them.

The fact that data filtering will take place on a client side brings many advantages – for a lot of users’ actions, there will be no network communication, it will save bandwidth, and it will also improve performance, as actions do not rely on network communication and speed, and take place on the user’s machine, which can provide better computational performance. Current data could also be available in case of temporal server downtime and the server is in general less loaded.

Concerning time, computational and functional requirements, the applic-
3. Analysis and design

Analysis and design is also designed in a way, that there will be a third part, which is run separately and independently and with no connection to the other two parts – the converter. Because of the size and format of data (ASCII file – which is very inefficient and large) and the time it would take the server to process them directly, the converter will transform these data to a more easily processable form, store them into a database, and save them to disk. These files will then be used by the server (loaded from disk) to quickly provide the correct pixel data.

This pre-processing of data will run in different time than the rest of the application. Thus the time constraints are not an important concern. The emphasis is put on providing the data in the best possible format. That means, all sorting, error repair and indexing is done in this stage.

3.2 Languages and technologies

To cater to the needs of the required end product, it is necessary to choose the right set of technologies and languages which the application will be built on. In this project, the goal is to choose the path of exploring the capabilities of combining the best currently available, modern and elegant web technologies with proven efficient and fast relatively low-level language for computational intensive data processing. For the reasons explained more in detail in the next sections, the following stack of languages and technologies was chosen: the data processing part of the application will be written in C++ using the ROOT libraries, the server will run on Node.js, front-end client part will be written in HTML5+CSS3(Sass) with the visualization itself utilizing the HTML5 canvas element and for providing the application logic and dynamic features – TypeScript with using jQuery²

3.2.1 C++

The C++ programming language was chosen for this project for its superior runtime and memory management performance over its alternatives. It can achieve as much as three times the runtime performance in median compared to Java [13, 14] and can provide very good results (runtime and memory consumption) in list as well as file I/O operations (which will be used in this project), especially in larger amounts of data, even when compared for example to scripting languages like Python [15]. Also, in general, C++ is more suited for low-level functionality, as required in the data processing part. Another reason for choosing C++ is to have more control in hands of the programmer. It allows easier and more precise optimization of the code and its performance and manually manage resources and memory.

²https://jquery.com/
Apart from C++’s general advantages, it was chosen also for reasons more circumstantial and more specific to this project. A very important factor is interoperability and common ground within the community of people who will be interacting with the resulting project. This application is developed at IEAP, where a number of similar programs are made in C++ and the programmers understand this language. Closely tied with the community is also the last reason for choosing: the ROOT library.

### 3.2.2 ROOT

ROOT[^root1] an open source project coordinated by the European Organization for Nuclear Research, CERN in Geneva[^16], is a scientific software data analysis framework widely used in (high-energy) physics. It can save any data in a compressed binary form into its own ROOT file format. ROOT provides a tree data structure, that is extremely powerful for fast access of huge amounts of data – orders of magnitude faster than accessing a normal file[^17]. It takes care of efficient data storage, caching, parallel processing, spanning too big amounts of data over several files, etc. It provides its own graphical user interface to browse stored data, but can also be incorporated within a C++ program.

### 3.2.3 Node.js

Node.js[^node1] is an asynchronous JavaScript runtime built on Chrome’s V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. It is designed to build scalable network applications[^18]. It is the proven go-to option for building modern web applications[^19] and it has great functionality, efficiency and documentation.

As it is built on Chrome’s V8 JavaScript engine, which is written in C++, it provides the possibility to create native C++ written add-ons (as dynamically linked shared objects). This feature is convenient, as it provides one of the possible options how to interconnect the code written in C++ and a web server.

Node.js allows very fast and efficient development. It is very well suited for creating APIs and required query structures with relatively low effort, because of the way how it is conceived and the fact that it is especially made for this purpose from its creation.

### 3.2.4 HTML5

HTML5 is a fifth version of the core language on the World Wide Web, the Hypertext Markup Language[^html5]. It introduces a lot of new features, elements

[^root1]: https://root.cern.ch/
[^node1]: https://nodejs.org/en/
[^html5]: https://www.w3.org/TR/html5/
3. Analysis and design

and attributes to make coding of web pages easier and more efficient. The most important new added feature for this work is the canvas element – it is an element built to be able to work with graphic primitives and images. It is possible to draw on a canvas and manipulate it via client-side scripting language. This is perfectly suited for interactive visualization tasks.

3.2.5 TypeScript

For modern application development and more complex client-side browser-based programs, the obvious choice, which would be pure JavaScript, is somewhat lacking in terms of features, which other application programming languages provide. For example, objective oriented programming in JavaScript is achieved by function prototyping and extending, which can result in less understandable code.

TypeScript introduces qualities to JavaScript that are needed for, or at least bring a lot more convenience into, large scale productive application development and using common practices. TypeScript is build over (and compiled into) JavaScript, uses its own compiler and produces clean code, which runs on any browser. TypeScript introduces a native support for OOP features like classes and interfaces, namespaces and a lot more (with similar syntax to languages like Java or C#). Unlike JavaScript, Typescript is a typed language (although it still supports “untyped” variables, which are assigned to a type when first used). Thus, in general, it provides better control and readability for large projects. But, at the same time, it only extends the syntax and semantics and therefore it is still possible to use anything from JavaScript (code, libraries, whole files) [20].

In brief, TypeScript allows the programmer to write clean, easy-to-read, efficient code which is reusable, modular, practices modern objective oriented design patterns, is easy to build on, scale, and still provides all the advantages of the powerful JavaScript.

3.2.6 Sass

Sass (Syntactically Awesome StyleSheets) is an extension of the CSS language, compatible with all versions of CSS. It adds support for nested rules, variables, selector inheritance, various useful functions and more. It is compiled into well-formatted CSS, makes it easier to keep all the project stylesheets organized, maintained, and, in general, makes the process of styling smoother. With the use of the Compass framework, it is possible to use countless of cross-browser CSS3 properties, helpers, and functions which further enhance the functions provided by Sass. Compass also comes with

3.3 Application architecture

Compass Watcher to compile all the stylesheets immediately during development.

3.3 Application architecture

The application architecture is illustrated in figure 3.1. It shows that the client part of the application, run in the browser on JavaScript, connects with the server via http protocol to receive the data. On the server part, it is possible to see the components and the flow of the data. The TPX3 Data Converter is a stand-alone program, run manually by the administrator and it produces the ROOT Database data. Root Data Facade works with this data. It is connected to the Node.js part of the application as a native add-on (linked shared object).

3.4 Back-end application

The data processing part of the application on the backend is structured into classes in a way as shown in figure 3.2. The first, important class is the AsciiToRootConverter. Although providing only one public method, it will encapsulate a large portion of functionality. It will be responsible for converting the data from ASCII files (each line of the file, recorded hit by the sensor, represented by SingleEvent class) into the ROOT database and performing all the calculations that are needed on the data, in order to get the correct values. It will also sort the data before writing them into the ROOT file. RootFile is a wrapper over the actual physical ROOT file, saved to disk and manages I/O operations with it.

The RootDataFacade is used by the main entry point of the server application. It provides API needed for manipulating with the pixel data, which will be sent to the client. This interface will strongly correspond with the API of the web application. The most significant public method is GetInterval(), which will return an instance of PixelInterval containing the data. PixelInterval is a list of single actual pixel hits on the sensor (SinglePixel) consisting of the correct calculated values, such as the ToA in nanoseconds. PixelInterval can be converted into some standardized format for transferring to JavaScript (to client), such as string encoded JSON object.

3.5 Data storage & database

The database is designed in such a way, so that it stores the data of each pixel. Each pixel entry in the database contains the following values in the specified type:

- Pixel Index – 64 bit unsigned integer
3. Analysis and design

Figure 3.1: Architecture diagram
3.6 Server and API

The web application part of the project will be implemented as RESTful server. It will provide the API for such query structure to return proper pixel data based on the needed/requested parameters. These parameters will be time

Figure 3.2: Class diagram of data processing part of the application

- Pixel X coordinate – 16 bit signed integer
- Pixel Y coordinate – 16 bit signed integer
- Time of Arrival – 64 bit floating point number
- Time over Threshold – 32 bit signed integer
- Fast Time of Arrival – 32 bit unsigned integer
- Slow Time of Arrival – 32 bit unsigned integer
- Trigger Number – 64 bit unsigned integer

The database is realized in the form of ROOT files and its tree structure.
3. Analysis and design

specifications (start time and length of the required interval, from which to retrieve the pixel data) or trigger number.

The server will also expose the API for manipulation with files, either the ASCII files with raw data or the processed ROOT file.

The most important paths of the API are summarized in the following list.

• GET /data/interval/{start_time}/{interval_period}
  This is the most important, and will be the most used path of the API as it provides the main functionality – it retrieves the pixel data from within the specified time interval from the ROOT database and sends them as a JSON response.

• GET /data/trigger/{trigger_number}
  Retrieves all the pixel data detected by the sensor within one specified trigger period. The response is again a JSON encoded collection of pixels.

• GET/rootfiles
  Retrieves the list of all the ROOT files stored on the server.

• GET/asciifiles
  Lists all the raw data ASCII files stored on the server.

• POST/asciifiles
  Path for uploading a new raw data ASCII file to be processed.

A full documentation and specification of the API is included in the enclosed CD (see appendix B).

3.7 UI & UX design

3.7.1 Target user group

To be able to start designing any user interface, it is necessary to understand, who are the users of the program. The largest target user group in this case are physicists (particle physics, solid state physics or nuclear physics) and electrical engineers with understanding of the displayed data. Some of them will also be experts working directly with used detectors, so they will have detailed knowledge about the data, its flow, structure, possible problems and all its other properties. They could be seeking more specific data, demand more precise display or otherwise customize the visualization.
3.7. UI & UX design

3.7.2 Usability

To maximize usability, the general UI layout and the placement of the main elements have to reflect (and support) how the program will be used. In this case, the 2D visualization (square histograms) of the sensor layers are the most important information to be presented by the program and will be the most viewed. To see the particle hits on the sensor, their placement, shapes and intensity is what users come for. It should take the dominant place in the layout so that it catches the user’s eye immediately upon visiting the page. A 1D visualization (timeline) also plays very vital role in the program and is also tied together with the tools, which provide most of the important active user interactions (selecting time interval size, querying for next or previous interval, zooming specific time range, etc.). Therefore, it should be also placed in an appropriate (always visible, if possible) position and its size should reflect its importance.

There is other functionality that needs to be present on the site, such as displaying statistics, various settings, filter controls and more. Although, this mostly falls into the category of features used not very often (at least not always) and/or used by narrower group of (more advanced) users. According to the pareto principle – the 80/20 rule, the biggest priority and focus should be assigned to the most important and most-used aspects of the website [21], while the rest of the functionality should not clutter the user’s viewport and can occupy less dominant position on the screen.

3.7.3 UI layout & wireframing

Based on the aforementioned principles and general ideas about the UI, the wireframe for the application was designed in the form as displayed in figure 3.3. There, it is possible to see that the 2D visualization of the sensors and the timeline are placed to occupy the whole width of the screen. This corresponds with its importance and also provides a much better clarity in the displayed data. Taking the nature of the visualization into consideration (on the squares, the detector’s whole 256x256 matrix of pixels will be displayed), the bigger the squares, the better the readability of the data. The smallest single recognizable element on the screen will be the size of the width of the square, divided by 256. This is also one of the deciding factors why this variation of UI layout was chosen over one with a settings panel on the side of the screen, with a more direct access to the controls.

The placement of the main menu at the top of the screen and the control section at the bottom was a direct consequence of the previously placed elements. The menu is also in a common place, where the users expect it and would look for it, resulting in a more efficient usage.

Very important in the UI design was to provide users with control tools that are intuitive to use and that the users don’t need to learn how to use
3. Analysis and design

Figure 3.3: Wireframe of the application

them. The controls should feel familiar, comfortable, and the user should trust the system. This is why the interval size input is designed as a standard form input, the tabs on the bottom of the page are standardized (the active tab will be highlighted) and for next/previous interval and animate functions, clear pictograms, as used in countless other applications, are used.

3.7.4 Responsivity

This application is designed to be used in a modern environment, allowing physicist to use it efficiently on various devices and be able to examine the sensor data on-the-go. To provide support for different screen sizes is one of the now almost standard requirements for web application development. Responsivity brings many challenges due to the nature of this application (preserve the accuracy of the visualization and all the functionality), but concerning the layout, it is more straight-forward. In figure 3.4 a wireframe, how the application is designed in case of smaller screen size, is shown.

It is important to note, that the main visualization element, the square with the sensor data, is kept wide enough, even for the price of re-arranging the layout of the page, thus preserving the readability of the data and the
3.8. Front-end application

application’s main purpose. As mentioned before, the most vital controls for user interaction should always be visible. Therefore, the timeline and the buttons are still placed on the top, spanning across the width of the screen. Also, the size of the controls is important for easy access. They have to be large enough to be easily accessible on a small screen by finger tapping. The menu is re-arranged, so that there are always all the options available on the screen and no overflow can happen.

3.7.5 Colors and graphic design

Moving further from general layout design and wireframing, more detailed features of the UI have to be designed. The graphical design and the overall feel of the page is an important, yet often neglected aspect, especially in projects and tools of such professional and formal nature. It is beneficial to provide users with convenient environment and strive to make them feel comfortable using it. However, the character of the product has to be taken into consideration, and the design should support it, not break it or try to provoke a different impression.

It is necessary to identify the constraints which the design has to consider. In this case, the visualization will be in greyscale in its default state, so there should be black and white in the color palette. The overall feel of the application should remain true to its nature, so it should still be conservative and simple. Therefore, very mellow variation of monochromatic color palette, build around light blue color was used as can be seen in figure 3.5. Blue and grey colors were chosen for the reason mentioned above – blue color is associated with intelligence and stability and due to its ample presence in the real world (sky, sea), seeing the color causes the body to produce chemicals that are calming [22] and people tend to be more productive and focused on their task. Grey color serves well as a non-intrusive supplementary color to connect the black, white and the two shades of blue.

For the design of specific features and elements on the page, it is important for them to feel familiar, clear and simple. Therefore, in this project it is suited to use (where it will be possible) the Twitter Bootstrap framework, as it will not only provide simplistic, well-known assets for look of the elements such as form inputs, tabs and menus, but will also help in layouting, responsivity and application development in general.

3.8 Front-end application

The front-end data visualization part of the application will be responsible for retrieving the data and display it in the correct form. TypeScript allows structuring the code clearly into classes, each providing the functionality of

http://getbootstrap.com/
Figure 3.4: Wireframe of the application – mobile version
one specific part of the visualization. There will be the `DataAccess` class, which will communicate with the server, manipulate and provide the pixel data for other classes. The main visualization elements, the Timeline (1D visualization) and the Sensor Histograms (2D visualization) will have their own classes with all the functionality connected to them. Due to their common properties, they will inherit from the abstract class `CanvasVisualization`. Structure of the application will be described in the implementation section [5.1 on page 37](#).
Data processing – backend

With a clear definition of how the application should work and look, a more detailed description of how all the parts were implemented in practice will follow. In this chapter, the focus is on the implementation of the back-end part of the application, the data (pre-)processing.

4.1 Raw data conversion

The first part of the application that needs to be addressed is the conversion of raw data, produced by the detectors to the processable format. As described before, the data are stored in the ASCII format in a single large file with each line representing one event occurring on the sensor. The class AsciiToRootConverter is the main responsible class for all this initial data manipulation and conversion. Initially, it loads the data from the file using C++’s fstream class and starts processing them.

There are several problems with the data. First of all, each event (data on each line) consists of raw values received by the sensor and not the real time values which will be needed later on in the application. For visualization purposes, the real ToA of each pixel is needed – in the data there are only ToA and fToA values and it is therefore necessary to calculate it from those. This is possible to do with understanding, what these values mean. As described in the Timepix3 chapter [1] page 5, ToA is latched into the register in moment of rising edge of global clock, which has 40 MHz frequency, and fToA is the number of rising edges of the fast clock with frequency 640 MHz [11]. Periods of those clocks are thus 25 ns and 1.5625 ns respectively (1/frequency). It is therefore possible to calculate the resulting real ToA as $25 \cdot \text{ToA} + 1.5625 \cdot \text{fToA}$, in nanoseconds.

Also, the pixel’s coordinates on the sensor matrix need to be calculated, using the event’s MatrixIndex attribute. The X and Y coordinated are encoded in this single number in such a way, that $X = \text{MatrixIndex}/256$ and $Y =$
4. Data processing – backend

MatrixIndex%256.

These conversions take place in the constructor of the SinglePixel class, which represents the pixel hit on the sensor with all its real, processed values. Its constructor takes either directly values, or an instance of SingleEvent as parameter. SingleEvent is the representation of one line of the raw data written in the file by the sensor. The AsciiToRootConverter reads each line as a SingleEvent object (GetOneEvent method) and creates a new SinglePixel with this SingleEvent as a parameter.

4.1.1 Data sorting

As the visualization requires querying of the data based on time specifications, the data have to be in a form that allows to quickly decide which pixels are within a specified time interval and which are not. This is difficult with the current data storage in ASCII files. The detector works with buffering and often data are not written to the file in chronological order. Luckily, it is not needed to do all the procedures and preparation of the data in real time and it is possible to sacrifice some computational time and pre-process the data in the most efficient way. In this case, it means sorting all the data chronologically, based on their real ToA. Unfortunately, due to the trigger-based nature of the data, it is not possible to sort all the data in a simple way.

4.1.2 Trigger-based approach

As explained earlier, once in a while, the data readout of the chip is reset (trigger is registered) and all the values stored in the file start again from 0. This trigger is marked by a line of zeroes in the data, a so-called “empty event”. Therefore, the class SingleEvent has a method EmptyEvent() and based on this, it is possible to recognize the trigger.

Because of this, the AsciiToRootConverter has to process the data trigger-by-trigger and somehow compensate the resets and re-calculate the cumulative ToA for each pixel. The main loop of the converting process is (simplified) outlined in the following code snippet:

```cpp
do
{
    // read one period between
    // the "trigger lines" from ASCII file
    if (!this->GetOneTriggerPeriod(prevPixVector, ...))
    {
        end = true;
    }
    // sort pixels within one trigger period
    std::sort(prevPixVector->begin(), prevPixVector->end());
```
4.1. Raw data conversion

    triggerNo++;  // write sorted data to ROOT file
    // ...
} while (!end);

bool AsciiToRootConverter::GetOneTriggerPeriod(...) {  
    while (getline(file, line)) {  
        c_event = this->GetOneEvent(line);  

        if (c_event.EmptyEvent()) {  
            break;  
        }

        c_pixel = new SinglePixel(c_event, triggerNo);  
        pixVector->push_back(*c_pixel);  
    }  
    return true;  
}

To provide the sorting functionality, AsciiToRootConverter pushes each read SinglePixel into C++’s std::vector and with the overloaded operator < in the SinglePixel class, it uses std::sort on this vector.

4.1.3 Error detection & repair

There are several more problems with the data. E.g. there are cases, when the data are not written to the file in the correct order (already solved as described), but it can also happen, that these buffered data overflow into next trigger period, which means the sorting would put them in completely wrong place. AsciiToRootConverter has to therefore detect such an overflow error and repair it (put the data into correct trigger). It is possible to detect such overflowed data by their ToA values. Because the buffering and wrong ordering of the events is happening only within a couple of lines after the “empty event”, the overflowed pixels are always in the beginning of the trigger and have large ToA value (they originated from the end of previous trigger). So it is enough to scan only the first few lines of the trigger for these overflowed values and calculate their difference with the value of the last ToA of the previous trigger period. Precise values for how far it is necessary to scan and how big the difference must be, have to be determined empirically and are set by the overflowSearchRange and overflowTreshold attributes, which can easily be modified.
There are two approaches, how to detect and repair (put in the correct trigger period) the overflowed values.

- **Forward detection:**
  While processing the current trigger, the program looks ahead to next following trigger period, detects the overflowed values and immediately adds them into current vector of pixels, before sorting them.

- **Backwards detection:**
  While processing the trigger, the program detects if some of the values are overflowed and should have been added already to previous trigger. It then takes these values and pushes them to previous trigger’s vector and (re)sorts them.

Both of these methods have their advantages and it was experimentally determined, which is more efficient. As shown in figure 4.1 and figure 4.2, backwards detection showed better performance, especially with large data instances and therefore was chosen.

In the end, the program works in such a way that it keeps two vectors, one for values from the previous trigger, one for values from the current one. While processing data from one trigger, it pushes all overflowed values to the previous trigger’s vector and the rest to current one’s. Upon completing the trigger period, the program sorts the previous trigger’s vector and saves the data from it into the file. It also marks the last (largest) ToA value from the previous trigger for later use in overflow detection and indexing (will be
described later). Then it swaps the current and previous vector and continues the same way to next period.

### 4.2 Data storage & database

After processing the data into correct format, the next important step is to save them to a database for later use and fast retrieval. For this project’s database storage, the TFile and TTree classes of the ROOT framework were chosen to provide this functionality.

#### 4.2.1 ROOT files

Within the ROOT framework, the ROOT file class (TFile) is responsible for saving any objects (inherited from TObject) into a binary compressed format and taking care of efficient storage and input/output operations with the file. The file works similar to the Unix file system and can thus contain objects, as well as whole directories. It also takes care of job crash protection, so that no data are lost while working with the file.

In the constructor of the TFile class, it is possible to specify the name of the file, the mode in which the file will be opened (reading, writing) and the level of compression.
4. Data processing – backend

### Table 4.1: Compression factors

<table>
<thead>
<tr>
<th>Compression level</th>
<th>1</th>
<th>5</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of original [KB]</td>
<td>4173</td>
<td>3025236</td>
<td>4173</td>
</tr>
<tr>
<td>Processing time [s]</td>
<td>11.56</td>
<td>12.59</td>
<td>18.28</td>
</tr>
<tr>
<td>ROOT file size [KB]</td>
<td>1298</td>
<td>1267</td>
<td>1242</td>
</tr>
<tr>
<td>Time of loading [s]</td>
<td>0.07</td>
<td>0.078</td>
<td>0.094</td>
</tr>
<tr>
<td>Compression*speed</td>
<td>90</td>
<td>10614</td>
<td>9070</td>
</tr>
</tbody>
</table>

4.2.2 Data compression

There are 10 different compression level settings possible in the ROOT file creation. They are defined as follows:

- 0 – objects written to this file will not be compressed.
- 1 – minimal compression level but fast.
- ...
- 9 – maximal compression level but slower and might use more memory.

For selecting the ideal level of compression, it is needed to identify the priorities within this work. The most important quality to look at is definitely fast retrieval of the data, because that will be the bottleneck of the whole application. The application server retrieves the data from the ROOT file, transforms them to JSON format, and sends them to the client. The client will therefore be waiting for the response during the time all this is happening. Thus, the optimal speed vs compression ratio should be not all the way, but at least partially in favour of speed.

Table 4.1 shows the comparison of converting and loading times for different compression levels. The critical loading time as well as the time of processing obviously increase and the file size decreases with higher compression level. The percentage of increase in processing time is however much larger than the percentage of improvement in file size. The most important value – the loading time – is increasing and with such small compression improvement, it is not advantageous to choose high compression factor.

4.2.3 Tree structure

For storing the data into the ROOT files, the data will utilize a tree structure provided by ROOT’s TTree class. This class consists of individual branches (TBranch) with their own definition, buffers and with the ability to store any simple structures, variables, objects inherited by TObject or ClonesArray (specialized list of objects). In this case, individual attributes of the SinglePixel object are stored each in its own branch.
4.2. Data storage & database

Branches of the tree have to be initialized by the Branch() method with the branch name, the address of the first item to be stored and the concatenation of the variable name and type as parameters. The following code shows the initialization of a branch with the name “b_index”, storing the variable “index” of type 64 bit unsigned integer [23] and binds the address of the “v_index” variable to the branch.

```cpp
TData->Branch("b_index", &v_index, "index/l");
```

Upon calling the Fill() method on a TTree class, all values of variables on the addresses bound to the branches are saved into the branch as the next entry. In this case, the current value of the “v_index” variable is added to the branch.

After each fully processed trigger period of data (as described above) the data from the vector are consecutively written to the bound addresses and pushed into tree branches. The whole tree is then saved into the file at the end of whole conversion process.

4.2.4 ROOT database

In the end, pixels are consecutively saved into the tree as separate entries with each of its attributes in its own branch and sorted by their real ToA. First they are sorted by their trigger number (the TriggerNo value) and within each trigger they are sorted by the ToA value. ROOT allows for fast retrieval of specified entry (by its index) and also reading only some specified branches. But because it is required to query and retrieve data based on their real time, some method how to find the correct data has to be introduced.

4.2.5 Indexing

Indexing of the database is advantageous as it serves for faster searching through such big amounts of data stored this way. It tells the program where exactly in the database storage it is supposed to look (at which row) to get to the required data as quickly as possible. Creating of the indexes takes place at the same time as the data about pixels are saved into the trees – after each processed trigger period. Indexes are stored in a separate tree in the same ROOT file and they have following values:

- Trigger Number:
  
  Number of currently processed trigger period. This value is not actually physically saved into the tree, as each next line of the tree marks each next trigger number and the numbers are the same, so entry/row number can be used for this purpose.
4. Data processing – backend

Figure 4.3: Scheme of the ROOT database storage with indexing

- **Start Time:**
  Value of cumulative ToA – it is the sum of all previous triggers last pixel’s ToA value. This way the number represents an estimate of the real amount of time that passed during all previous triggers, therefore value at which the measurement of the current trigger period started. The estimation is based on the assumption that there is 0 time between last pixel hit of the previous trigger period and the time of next trigger happening.

- **Start Row:**
  It is the number of all pixel hit entries stored into the tree before this trigger. In practice, it marks the index (row) of the database, where the entries belonging to this trigger period start.

In figure 4.3, the final resulting scheme of how the data are physically stored within the ROOT file is shown. It is possible to see both the above described indexing tree and the data tree, which are stored in one file side by side. Each row of indexing tree represents one trigger period and its StartRow value points to a specific row (entry) in the data tree, where the corresponding trigger period starts. It can be read from the figure, that for example data from trigger 2 (entry number 2 in Index Tree) start on row 13 in the Data Tree and the measurement of this trigger started at time 1350 (calculated as 150 + 1200, sum of last ToA values of previous triggers).

All other values stored in the Data Tree, apart from TriggerNo and ToA, are omitted from the schema for simplification.
4.3 Processed data retrieval

With the data correctly saved in the tree structure in the ROOT file, they are ready to be read by the rest of the program, transformed into JSON format and sent to client. It should also be possible to efficiently retrieve only the data, that meet specified criteria (ToA limits).

4.3.1 Fast interval retrieval

While retrieving data, it is possible to quickly search through the database using the indexes. The class RootDataFacade provides an interface for working with, and retrieving the data. It is the main responsible class for manipulating with the database. Its method GetInterval() is an integral part of the whole application – it takes two parameters, startTime and timeSlice and returns data, which were measured by the sensor within an interval of size timeSlice, beginning at time startTime.

With the data stored the above defined way, the retrieval function has to go through several vital steps:

- Find the closest trigger, where to start searching,
- find the index of the first pixel, meeting the time criteria,
- find the trigger, where to start searching for the last pixel,
- find the index of the last pixel meeting the time criteria,
- return all data from the database between start pixel and end pixel.

For a fast searching of the closest trigger to the specified time, the binary search is utilized. The method FindClosestTrigger() takes the time parameter and searches for this time in the StartTime values in the Index Tree (as seen in figure 4.3). It starts in half of the tree, compares its value on the current row with the specified parameter, and, if the time doesn't belong to this trigger period, performs the same search recursively in the first, or second half of the tree, depending on the value being greater or lower.

After the correct trigger number was found, the program can start searching for the specific pixel, with which the resulting pixel interval should start. For this purpose, there is the method GetIndex(), which takes two parameters. The first one is the trigger number that specifies, within which trigger the search for the pixel should be performed. Thanks to the indexing, this method simply looks at a row with this number in the Index Tree and takes the StartRow value. This value points to a specific row in the Data Tree, where the data from this trigger start. The method starts there and sequentially searches for the value of its second parameter – the time. When it finds the first pixel with a ToA larger than this time, it returns its index (current row number).
This index marks the start of the interval of pixels, which will be returned by the `GetInterval()` method. To find the ending pixel's index, the same procedure is repeated (finding trigger number by binary search and finding correct pixel index). With starting and ending indexes, the program reads all the pixel data sequentially from the Data Tree and pushes them into an instance of the `PixelInterval` class.

`PixelInterval` represents a sorted collection of pixels (SinglePixel class) and it is a wrapper (decorator) over the standard C++ `std::vector` class. It inherits most of its methods (`PushBack`, `PopBack`, `Empty`, `Size` etc.) and enriches it with some additional functionality, more specific to this task. It adds `TriggerCount()`, which returns the number of triggers which the selected data are taken from, `SumEnergy()`, which returns summed ToT value of all contained pixels, and `Print()` which is an interactive print method, that replicates how the `Browse()` method of ROOT’s `TTree` class works, to provide consistent browsing of the data throughout the application. It lists some amount of first lines and waits for user’s input to show more or to quit.

### 4.3.2 Data serialization

With the proper data returned in the form of the `PixelInterval` object and with the application requirement for sending those data from the server to the client part of the application (written in JavaScript), a method how to serialize such data for easy sending and processing has to be introduced. The JSON (JavaScript Object Notation) data-interchange format is used. It is very popular, widely used and very well suited format for such purposes.

The `PixelInterval` class provides a method `JSONify()`, which transforms all the pixel data to JSON format and returns them as a string. For generating the JSON string, it uses RapidJSON and its Writer class, which allows an easy creation of JSON objects by using methods such as `StartObject()`, `Key()`, `Int()`, and `StartArray()`. It writes the JSON into a `StringBuffer`, which can then be used by the `JSONify()` method to return a string. This way, all the data are complete, in correct format, and ready to be sent to the front-end part of the application.

---

This chapter describes how the data, received by the server, are processed and visualized at the front-end, the client side of the application. It will go into details about the functionality of the visualization features and their implementation.

### 5.1 Application structure

The whole application follows the large-scale application development trend of reusability, modularity, and low coupling. TypeScript allows a structuring into classes and the usage of basic and very useful object-oriented programming patterns. Figure 5.1 shows a class diagram of the front-end application.

Exemplary are the class `Redrawer` and interface `IRedrawable`. The `IRedrawable` interface represents any object, which is subject to change upon changing the data. This structure utilizes the so-called Observer design pattern. It is possible to register (subscribe) any class implementing the `IRedrawable` interface to the `Redrawer`. Once the `Redrawer` receives new data, it notifies all subscribed observers (via `redrawAll()`) by calling their `redraw()` method.

Another notable structure is the `BaseObject-CanvasVisualization` hierarchy. `BaseObject` represents any object of the application with a name and with any functionality, that can result in error or warning. `CanvasVisualization` is an abstract class representing a two-layered, responsive visualization element using the HTML5 canvas element for displaying the data. Two-layered means, that it uses two separate instances of a canvas of the same size on top of each other. This can be used for various purposes: drawing background/foreground separately, drawing two different instances over each other for comparison, drawing temporary images and shapes, etc. This class is universal and contains all the common basic properties of a po-
5. Data visualization - frontend

Figure 5.1: Class diagram of front-end application

tential concrete visualization tool. For example, the method `drawNoData()`, which is used in case no data are received from the server. It displays the message “no data” in the center of the canvas and stops the rest of the class from trying to draw data or make any calculations.

5.2 Data retrieval

On the client side, to be able to work with the pixel data and use them for visualization, they first need to be retrieved from the server. The application was designed in such a way that the `DataAccess` class will be managing all the handling with the data. All communication with the server and querying for new data takes place within this class. This way, the rest of the application
can be more independent. It was also advantageous in the testing phase.

The constructor of the `DataAccess` class takes a string parameter, `serverAddress`, specifying the base URL, on which the API and the data will be available. When not specified, it defaults to localhost (the back-end application is running on local machine). At the entry point of the application (main.ts), the program loads the server address, which will be passed to the `DataAccess` constructor from a config file .config.js. Thus, it is easy to switch between local or remote location of the data and an independent distribution of the client part of the application is made possible.

### 5.2.1 Time interval querying

To get the correct new data from the server, a query according to the user specified input has to be sent. The most important and most used type of data query for the visualization is getting data from the time interval, defined by the start time and the time slice (interval size). The user can input these values in the form on the top of the visualization page and update all the data displayed this way.

For this purpose, the `DataAccess` class has a method `GetNewData()`, which takes three parameters: start time, time slice and callback. All the data querying utilizes asynchronous AJAX calls. The event-driven nature of this allows this method to notify the rest of the application, when new data are received (by calling function specified by the callback parameter), and not block the whole application for the time needed to receive the data. The AJAX call is implemented using jQuery in the following way:

```javascript
$.ajax({
    url: this._serverAddress + "/data/" + startTime + "\" + timeSlice;
}).done(function(data){
    dataObj.setData(JSON.parse(data));
    callback(dataObj.getData());
}).fail(function(err){
    dataObj.addError("There was an error contacting the server. No data received.");
    callback(null);
});
```

As a response to this call, the server sends the data encoded in a string in JSON format. Upon receiving the data, the method `setData()` is called to save this data in memory, so that the `DataAccess` object can work with them later. Also, the specified callback is immediately called with the new updated data. In case that no data were received, an error is shown and the data are not updated in the visualization.
This querying for new updated data takes place any time user specifies the new start time or time slice and presses the update button. The application queries the server also every time the user presses the next or prev button. In this case, the interval size (time slice) stays the same and only new start time is calculated to return the data of the interval directly preceding, or directly following the currently displayed one.

5.2.2 Animation

The animation function allows the user to observe the change of the sensor data in time. When the user presses the animate button, the application starts making periodical AJAX calls. It queries the server for new data every 3 seconds (more precisely, each time 3 seconds after receiving the data). It is implemented by using the setTimeout() function and recursive calls of the function. The basic principle is outlined in the following code snippet:

```javascript
function animate() {
    dataAccess.getNewData(sTime, tSlice, function(data) {
        // process the data here
        setTimeout(animate, 3000);
    });
}
```

Each subsequent call returns data from interval of the same size, starting immediately after the current interval. Therefore, it behaves in the same way, as if the next button was pressed repeatedly. The data update in all the visualization components, so that user sees the progress of data in time.

5.3 Data filtering

An important functionality of the application is to allow the user to filter data. A comprehensive set of options how to filter the data should be provided, because isolating and identifying specific events can help understanding the detector performance, the type of impacting particle or to remove unwanted signals (such as noisy pixels).

Filtering takes place in the DataAccess class. It manipulates the data which will be provided to all visualization components. All the components retrieve the data to display from calling the GetData() method on the DataAccess object. For applying the filters, DataAccess further calls its method GetFilteredData(), which goes through all currently set filters and changes the returned data accordingly.
5.3. Data filtering

5.3.1 Pixel property filtering

The first option to filter the data is by any of the stored properties. In the filters section of the visualization tool, the user is provided with form inputs for each of the pixel’s properties (ToA, ToT, TriggerNo, fToA, sToA). It is possible to set both, required minimum and maximum of these values.

After setting any of these inputs, a new filter is added to `DataAccess` object using the `addFilter()` method. This method takes three parameters: the quantity to filter on, the minimum and the maximum value. Filters are implemented as a separate class `DataFilter`. A new instance of this class is created with these specified values. This new filter is then attached to the `DataAccess` object (pushed into the filters array).

When retrieving the filtered data, `DataAccess` loops through all the currently attached filters and returns only those values, which meet the criteria specified by each individual `DataFilter` instance in the array.

5.3.2 Time filtering

It is also possible to filter the data based on a time specification. Not only by the start time and time slice user inputs, as described above (server side), but also within the already received interval of pixels. It is possible to limit these data based on a time range selected in the timeline component of the visualization. This filtering takes place on client side. It limits the data to only include those within a smaller interval, defined by starting pixel index and ending pixel index. It is described in section 5.4.6 on page 45 how these values are sent from the timeline.

5.3.3 Custom user filtering

An option to create arbitrary filters defined by the user was implemented. It is possible to fill the “own filter” form input at the filters section and specify basic expression that will be evaluated on each pixel’s values to decide whether to include the pixel in the filtered data. The expression can consist of variables representing all the pixel attributes and simple mathematical expressions – addition, division, modulo, bracketing, etc.

Evaluating user input expressions is not trivial. JavaScript offers a function `eval()` that evaluates and executes the string parameter. It has to be used with caution, as it executes the user specified code with the application’s privileges and, thus, is susceptible to attacks, misuse and associated damage. It is also relatively slow, since it has to invoke the JavaScript interpreter. Although, in this particular situation, it is still the most suited and most efficient solution. The possible risks were minimized by allowing the execution only in definitively safe conditions.

The user defined filter is added to the `DataAccess` class by the `ownFilter()` method. It first splits the string to individual terms (split by space) and then
compares each of them with a pre-specified array of allowed terms. If the user input is neither an entry of the array nor a number, the string is not evaluated and an error message is displayed.

The array of allowed usable terms is stored as a class attribute of the DataAccess class. It is also used by jQueryUI\(^\text{12}\) Autocomplete, which helps the user with inputting the correct expression. It suggests all possible options as the user writes. The autocomplete function binds user interaction (in this case pressing any key – event “keydown”) to a function that displays the suggested items. Those are specified in the “source” parameter, which in this case uses the DataAccess’ method `getAllowedFilterTags()`.

The user’s selection of any of items is bound to a function. This function splits all what is currently written in the form input by space (to separate individual items), then removes the last item from the array created this way, as it can be a not completed one (happens when user selected suggested item mid-typing). After that, the function pushes the selected item into the array instead and writes the whole array back into the input with a space between each of the item.

### 5.4 1D Visualization

The first major visualization element is the timeline. It displays the data within the interval in time and provides some tools for data control and display.

#### 5.4.1 Binning

Due to the nature of data, as a continuous stream of pixel hits, and due to its volume, it is not possible to display them individually, so that some aggregation has to be made. The timeline is implemented as a bin chart, where each bin marks a certain, short period of time, and its content resembles the number of pixel hits within this period.

Data are retrieved in a chronological order. For the proper bin assignment, these data are looped through from the lower bin edge to the upper bin edge. When the time value exceeds the upper bin edge, the program moves to the next bin and updates the proper bin edges (adds bin size value). The Timeline class keeps the attribute `_data`, which is an array of the number of pixel hits in each bin.

It is possible for the user to set the size of the bins. Whenever a value in the bin size form input changes, the value is changed in the Timeline class, the bins (_data array) are recalculated, and the timeline is redrawn. The method `setBinSize()` first converts the user specified value from seconds to nanoseconds (which is the correct unit of time in the data) and checks

\(^{12}\text{https://jqueryui.com/}\)
this newly input value to make sure it is not too large or too small – it does not allow inputting of values resulting in bin sizes of less than 1px (it is calculated using \texttt{canvasWidth} attribute of the \texttt{Timeline}), or resulting in less than 10 displayed bins.

5.4.2 Trigger detection & display

A problem with the binning approach described above arises from the trigger-based nature of the data. Since the time is reset on a received trigger signal, once the time is reset the bins would be filled a second time, which is not intended. Therefore, inside the loop, every time the current pixel’s TriggerNo is greater than the previous, the bin starting time is set to 0, the bin ending time set to bin size and the binning continues from the start. This could lead to inaccuracies in the displayed timeline, since the trigger can happen in the middle of the “bin period”, and the bin is divided into two. This is due to the nature of the data and cannot be avoided. When a trigger appears in the data, there are no information about how much time passed between the last pixel detected and the trigger. For example, at the end of the trigger period, there could be several actual bins, which should be empty, before next trigger is displayed. For this reason, each time the data consist of pixel hits from multiple trigger periods, a warning is displayed that data may be inaccurate, and a marker is drawn onto the timeline, where the trigger happened.

5.4.3 Data drawing

The method \texttt{drawData()} is responsible for the actual drawing of the bins onto the timeline. It uses the _data array, storing the bin sizes and triggers, as a source. The timeline is divided as many times as is size of this array (which equals to total number of bins) and on each of this divided parts, a rectangle, representing the bin, is drawn. The rectangle has a width equal to the width of the timeline canvas (\texttt{canvasWidth}) divided by number of bins. The height of each bin rectangle is calculated as a value between 0 and the \texttt{canvasHeight} attribute, according to the proportion of the bin value to the maximum value of all bins.

By this, the timeline stays fully responsive and the bins are always displayed in proportion to the timeline width and timeline height.

The \texttt{drawData()} method is also used for drawing the trigger marks wherever needed. This is implemented in a way, that every time the trigger is detected in the data, a negative value of the current bin size is saved into the _data array instead of the normal one. While drawing the data, whenever a negative value is found, a thin red line is drawn at the upper edge of the bin and the current trigger number is written on the canvas.
5.4.4 Timeline scale

Apart from drawing the histogram, the axes labels of the chart have to be displayed. Therefore a separate canvas element with the same width (\(\text{canvasWidth}\)) as the canvas is created and placed directly below the timeline. Short lines are drawn at the bin edges labelling the regions by their corresponding time values. The lines are always drawn at the edge of the bins, so that user can see the time scale in relation to the bin size. The program automatically decides, where to draw these regions. For a good readability, it doesn’t allow the region to be smaller than \(1/20\) of the timeline width. Thus, any time the bin size is smaller than this limit, one region contains more than one bin (the minimum number needed to exceed the limit).

5.4.5 Timeline interactivity

The timeline also features interactive elements. Not only the bin size is changeable by the user and next/prev/animate buttons can be pressed, but there are also two handles, one on each side of the timeline initially, that allow the user to more precisely select the desired time interval to be displayed. It is possible to drag and drop them anywhere on the timeline and it causes update of the displayed data in the rest of the visualization – the 2D histograms and statistics – to only show those pixels, that were detected by the sensor within this narrower timeframe.

These handles are implemented as a regular HTML div elements, placed on top of the timeline. Their background is an image, that clearly evokes the “drag” functionality. For providing this functionality, jQueryUI Draggable\(^{13}\) is used. This, added to an element, allows the element to be dragged across the screen. It offers multiple parameters for further customization. It is possible to set an “axis” parameter to limit the movement only to the horizontal direction and a “containment” parameter to limit the movement within the specified parent container element. However, the most important parameters of the draggable element are “drag” and “stop”, which allow to bind an event callback function, which is called each time the element is moved, or when the movement is stopped, respectively.

This is done in Timeline’s method setupHandles(), which takes care of assigning the draggable functionality to correct elements, sets up the parameters and registers the methods rightHandleStop() and leftHandleStop() as a callback functions for the “stop” event. Each of these methods calculates the handle’s current position in the timeline by using its “left” CSS attribute. It strips this attribute of its units (“px”) and converts it to a number. With this number, the calculation to determine on which bin is the handle currently placed starts. It firstly compensates for the handle’s width, to get where the center of the handle is, and then divides the given number by the width of

\(^{13}\)https://jqueryui.com/draggable/
the bin (i.e. the width of the canvas divided by number of bins) to get the correct bin number. With known current bin number, the update of the data can be performed.

The timeline also provides the “snap to non-zero values” functionality, which searches for the closest non-empty bin, when the handle is placed on the empty one. In case this option is selected, upon placing the handle, the program loops through the bins stored in the _data array, and returns an index of the first element with a value greater than zero. From this number (bin number) the correct new real position of the handle has to be re-calculated (bin width times the found bin number) and the “left” css attribute of the handle is set, to correctly move the handle over the bin.

5.4.6 Data updating

All the updating takes place in the updateData() method of the Timeline, which is called in each of the handleStop() methods. The first part of updating the data are the labels below the timeline, displaying how many pixels are within the specified interval, what is the interval real start time and what is the end time. Because the timeline allows dragging the handles freely with no restrictions on which one is left and which is right, the minimum value of those two is always taken as the starting point and the maximum value as ending point. Then, all the pixels in the bins between are counted and displayed in the “Pixel Count” label. The start and end times of the interval are calculated as the bin size times the respective bin number.

The most important data updating functionality is updating of the data displayed in all other visualization elements according to the selected interval. This is achieved by associating the DataAccess class, which is responsible for the general data updating, with the Timeline. The Timeline keeps a reference to the DataAccess object as a class attribute. In case of moving the handles and specifying the time interval, it calls its setInterval() method. This method takes two number parameters representing the starting and ending index of the pixel in the pixel data array. These values are saved in the binning stage (5.4.1 on page 42) into the _pixelIndexes array. A number, representing the index of the first pixel contained in the bin, is assigned to each bin in the timeline. The updateData() method then has to only pass those indexes for the currently selected bins to the setInterval() method and the data are correctly updated.

However, in order to see the results of updating the data, it needs to be displayed on the screen in the visualization components. Another class is responsible for the (re)drawing, the Redrawer. Therefore, the Timeline also keeps a reference to the Redrawer object and after successfully updating the data in the DataAccess, it calls its redrawAll() method, which notifies all its subscribers and calls their own redraw() method. This way, the new data are displayed on every visualization element on the screen.
5. Data visualization - frontend

5.4.7 Filtered data display

Another functionality of the timeline is the possibility to display two different sets of data on top of each other. When the user specifies any filtering of the data in the “Filters” section of the visualization (on the bottom of the page), either by setting the values for data attributes or writing custom expression, data are filtered in the DataAccess class. The timeline can display this data in a different color, for comparison of the amounts of pixels before and after applying the filter. This is demonstrated in figure 5.2.

The filtered data display utilizes the two-layered character of the CanvasVisualization, which allows the separate drawing and deleting of graphics in a second layer, without affecting the original one. The Timeline therefore keeps the separate array for the filtered data (_filteredData) and draws it on a complete separate (transparent) canvas, which is placed directly on top of the unfiltered one. For doing this, it has separate redrawFiltered() method, that takes the new data as a parameter. Getting the filtered data into correct bins is easier than doing it with all the data in the first place, because the number and size of the bins are already known. It loops through the bins, fills them with data from _filteredData array, and checks for triggers (negative values) to start binning from zero in each new trigger period. The drawing is done in the same way as for the non-filtered data (5.4.3, page 43), only without the necessity to redraw the “trigger lines”.

5.5 2D Visualization

The 2D histograms of the sensor data are the most important and key feature of the whole visualization tool. It shows the data from the detectors, visualizes how, and where exactly they were detected on the sensor’s 256x256 matrix of pixels. It shows their various attributes using scaling color values to help the visual recognition of the data. It does not only display the data in clear way in colors, but also provides the exact pixel values, giving the possibility to examine them in detail.

A screen capture of the 2D histograms is shown in figure 5.3.

The aspect ratio of the histogram is fixed to keep it in the form of a square pixel matrix of 256x256 equally sized points, independent of screen
5.5. 2D Visualization

A color scale is used to indicate the actual pixel values on each of the individual points.

The abstract Sensor2DHistogram class is responsible for the majority of the functionality of the 2D visualization. This class extends the CanvasVisualization class and implements the IRedrawable interface. There are three different sensor histograms displayed on the screen. Each of them is represented by its own concrete class, which extends the Sensor2DHistogram abstract class, and defines the functionality specific for this individual histogram. The three currently used histograms in the visualization are:

- Histogram with ToA values:
  This histogram shows the ToA value of the last pixel hit on a certain coordinate.
  It thus indicates the relative time of recorded pixel hits with respect to a trigger signal and with respect to each other. This histogram also stores the raw fToA and sToA attributes of each of the displayed pixel hits.

- Histogram with ToT values:
  This histogram shows the summed value of the ToT value of all the pixel hits on a certain coordinate.

- Histogram with number of pixel hits:
  This histogram shows how many times a pixel on a certain coordinate was hit during the selected time period. Because the previous (ToT) value is summed over all the pixel hits, this histogram helps to determine how many hits contributed to the displayed energy deposition.

Due to the way the program and the classes are structured, it is easy to add further histograms with other quantities of interest. This is possible by
adding another class extending the Sensor2DHistogram and implementing
the needed abstract methods. Most notable of them is ProcessPixel(), which
takes a pixel hit with all its attributes, performs any potential calculations on
them and saves the value to be displayed. The parent class takes care of all
the actual data displaying, proper scaling etc.

5.5.1 Data management

The most important aspect of the visualization is getting the correct data and
properly displaying them on the canvas. For this purpose, the Sensor2DHist-
ogram class has a method getCanvasData(), which takes the pixel data (re-
ceived from server/DataAccess class) as a parameter, processes them and
saves the specific values that need to be displayed on the canvas on each
of its coordinate. It keeps a private attribute data for saving those values.
It is a three-dimensional array with width and height equal to the number
of pixels, and with a third dimension, used for saving various values of the
pixels. When there is only one value to be stored and displayed (in ToT and
#hits cases), the size of the dimension is 1, so the array is a regular 2D array.
However, when it is needed to store additional values (such as the fToA and
the sToA in the ToA case), they can be saved at the same position in the array’s
third dimension.

This method firstly initializes this array to proper size (initially 256x256,
but can be changed later by zooming) and puts zeroes on every position. Then,
it loops through all the received pixel data and performs processPixel() method
on each of the pixels. This is an abstract method with different
concrete implementations in each of the child classes. This implementation
determines, which exact values will be saved into the _data array. In any
case, it saves the value in the array at the coordinate specified by the PixX and
PixY attributes of each pixel. Therefore the array reflects the sensor’s matrix
of pixels.

By this, the class keeps in memory which specific value should be used
for visualization on every coordinate of the displayed square and does not
need to work with all the pixel data.

5.5.2 Data display

When drawing the data on the screen, the drawAllPixels() method is
used. Because of the responsiveness of the whole visualization, and be-
cause the method will need the real width and height of the canvas, firstly
updateSize() is called, which checks for the real current element size and
saves the values into the _canvasWidth and _canvasHeight class attributes.
The method then loops through the data saved in the _data 2D array (starting
on the _startX, _startY position, which is by default 0) and calls two methods
on each element:
5.5. 2D Visualization

- getPixelColor()

This is an abstract method implemented individually in each of the sub-classes. It receives a value as a parameter and returns a color, in which this value should be displayed in the visualization. The color is determined by the currently selected scale, by minimum and maximum values in the data and by any other possible calculations.

- drawSquare()

This is the method that finally puts the data on the screen by drawing a square of the previously selected color on the appropriate coordinate on the canvas. It uses a “point” as the unit of measurement, because of the variable real size of the canvas (changing with screen size) and the variable number of displayed pixels. One “point” is defined as the canvas real width/height in px, divided by number of pixels to be displayed (default value is, in case of whole not zoomed matrix, 256). This way the method every time converts the pixel’s coordinate in the matrix to real coordinate on the canvas element and draws each pixel in the appropriate size.

For optimization purposes, when the data are drawn on the canvas, the method drawAllPixels() firstly gets the proper color for 0 value and draws a square of this color across the whole canvas. When drawing the individual pixels, all with value 0 are omitted. By this, a significant speedup of the whole process is achieved, because

- the data are sparse and contain a very large percentage of 0 values,

- changing the state of the context (e.g. starting and finishing drawing something on the canvas) is the most time-consuming operation while working with the canvas element. Drawing one large square is many times faster than drawing many individual small squares.

5.5.3 Interactivity

The visualization is not only a static display. The histograms provide the users with the possibility to interact with them. The most important interactive feature is displaying the exact values of the selected pixel. The selection of the pixel is realized by hovering with the cursor over it and the values are displayed in the labels below the histogram square.

For registering the user’s mouse movement over the visualization, event callbacks are set up. This is done in the setupHandlers() function using jQuery events. It binds a method histogramMouseMove() to a “mousemove” event of the canvas jQuery element. Therefore, it is called every time the mouse is moved over the canvas. In this method, the current coordinates
of the mouse cursor are calculated. Since jQuery passes the Event Object to the callback function, its `pageX` and `pageY` attributes are available. They indicate the global coordinates of the mouse cursor on the screen. To get the real coordinates on the canvas, the canvas' position on the screen has to be compensated for. As the canvas is placed in wrapper elements, the `offsetLeft` and `offsetTop` attributes of those are subtracted to get the final coordinate value.

This value is the mouse cursor’s position on the canvas in pixels. To calculate over which actual drawn pixel the cursor currently is, the coordinate has to be converted to a position in the pixels 256x256 matrix. The `canvasToSensorCoordX()` and `canvasToSensorCoordY()` methods serve this purpose – they take the canvas coordinate in pixels as a parameter and return the coordinate within the pixel matrix.

With the correct pixel known, the `updateLabels()` method is called. It updates the HTML content of elements specified in `_labelXid`, `_labelYid` and `_labelValId` attributes, to display the detailed values of the pixel. These elements are assigned to the class attributes in the `setupLabelIds()` method, and because they are different for each histogram, this method is defined as abstract and implemented in each of the concrete sub-classes.

### 5.5.4 Zooming

Another, more complex, interactive feature of the visualization is the possibility to zoom into any area on the histogram. The user can freely click and drag on the canvas to draw a temporary square to define the area to be zoomed into. After releasing the mouse button, the visualization redraws itself to only show pixels within this specified square.

To draw the zooming square and make the functionality possible in general, it is again necessary to register event callbacks, this time for pressing the mouse button (`mousedown`), releasing the button (`mouseup`) and moving the mouse (`mousemove`).

On `mousedown`, the current position of the cursor is saved into `_startMouseX` and `_startMouseY` variables, the `_zooming` boolean variable is set to true (and then is set to false on the `mouseup` event).

When the `mousemove` callback function is called, it can determine whether the user is currently zooming. If yes, the temporary zooming square is drawn. Again, the two-layered character of the visualization is utilized. Every time the mouse is moving, a semi-transparent square is drawn into the second layer of the visualization and the whole layer is completely cleared when the new square is drawn. Therefore, the effect of drawing a temporary square is created, without interfering with the already drawn histogram visualization in any way.

While drawing the zooming square, its 1:1 ratio has to be preserved; it is not possible to zoom into a non-square rectangular area. The program
preserves this ratio by calculating the difference between starting and current X and Y mouse coordinates and by taking the greater of these two as a size of the square. However, this may lead to overflowing the edges of the histogram. Thus, the zooming is clipped as shown in figure 5.4. A marks the point, where the user started the zooming (pressed mouse button), the dotted rectangle represents the dragging movement (mouse button stays pressed), and B is the ending point (mouse release). This way, the square marked by the dashed line would be drawn, but the program detects if any of its points are beyond the histogram boundaries, and draws only a smaller, clipped square, marked by the filled region. The program also checks which of the points is on the topmost and leftmost position, saves its coordinates and saves the square size by calling `updateZoomingCoordinates()`.

The coordinates saved are used to finally draw the square in the second visualization layer. The color of the square is given by the inverse color of the one used for 0 values in the histogram, and it has 50% transparency.

Upon releasing the mouse button (mouseup event), the actual zooming is performed. It is necessary to determine which data exactly will be displayed on the histogram. This is done by specifying which part of the _data_ array should be used, by defining starting and ending indexes. The previously mentioned `canvasToSensorCoordX()` and `canvasToSensorCoordY()` methods are used to convert the saved zooming coordinates to the corresponding indexes in the _data_ array. The visualization is then redrawn to only show the zoomed area, based to these indexes. Since the drawing functions are parameterized by the starting index and the number of pixels to be displayed, it works exactly the same way as described in the Data display section 5.5.2 on
5. Data visualization - frontend

page 48 only without the initial default values of 0 and 256.

5.5.5 Color maps

For each histogram it is possible to set a desired color map for the visualization. The color map decides in which color specified values are shown. For example, the simplest one – greyscale – returns a black color for minimum value, white color for maximum value, and different shades of grey for all values in between. The color maps are implemented as classes extending the abstract class ColorMap with each of them individually implementing its abstract method getRealColor(). This method takes a number between 0 and 100 as a parameter and returns a color assigned to this value.

All the color maps have some common functionality inherited from the ColorMap parent class. The getColor() public method trims the input value in case a value below 0 or above 100 was sent, before calling the getRealColor(). This class also provides the boosting functionality, which allows adding a flat value to all non-zero values passed to the color map, making the effect of shifting all the colors closer to the maximum, thus making every value appear brighter. It works also the other way around, when all the colors are toned down.

The ColorMap has a method called preview(), which takes an ID of an HTML5 canvas element and fully fills this canvas with a preview of the colors used by this specific color map. The color used for value 0 is on the left, the value 100 on the right, and other colors according to their proportionate corresponding values in between. The preview also offers an option to draw thin lines at the start, at 1/4, at 1/2, and at the end of the canvas to clearly mark these proportions for better and easier captioning with specific values.

To better separate pixel hits from the background, an option to display zeros in white color was implemented. It is for better display of very low values. For example, in greyscale color map, 0 is shown as black and very low values in very dark grey. Therefore, they are almost impossible to see. When the color of 0 is changed to white, the contrast between an “empty” pixel and some pixel hit with low value is much clearer. For providing this functionality to all color maps, without interfering with their code, the WhiteBackgroundDecorator class is made. It implements the IColorMap interface, therefore (publicly) acts in the same way as a regular color map. It keeps an instance of the specific color map being decorated as a class attribute and thus encapsulates all its functionality. It only changes the way, how the getColor() method returns a value when given 0 as an input parameter. The decorator takes any implementation of IColorMap as parameter of a constructor. Hence, it can be created over any already existing concrete color map.
5.5.6 Scales

To properly pass the correct value between 0 and 100 to the ColorMap class, scaling of the pixel values is needed. Therefore, the histograms have a reference to a Scale object. Scale is an abstract class and its most important method is the getScaledValue(), which takes three number parameters: a value, a minimum and a maximum, and returns a value between 0 and 100, according to the position of given value within minimum–maximum interval. Currently, two concrete classes, extending the Scale class, are the LinearScale and the LogarithmicScale.

One unit of difference on the logarithmic scale represents a change in the order of magnitude, which means that each value is the previous value multiplied by a specified number. It is useful when large values with big differences are present in the data. The linear scale is the standard scale. One unit of difference represents the equal difference in values.

5.6 Layout and styling

The Twitter Bootstrap is used for the coding part of the front-end implementation. The layout of the page is based on the Bootstrap grid system, which provides a fully responsive, mobile first approach using a series of rows and columns to house all elements within the grid layout, media queries, and predefined classes for easy layout options. The width of the screen is always divided into 12 columns. Classes for elements spanning across any number of these columns are provided. Other classes, like fixed-width containers, full-width containers, navigation bars, etc., are also available.

The whole visualization part of the page is placed within a fixed-width container and is centered on the screen. The first row contains the timeline controls, the second row is the timeline itself. It is placed within a bootstrap jumbotron element, which spans across the whole container width. The visualization sensors are each placed in an element with a width of 4 columns of the grid. The statistics, settings, filters, and errors section at the bottom of the page is using the toggle-able dynamic tabs. These allow switching between divs on the same place. The screenshot of the final application is shown in figure 5.5.

http://getbootstrap.com/
http://getbootstrap.com/css/#grid
5. Data visualization - frontend

Figure 5.5: Screenshot of final application
With the data processing back-end application and the front-end visualization tool, there has to be something facilitating the communication between those two parts. This chapter will describe the implementation of this communication, the server, and overall interconnection of all the parts of the application.

6.1 Integrating C++ on web

The majority of the functionality of the application is written in C++, including some functions vital for the visualization, such as retrieving the correct pixel interval from the ROOT file storage. It is needed to interconnect the C++ application with the web-based part of the application. There are several possible ways how to do this.

6.1.1 CGI

Common Gateway Interface is a standard for communication between web servers (such as http servers) and external applications. Using the CGI, the server runs a certain script in a pre-specified CGI directory, and serves its standard output as a http response. Instead of serving static files, it provides a dynamic content created by the executables (can be written in various languages, such as C++, Perl, Fortran, Unix shell, etc.) executed in real time [24]. It is a rather old technique [25] and has number of drawbacks, such as the need of a functioning http server supporting CGI, performance, and overhead issues caused, for example, by starting a new process every time the command is called, or by interpreting or compiling in case of not pre-compiled executables. These are a major issue when combined with the synchronous character of the solution. The user can be waiting for the server’s reaction for a long time, without receiving any response.
It is possible to at least partially bypass the drawbacks by implementing one of the improved protocols derived from CGI, such as FastCGI [26] or SCGI [27], which are supported, for example, in Apache HTTP Server, Nginx, Lighttpd, and others. However, the implementation of such solution still has disadvantages, in particular, the fact that the executable is run each time the data are requested. Therefore, the program has to start and end completely with each request. This makes caching or optimization during the program execution impossible. As the part of the application responsible for working with ROOT files and the stored data is utilizing caching of the opened files, this would introduce an avoidable performance decrease to the application.

6.1.2 Web server in C++

It is possible to write the whole web server in C++ from scratch using Internet sockets and other C/C++ features of network programming. There are several frameworks for implementing this, making the development easier and taking care of some basic functionalities. This option has a potential to provide a very good performance, but the time requirements for the development were estimated as too high for the benefit it can bring. Especially compared to using some other, standard solution that provides higher level of abstraction and takes care of the handling and parsing of requests, multithreading, asynchronosness, etc.

6.1.3 Node.js integration

As described in the analysis section[3.2.3, page 15], Node.js is a modern, lightweight, and efficient event-driven runtime used for web application development. It is simple to use and, due to its attributes, it is ideally suited for this application. It provides tools and features to build the required API for providing the data for the visualization and makes it easy to implement the server part of the application in general. It also offers ways how to interconnect with the application written in C++. They are as follows:

- Automation:
  Node.js supports the creation and management of child processes[16]. In its API, it has options for both, synchronous and asynchronous spawning of the process, running any .bat or .cmd files on Windows systems, or any other executable command line program. It also has a class ChildProcess that represents the spawned process for manipulating with it. As this only runs the compiled program, it is easy to use and it requires no language integration, but also has the disadvantage of CGI – poor performance, scaling, and running the whole program each time (therefore, no caching).

[16]https://nodejs.org/api/child_process.html
6.2. Creating native node.js add-on

- Shared library (dll):
  It is possible to build the C++ part of the application as a shared library, as Node.js allows calling C++ sub-routines from within its code, using for example the Foreign Function Interface add-on\(^\text{17}\).

- Native add-on:
  Creating a native add-on represents the complete integration of C++ code within Node.js. The add-ons are realized as dynamically-linked shared objects and can be loaded into Node.js using the `require()` function, same way as all other already made Node modules, and it is possible to run any C++ code as a native part of the Node.js application.

The latter option was selected for the implementation of interconnecting the C++ with the web server, as it can potentially provide very good performance, reduces latency, and allows for better customization and optimization within the C++ part of the application.

6.2 Creating native node.js add-on

The Node.js implementation itself states that “At the moment, the method for implementing addons is rather complicated”\(^\text{28}\), as it requires the involvement of several components and APIs, such as the internal Node.js libraries, C++ APIs\(^\text{18}\), the libuv C library implementing the Node.js event loop\(^\text{19}\), the node-gyp compilation tool\(^\text{20}\), and the V8 library for the JavaScript implementation\(^\text{21}\).

6.2.1 V8 engine

Node.js runs on Chrome’s V8 engine\(^\text{22}\) which is a JavaScript engine written in C++. It compiles JavaScript code, handles memory allocation, and garbage collects objects that are no longer needed\(^\text{23}\). It allows exposing the C++ functionality to JavaScript.

6.2.2 C++ code wrapping

In order to expose the C++ code as the add-on, it is necessary to create a V8 entry point and build wrapper code for the C++ functions. The entry point is specified in the `entrypoint.cc` file in the following way:

\(^{17}\text{https://github.com/node-ffi/node-ffi}\)
\(^{18}\text{https://nodejs.org/api/addons.html}\)
\(^{19}\text{https://github.com/libuv/libuv}\)
\(^{20}\text{https://github.com/nodejs/node-gyp}\)
\(^{21}\text{https://v8docs.nodesource.com/}\)
\(^{22}\text{https://developers.google.com/v8/}\)
\(^{23}\text{https://developers.google.com/v8/intro}\)
6. Server & communication

```cpp
void init(Local<Object> exports) {
    NODE_SET_METHOD(exports, "add", Add);
    NODE_SET_METHOD(exports, "getInterval", GetInterval);
    NODE_SET_METHOD(exports, "getTrigger", GetTrigger);
}

NODE_MODULE(tpx3data, init)
```

The node module has to export the initialization function which is called when the module is included/required. It does so in the NODE_MODULE macro, where the name of the module and the initialization function are specified. The function exports/registers the functions to be callable from Node. The NODE_SET_METHOD adds a function with name specified by its second argument to the exports and associates it with a native function in the third argument.

The signature of a such function is dictated by the V8 API. It is always a void function, taking one parameter of type const FunctionCallbackInfo<Value>& which represents a collection of arguments passed from JavaScript in the form of V8 Objects. This is the wrapper function and can contain any C++ code, but to keep the C++ code unchanged and clean, the code for the main application functionality is in a different, separate place. The following code shows the wrapper function for getting the pixel interval:

```cpp
void GetInterval(const FunctionCallbackInfo<Value>& args) {
    Isolate * isolate = args.GetIsolate();
    TPX3EntryPoint * ep = new TPX3EntryPoint();
    if (!CheckArgs(args))
    {
        return;
    }
    std::string s = ep->GetInterval(
        args[0]->NumberValue(),
        args[1]->NumberValue();
    args.GetReturnValue().Set(
        String::NewFromUtf8(isolate, s.c_str())
    );
}
```

The TPX3EntryPoint class represents the entry point (main) for the C++ application. It provides an access to the RootDataFacade and the other classes used for retrieving the correct pixel data. Isolate represents the V8 instance and thus, the current heap of the add-on. It is required to create new objects and primitives, for example, when creating a new string as a return value of the add-on’s function. A new string object is created in the Isolate instance based on a C string parameter. It is then set as a return value of the args
6.2. Creating native node.js add-on

collection. By this, a value from the C++ code is mapped to a JavaScript object. Mapping takes place also the other way around – the arguments are passed from the JavaScript function to a C++ code. The arguments are checked for being in the number format and converted to a C++ double by the `NumberValue()` method.

6.2.3 Building

For compiling the code and building the add-on, the node-gyp\(^\text{24}\), a tool specifically written to compile Node.js add-ons, is used. It is using a `binding.gyp` configuration file, which is written in a JSON-like format and specifies the add-on’s name, the source files, libraries, included directories, and a lot of other attributes. To compile all the required source files and produce the output add-on “tpx3data.node”, the following configuration is used:

```
"target_name": "tpx3data",
"sources": [ "tpx3entrypoint.cpp",
"entrypoint.cc",
'<!@(ls -1 dataprocessing/*.cpp)' ],
```

It includes both entry points as source files (both pure C++ application’s entry point and the V8 entry point), executes the command to list all the .cpp files in the dataprocessing source directory, and includes all of them.

With such a file prepared, node-gyp can generate and prepare the appropriate project build files, depending on the platform (it can generate Makefile for Unix and vcxproj for Windows). Node-gyp can then build the add-on and produces the compiled .node file, which is a binary representation of the created add-on.

The add-on can then be used from within Node.js code as follows:

```
var tpx3data = require('../cpp/build/Release/tpx3data');
```

6.2.4 Externalities handling

An important aspect of the application is working with external libraries, especially those needed for the ROOT framework. Node-gyp has the “libraries” option in the `binding.gyp` file, as well as “include_dirs”. All the ROOT’s libraries from its /lib directory, static and dynamic libraries (.a and .so files, respectively) are specified in the “libraries” options. For accessing the ROOT’s own functionality and including it in the code, the headers and the binaries have to be available. This is done by adding ROOT’s /include and /bin directories to “include_dirs” option. Any of the ROOT header files are then included by:

```
#include "TTree.h"
```

\(^{24}\)https://github.com/nodejs/node-gyp
6. Server & communication

6.2.5 Platform compatibility

The node-gyp tool and Node.js in general are platform independent. The ROOT framework has a distribution for each of the platforms, although so far, all versions, including the used 5.34, support only 32bit versions of Windows. At the moment, the presented application uses 64bit versions of Node.js and node-gyp. Therefore, the produced add-on can use only the 64bit version of ROOT and is thus compatible only with Unix based platforms. By using the 32bit version of Node.js and making some minor changes, the application is compatible with Windows, as the C++ code itself is written to be platform independent using #ifndef constructs, such as:

```c
#ifndef _WIN32
 #include <Windows.h>
#endif
```

6.3 Node.js application

The Node.js application and the implementation of server itself uses an Express framework. It is a minimalist framework for creating web and mobile applications, and for quick and easy building of APIs.

6.3.1 Routes & middleware

The Express framework uses routes–views structure. Each route defines some functionality that is bound to a client request to a certain endpoint (URI and a request method) of the application. The route is defined by the request method, a path and a handler:

```javascript
router.get('/trigger/:triggerNo', function(req,res,next) {
  //handler logic
  res.json(tpx3data.getTrigger(Number(triggerNo)));
});
```

In this case, when the user sends a GET request on a “/trigger/:triggerNo” path on the server, where :triggerNo is a number parameter, the server responds with a JSON data from the getTrigger() function of the tpx3data add-on.

The views are templates, which can be rendered by the templating engine to display an HTML page. The chosen templating engine and language is jade. It simplifies the syntax of writing HTML structure and the processing of any variables passed from Node. For rendering a template, the route calls a render() method on a response object, with any number of arguments.

6.3.2 Query structure

The application uses an Express.Router class, that helps to create a more modular and better structure of the route handlers. It creates a router as a module, which defines its own middleware function and routes. This router module can be mounted on a certain path in the application by:

```javascript
app.use('/data', data);
```

This mounts a router module “data” to a “/data/” path on the server. The above mentioned route for getting the trigger data is part of this module. Therefore, the complete path to get the data from trigger X is “/data/trigger/X”. There are three router modules in the application: data, rootfiles and asciifiles. Each encapsulates the corresponding functionality and is mounted on its own path (“/data”, “rootfiles” and “asciifiles”, respectively).

Using the routes with the specified paths, combined with using the router modules, results in creation of the query structure defined in the API description in section 3.6 on page 19.
Chapter 7

Future

7.1 Deployment

The application is fully functional and ready to be deployed on a Unix based server. The requirements, prerequisites and installation guide is in the appendix C. It will be running on one of the servers at IEAP and providing the data from several Timepix3 devices. The API will be accessible from separately distributed client parts of the application.

Figure 7.1: Multi-sensor practical application
7. Future

7.2 Multi-sensor view

There is a room for further improvement of the application and potential in expanding the functionality by adding new useful features. One of the required and prepared new functionalities is a multi-sensor view. The visualization will allow a simultaneous display of data from multiple detectors (multiple data sources). The user will be able to choose any number of available data sources and the application will provide a clear way how to display a comparison of the selected datasets on the sensor squares side by side. Each of the sensors will also have its own timeline and tools to set the desired time intervals. This way, it will be possible to examine what happened on one sensor in one time and compare it with what happened on another sensor any specified (short) time later. This is beneficial for particle tracking applications (reconstructing the trajectory of particle passing through a telescope of several detectors) or for coincidence analyses. In some applications, several sensors are placed one after the other and from position and strength of certain particle hit in at least two of them, it is possible to re-calculate the particle movement through space. A sketch of a Timepix based particle telescope is illustrated in figure 7.1.

The designed application layout in the multi-sensor view is shown in wireframe in figure 7.2. The elements on the page are slightly repositioned.
7.2. Multi-sensor view

The biggest difference is that each individual histogram square will not represent one type of data attribute, but instead the data from one specific sensor. The different data attributes can be displayed by switching the tabs on top of each 2D histogram. The timelines for each sensor will be placed below each other, to always display the chosen interval and the amount of data in time on each sensor. This facilitates the identification of time coincidences, even by eye.
A fully functional visualization tool for the Timepix3 data was created. For the development, various skills and techniques, such as C++ programming, web development, front-end programming and user interface design, had to be used and combined to create a complex and functional project. The application was written in a highly modular form, so that each of its parts can be used, improved, or changed separately. The emphasis was put on good application design and clear, readable and re-usable code.

In the data processing part, an independent command line executable was created, which converts the raw data from the ASCII format to a more efficient ROOT database, while performing a pre-processing of the data. The Data Facade was implemented, exposing an interface for working with the data. It uses data indexing and other methods to ensure a fast and efficient retrieval of certain data based on time specification or other user queries. It can be also included in any C++ application for further data analysis.

The visualization client-part of the application was implemented, providing a complex toolset for displaying and examining the pixel data, including interactive 1D and 2D charts, data statistics and custom filtering options. The visualization is compatible with all modern web browsers and it is responsive to screen size.

To put all the parts of the application together and construct the server-client structure, a Node.js web application was developed and options of interconnecting between Node.js and the C++ language were examined. For efficient and fast connection, the possibility of creating native Node.js addons was explored. Using Chrome’s V8 engine, the add-on for the Timepix3 data retrieval was created, the C++ code successfully connected, and made usable from JavaScript.

Future application specific improvements, such as a multi-sensor view, of the web visualization are addressed. The modular character should make improvement of each of its part a smooth process and make it possible to add whole new modules of functionality.


Bibliography


Acronyms

ASCII  American Standard Code for Information Interchange
API   Application Programming Interface
REST  Representational State Transfer
CERN  Conseil Europen pour la Recherche Nuclaire
LHC   Large Hadron Collider
ASIC  Application Specific Integrated Circuit
MRI   Magnetic Resonance Imaging
IEAP  Institute of Experimental and Applied Physics
CTU   Czech Technical University
GUI   Graphical user interface
HTML  HyperText Markup Language
CSS   Cascading Style Sheets
AJAX  Asynchronous JavaScript and XML
HTTP  HyperText Transfer Protocol
URI   Uniform Resource Identifier
URL   Uniform Resource Locator
SASS  Syntactically Awesome StyleSheets
JSON  JavaScript Object Notation
A. Acronyms

 DLL Dynamic-link library
 CGI Common Gateway Interface
 UI User Interface
 IDE Integrated Development Environment
Appendix B

Contents of enclosed CD

- readme.txt ......................... the file with CD contents description
- dist .................................. the directory with compiled sources
- converter .......................... binary files for running the converter
- server .............................. files needed for deploying the server
- client ............................... files for separate, individual client distribution
- src .................................. the directory of source codes
- wbdcm ................................ implementation sources
- thesis .............................. the directory of \LaTeX\ source codes of the thesis
- text .................................. the thesis text directory
- DP_Vycpalek_Jiri_2016.pdf ........ the thesis text in PDF format
- docs ................................... documentation
- cpp .................................. Doxygen documentation of C++ source
- api .................................. API specification and documentation
- ts .................................. TypeScript documentation

75
C.1 The server

The following installation instructions for the server part of the application require Unix based system. It was not tested and functionality is not guaranteed on other platforms.

C.1.1 Deploy

The needed files are located in the /dist/server/ directory.

1. Prerequisites:
   - node.js v4.4.3 (JavaScript runtime)
   - npm v2.15.1 (Node.js package manager)
   - ROOT v5.34 (Data analysis framework)

2. Export the ROOT /build/lib/ directory to default library path variable by running command

   ```bash
   export LD_LIBRARY_PATH=path/to/root/build
   /lib/:$LD_LIBRARY_PATH
   ```

   for server application to correctly locate the ROOT libraries.

3. Install the server and its dependencies by running

   ```bash
   npm install
   ```

---

27 The application was developed and tested on Debian 8 (“jessie”) 64-bit
28 http://nodejs.org/
29 https://www.npmjs.com/
30 https://root.cern.ch/installing-root-source

Follow the Building ROOT manual:
C. Installation guide

from the server’s root directory.

4. Run the server

    npm start

Server is now running on the local machine on port 3000. It provides the API service for the separately distributed client part of the application, or any other application.

It can also be accessed directly from browser to provide the whole functionality of the application.

C.1.2 Build

The source files are located in the /src/wbdcm/server/ directory.

1. Prerequisites:
   - All of the above prerequisites
   - node-gyp v3.3.1 (Node.js native add-on build tool)
   - tsc v1.8.10 (TypeScript compiler)
   - Sass v3.4.22 (CSS extension language compiler)
   - Compass v1.0.3 (CSS authoring framework)

2. Edit the binding.gyp file in the /cpp/ directory to contain correct paths to ROOT libraries and sources in the “libraries” and “include_dirs” options.

3. Export the ROOT libraries directory to library path (see above)

4. Run

    node-gyp configure

    in the /cpp/ directory, to create platform-specific project build files (Unix Makefile, Windows .vcxproj) according to specified configuration.

5. Compile the project by running

    node-gyp build

    in the /cpp/ directory.

31 https://github.com/nodejs/node-gyp
32 http://www.typescriptlang.org/
33 http://sass-lang.com/
34 http://compass-style.org/
6. Run command
   
   \texttt{tsc}

   in the /public/javascripts/ts/ directory, to compile TypeScript sources into JavaScript.

7. Run command
   
   \texttt{compass compile}

   in the /public/stylesheets/sass/ directory, to compile Sass sources into .css.

8. Run the server.

\textbf{C.2 \ The converter}

\textbf{C.2.1 \ Run}

The program for converting the data from the ASCII format to ROOT storage is a binary file \texttt{tpx3dataconverter} located in /dist/converter/ folder. It can be run as follows:

\texttt{tpx3dataconverter input_ascii_file output_root_file action_option subject_option}

Where

- \texttt{input_ascii_file} is the location of the source ASCII file
- \texttt{output_root_file} is the desired name of the ROOT file
- \texttt{action_option} is the desired action to be performed, can be \texttt{load} for loading and converting the data or \texttt{show} for browsing the data
- \texttt{subject_option} can be either \texttt{data} or \texttt{indexes} to choose whether the shown data from ROOT file will be the pixel data, or the indexes

\textbf{C.2.2 \ Compile}

Use any IDE with correctly set include and header directories to ROOT /build/include/ directory and linker libraries to /build/lib/. Compile all the C++ sources in the /src/wbdcn/converter/ folder to create an output executable (Unix binary or Windows .exe).
C. Installation guide

C.3 The client

C.3.1 Run

The files needed for running the client part of the application are located in the /dist/client/ folder.

In the config.js file, the address of the API server providing the data can be specified. Default option is “localhost” for server running on a local machine. For testing purposes, “test” can be used for accessing only local data from within JavaScript.35

To run the client, open the index.html file.

C.3.2 Compile

All the needed source files are located in /src/wbdcm/client/ directory.

1. Compile the TypeScript sources by running the command
tsc
   in the /javascripts/ts/ folder.

2. For compiling the SASS sources, use command
   compass compile
   in the /stylesheets/sass/ folder.

3. Open the index.html file.

---

35Note that in this case, any functionality, that uses querying the server for new data, will not be available. This option is only for static display of one interval of data.