KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Silesian University of Technology
Faculty of Automatic Control, Electronics and Computer Science
Institute of Computer Science

Thesis for the degree of Doctor of Philosophy
in Computer Science

Quaternions based human motion analysis algorithms implemented with data flow processing framework for Motion Data Editor software

Mateusz Janiak

Thesis Advisor:	prof. dr hab. inż. Konrad Wojciechowski
Consultants:	dr inż. Agnieszka Szczęsna dr inż. Janusz Słupik

Gliwice, September 2013

Contents

1 Introduction **1**

2 Background knowledge **7**
2.1 Motion analysis 7
2.2 Motion data 7
2.3 Acquisition equipment 9
2.4 Data storage 10
2.4.1 File formats 12
2.4.2 Personal data protection 12
2.5 Available motion analysis tools 15

3 Motion Data Editor **19**
3.1 Data processing pipeline 21
3.1.1 Browse 21
3.1.2 Load 21
3.1.3 View 22
3.1.4 Process 23
3.1.5 Save 23
3.2 System architecture 23
3.2.1 Core data types 25
3.2.2 Core processing logic components 35
3.2.3 Core functionality 41
3.2.4 Managers 49
3.2.5 Code organization 63
3.3 Built-in plug-ins 67
3.3.1 c3d 67
3.3.2 communication 68
3.3.3 kinematic 69
3.3.4 chart 70
3.3.5 timeline 71
3.3.6 subject 73

3.3.7 video . . . 73
3.3.8 python . . . 74
3.4 Implementation and development . . . 75
3.5 Future work . . . 82

4 Data flow processing and visual programming 84
4.1 Introduction to data flow processing . . . 84
4.1.1 Data flow graph concept . . . 84
4.1.2 Data flow model design . . . 86
4.1.3 Data flow model processing extensions . . . 88
4.1.4 Data flow processing logic . . . 90
4.2 General purpose data flow processing library . . . 93
4.2.1 Processing logic characteristic . . . 93
4.2.2 Public interfaces . . . 94
4.2.3 Processing logic details . . . 96
4.2.4 Executing processing logic . . . 101
4.3 Data flow plug-ins for Motion Data Editor (MDE) . . . 102
4.3.1 Data exchange mechanism . . . 102
4.3.2 Optimal computational resources utilization . . . 103
4.3.3 Visual programming . . . 103
4.4 Future work . . . 108

5 Motion analysis 110
5.1 State of the art . . . 110
5.2 Quaternions . . . 111
5.2.1 Introduction . . . 111
5.2.2 Unit quaternions . . . 113
5.2.3 Quaternion functions . . . 114
5.2.4 Rotations . . . 114
5.2.5 Interpolation . . . 116
5.3 Multi-resolution analysis . . . 117
5.3.1 Introduction . . . 117
5.3.2 Lifting scheme . . . 118
5.4 Lifting schema for quaternions . . . 120
5.4.1 Rotation average value . . . 120
5.4.2 Proposed lifting schemes . . . 122
5.5 Applications . . . 131
5.5.1 Noise reduction . . . 131
5.5.2 Data compression . . . 131
5.6 Tests . . . 132
5.6.1 Test data . . . 132

5.6.2 Comparing results 133
5.6.3 Signal reconstruction 135
5.6.4 Noise reduction 141
5.6.5 Compression 148
5.7 Implementation overview for MDE 154
5.8 Summary and future work 160

6 Summary and final conclusions **162**

Appendix **166**

Streszczenie **167**

List of Figures **188**

List of Tables **190**

List of Algorithms **191**

List of Acronyms **193**

Bibliography **194**

Chapter 1

Introduction

Nowadays, due to the possibility of collecting great amount of different kind of data about the surrounding world, the topic of simple and efficient data analysis became very important. Simple - because analysis is not done any more only by scientists and engineers, but also by less qualified stuff in the field of mathematical analysis and computer science. Efficient - as analysis does not cover any more only scalar (numerical) values, but also complex digital data like images, audio or video, which require much larger storage space with memory and greater processing power.

For managing and analysing big data specialized solutions exist, based on a dedicated software with database servers and data warehouses [36, 30, 69]. For processing small and medium data size there are much more possibilities, starting with general purpose applications like spreadsheets, through computer algebra systems [73, 74, 44] and dedicated scientific programming languages [21, 17], ending with custom solutions, fit to the particular needs of their users. However, going through many different software and technologies, there are very few, that are oriented on flexible, intuitive to use, general data processing. Even less solutions supporting motion data processing can be found. The lack of dedicated motion analysis tools slows down and limits significantly research and development in areas like:

- medicine (orthopaedics and neurologists),
- sport (training progress measurement, personal training programs, technique/strategy comparison),
- security (recognition of people based on their moves, detection of possibly dangerous situations),

- entertainment (more realistic computer games, avatars, augmented reality).

In this thesis a new software - Motion Data Editor (MDE) developed at Polsko-Japońska Wyższa Szkoła Technik Komputerowych (PJWSTK) - for general purpose data processing is presented. It is shown, how this application supports complete data processing pipeline, from data loading, management and normalization, browsing and visualizing, up to data processing and results reporting. Its architecture and logic are described at conceptual level, with limited code examples, as they can be implemented with various programming languages based on different technologies. The most important framework features are presented with delivered out-of-the-box functionalities, where some of them are dedicated explicitly to motion data analysis.

I am one of the core programmers developing presented framework. I have designed most of its architecture and logic presented in this thesis. My most important and original work in the field of MDE development is:

- creating *DataChannel* generic type for time indexed data with various helper classes for time specific operations,
- parsers functionality decomposition on custom input/output (I/O) operations and data stream support, optimizing performance of data loading,
- designing lazy initialization mechanism based on parsers functionality for data loaded from files and streams,
- proposing meta-data extension for novel mechanism handling data in an uniform manner, independently from data types,
- designing hierarchical messages logging system architecture,
- decomposition of core system elements functionalities on public and private,
- proposition of concepts supporting efficient and optimal data processing: thread pool, job and jobs manager,
- introduction of application context,
- design of plug-ins initialization procedure with specialised application contexts,

- proposition of abstraction layer for time operations for hierarchically organised time based data (*Timeline*),
- concept of abstract application object, initializing complete framework logic with graphical user interface (GUI) and application launching procedure,
- data management functionality decomposition on independent managers: supported data types, data objects, files and streams,
- concept of transactions, offering isolation for application state modifications, increasing their performance in multi-threaded environment,
- design of flexible and efficient data processing module in form of data flow,
- concept of hierarchical motion data organization,
- introduction of well known design patterns [54] to most of the presented objects,
- design of Continuous Integration (CI) process [29, 16] supporting MDE development.

This thesis also summarizes and extends information about novel tools for motion analysis presented in [67, 66, 68]. My most important and original work in this field is:

- designing an universal mechanism for loading medical data,
- proposing testing framework and general purpose tools for various experiments with real and synthetic motion data,
- designing lifting scheme based on *squad* interpolation method,
- proposition of motion data compression based on multi-resolution data representation,
- designing developed tools to work with MDE software.

For the clarity and completeness, all proposed motion analysis tools are described in this work with their applications. Their development and this work were supported by the European Union from the European Social Fund (grant agreement number: *UDA-POKL.04.01.01-00-106/09-02*).

Additionally, in this thesis I give a brief overview of motion analysis research field. I point out potential problems, that I have encountered taking part in different projects connected with human motion. Based on gained experiences, I give some guides and hints to those problems in form of general concepts, that were applied in MDE.

There are formulated two thesis statements:

Thesis 1 MDE software is designed as a general purpose data processing application. It allows simple customizations to particular users needs with a dedicated plug-in system. MDE supports complete, well defined, data loading procedure. Application ensures efficient and uniform data management, providing tools for optimal utilization of local computer computational resources. Data browsing is standardized for all supported data types to present them at various perspectives.

Thesis 2 Proposed tools for motion analysis, based on multi-resolution analysis and quaternion signal representation, provide good motion data decomposition properties for noise reduction and compression algorithms. Moreover, they can be easily implemented for MDE software, making various experiments extremely easy to perform. Very few work must be done to adopt already developed solutions to presented data processing framework and utilize automatically all available computational resources for custom calculations.

The goal of this thesis is to present in the first step the power and flexibility of developed data processing framework, which come from MDE design and provided functionalities. This is presented in Chapter 3 and Chapter 4. Secondly, I want to implement and present developed tools for motion analysis to work inside MDE. This is elaborated in Chapter 5.

As thesis presents not only my work, but I have precisely pointed out my most important and original work, I am using "we" along the rest of the thesis to underline team work in the following areas:

- MDE development,
- motion analysis research work.

Finishing the introduction, the following thesis structure is given:

Chapter 2 This chapter gives a brief introduction to motion analysis domain. We describe the most important data types, their various formats and technologies allowing to record them. Available motion data

sources are pointed out. Potential problems with storing motion data are presented. We show currently available software dedicated to motion analysis. We describe their limitations, which make them incomplete in therms of data analysis. This chapter presents the motivation to create a general purpose data processing and analysis software with dedicated motion analysis solutions.

Chapter 3 In this chapter MDE software is presented. Elaborating system logic and architecture only general concepts are described, skipping most of the technical and implementation details. We want to familiarize potential users and developers with application structure by presenting its main features and advantages over other solutions. Sophisticated diagrams are given to make the logic and concepts understanding easier.

To show application flexibility, several built-in plug-ins are presented with a plug-in system itself. Additionally, application development process is described to show, how small, geographically spread developers team members can co-operate to produce high quality software, limiting costs of development and its risk to minimum.

We describe a new data storage and management concept for C++ programming language [42, 33], considering data representation in programming languages like *Java* [50] or *C#* [3], where all types have a common base type called *Object*. To prove MDE is a general data processing tool with great support for motion analysis a generic time indexed data type is presented.

This chapter presents capabilities and advantages of MDE over other motion analysis tools. We want to encourage interested in researchers and developers to think about MDE in terms of their own work and find its features useful enough to try it out. This chapter is mostly dedicated for software developers who would like to contribute in development of different kind of plug-ins for MDE and MDE application itself. Also a general list of application features is given with a brief summary, pointing out further software development directions.

Chapter 4 A concept of a simple, yet efficient data processing in form of data flow is presented. To make it easy to use for various users (especially those without programming skills and mathematical background)

an extension is given in form of a visual programming environment. This part presents additional MDE functionality that simplifies data analysis procedure. It is shown how easily such feature can be implemented for MDE, based on MDE core architecture. Similarly to Chapter 3, most sections are written for engineers and software developers with intermediate knowledge level in topics of parallel programming and data processing. In the end a short summary should give the final impression, how to use the data flow hidden behind intuitive GUI.

Chapter 5 In this chapter a new approach to motion analysis is presented. It is based on multi-resolution techniques for motion data in a quaternion representation of rotations. Additionally, we propose applications of presented algorithms for data compression and noise reduction. It is described how various tests and experiments of developed tools were implemented for MDE, introducing custom solutions to a general data processing framework through dedicated plug-in system.

Chapter 6 This chapter recapitulates thesis work. MDE main features, supporting general purpose data processing and analysis, are collected and recalled. Proposed tools for motion analysis are summarized and their further research directions are proposed. Also their implementation for MDE is recalled to underline simplicity of creating new functionalities and their usage in MDE.

Appendix A description of content and structure of attached CD is presented in this short chapter.

Chapter 2

Background knowledge

This chapter is dedicated to readers getting familiar with motion data and analysis, giving a rough introduction to this topic. For readers having experience in motion analysis it is suggested to move forward to Chapter 3, if they are interested in MDE software and its features, or to Chapter 5, if they are only interested in novel approach to motion data analysis tools.

2.1 Motion analysis

Motion analysis is a wide research area. By motion analysis we consider analysis of hierarchical models representing human (potentially other living being) musculo-skeletal system (Figure 2.1). They are built up with segments (bones), connected through different type of joints. In particular, it is possible distinguish two types of motion analysis:

- basic - covering only rigid body kinematics and kinetics [28, 76],
- extended - trying additionally to analyse correlation between mechanical and medical data [48, 53].

It is explained in Section 2.2, what kind of medical data can be used for extended motion analysis. We do a research in that direction, but we are concentrating mainly on the basic analysis, as firstly its superb understanding is required in this field. As an example, a research on Parkinson's disease was done [55].

2.2 Motion data

Motion analysis concerns variety of motion-specific physical measurements with different representations. Most of motion data types are presented in

Figure 2.1: Human musculo-skeletal system
(`http://tylersmusculoskeletalsystem.weebly.com/uploads/1/7/2/8/17289344/465421_orig.jpg`)

Table 2.1: Motion data types

Data type	**Description**
video	Video streams are crucial source of information about motion presenting its general view.
electromyography (EMG)	This data describes how muscles were activated during the motion either by electrical or neurological signals. Generally, several such signals are recorded for different body parts connected together (i.e. arm and fore-hand).
ground reaction forces (GRF)	Measure the force vectors applied to the ground (typically by the feet). They allow to analyse kinetics of body mainly during different gait phases.
linear and angular velocities with accelerations	Represent local relations between segments in hierarchy or their global behaviour.
points cloud	Positions in a 3D space of a well defined set of points on the body.

Table 2.2: Basic motion recording equipment

Equipment	**Recorded data**
video camera	video recordings, usually High Definition (HD), minimum one stream, typically four streams from different perspectives (left, right, front, back), after post-processing 3D reconstruction is possible
infra-red cameras	Distance measure: depth map, points cloud
electrocardiograph	EMG data, usually analogue, several streams on connected body parts (i.e. arm and forearm)
ground reaction force plates	GRF stream, 3D vector describing force that feet generates when touching the plate, minimum one platform required, but usually two platforms are used in gait analysis to capture complete gait cycle
inertial measurement unit (IMU)	Provides angular velocities, accelerations and magnetic field vectors, additionally can provide orientations according to fixed reference coordinate system

Table 2.1. The number of presented types underline motion multi-modal characteristic and reveals complexity of motion analysis.

Despite variety of data types describing the motion, there are also many different kind of meta-data, that need to be delivered to make motion description complete. One of such meta-data might be skeletal model with proper data mapping, connecting raw physical measurements with musculo-skeletal system. Also, properties of recorded values must be given. Some of them describe registration frequencies, value ranges and units. Those properties make motion analysis even more complicated, as all data must be normalized and this is not a trivial task, especially when different skeleton models and hardware are used. What is also very important, motion data are indexed with time and that forces analysed data to be synchronized in time. This task also might be very difficult, as many different types of recording equipment are used with different characteristics and synchronization techniques.

2.3 Acquisition equipment

As already mentioned, many different types of hardware can be used to record motion. Now we briefly present how to obtain particular motion measurements. Table 2.2 presents basic measurement devices with data types they

can provide. Complex solutions are presented in Table 2.3, where composition of basic measurement devices allows more detailed motion registration. If someone is not interested in recording motion data, there are also some free, open-source databases in the Internet. The most know motion databases are:

- the Carnegie Mellon University Library - `http://mocap.cs.cmu.edu`
- Kitchen Challenge - `http://kitchen.cs.cmu.edu`
- the ACCAD Motion Capture Lab Library - `http://accad.osu.edu/research/mocap`
- the EYES JAPAN library - `http://www.mocapdata.com`
- the Motion Capture Club motions - `http://www.mocapclub.com`
- the Universität Bonn Mocap Database HDM05 - `http://www.mpi-inf.mpg.de/resources/HDM05`

Using them, however, require great care, as their quality might be poor and not all required information to use the data might be given explicitly or not at all. It is always a good solution to take data from specialized motion laboratories.

A short note about complex motion measurement solutions must be given. Although they are very advance, they always require specific calibration procedures. They involve usually a set of well defined moves, which allow particular systems to fit built-in skeletal model to the person being actually recorded and to define precisely initial recording state. Those exercises might seem to be simple, but they should be done with great care to obtain reliable data. Despite calibration procedure and the advance of technology, it might be still required to fill or improve some of the recorded data. This procedure is called post-processing, it is done by a qualified stuff, as hardware might sometimes miss data samples because of technological limitations and particular recording conditions (i.e. MoCap systems, when particular marker is not visible by the required, minimal number of cameras).

2.4 Data storage

This section discuss topic of storing motion data. Two main problems are presented - large amount of possible file formats and law regulations about personal, private data security in Poland.

Table 2.3: Complex motion recording systems

System	**Provided data**
Motion controllers	points cloud in space based on the distance measure with stereo infra-red cameras, usually simple skeleton is given
Vision system	2D images from different cameras in known locations and orientations are used to reconstruct 3D environment, depth maps, silhouettes or skeletons can be obtained
motion capture (MoCap) system	Usually complete set of data from points cloud, through EMG, GRF and video. Usually based on infra-red distance measure, which limits more complex motions. Additionally motion must be recorded in particular place, where its dimensions might limit its application (cycling, skiing)
Acquisition costume	Dedicated clothing with built-in IMUs and other required hardware. In contradiction to MoCap systems it provides recording of almost any kind of motion, as it is mobile. Provided data quality depends on previous calibration process and environment (magnetometer measurements vulnerable to metal objects)

2.4.1 File formats

Despite different kinds of registered motion data types, it is possible to choose between many specialized data formats to store this data. There is no one, well defined format allowing to store all of motion data. This forces to store different data types separately, although they describe logically the same motion. There were developed many formats to handle particular data independently or in some specified combinations, but in the end it is very confusing to manage and use them together. In Table 2.4 the most known motion data formats are presented based on the Biomechanical ToolKit (B-tk) library capabilities.

Table 2.4 omits video data formats purposely, as this is completely different topic and it is not relevant for this work, although it also has to be considered while storing motion data. Also file formats for describing skeletal model and data bindings between model and recorded data were skipped in this summary, but they also must be delivered for reliable motion analysis. Choice of accepted formats affects software performance. Based on our experience and general trends in motion analysis we suggest using C3D as a container for most motion data (despite video). It allows storing custom data, making it universal and flexible for almost any application.

We also suggest to develop a dedicated motion data storage system for efficient and secure motion data sharing. As an example a Human Motion Database (HMDB) [19] can be given, developed in parallel to MDE at PJWSTK. It combines Web Services with File Transfer Protocol (FTP) and file database to provide motion data. HMDB manages all motion data recorded in Human Motion Laboratory (HML) (`http://hml.pjwstk.edu.pl/hml.pjwstk.edu.pl`), that PJWSTK poses. It is worth mentioning that PJWSTK plans to offer through the HMDB a set of free motion data on-line for common use. With such an approach it can be noticed, that PJWSTK provides complete solutions for motion analysis from data acquisition and storage to data processing software.

2.4.2 Personal data protection

It must be mentioned, that motion data require special treatment. This is caused by the law regulations, defining various data types as being sensitive data, which must be secured from an unauthorized access and usage. In particular, the law regulations treat video recordings with people faces to be sensitive data. As this is one of the most fundamental data types describing motion, storing it forces some additional solutions to ensure its protection.

Table 2.4: Different file formats for storing motion data

File Format	**Comments**
ANB	Motion Analysis binary file format containing analogue channel data
ANC	Motion Analysis ASCII file format containing analogue channel data
ANG1	BTS Bioengineering (Elite) binary file format containing (joint) angles
ASC2	AMTI ASCII file format containing FxFyFzMxMyMz data
C3D	Most common used file format. Supports storing MoCap data as well processed data and custom analogue channels and other custom scalar data. More informations on `http://www.c3d.org`
CAL	Motion Analysis ASCII file format for force platform calibration
CLB	Contec Inc. binary file format containing analogue channel data
EMF	Ascension Technology Corporation ASCII file format containing 3D trajectories
EMG	Delsys Inc. binary file format containing EMG data
EMG	BTS Bioengineering (Elite) binary file format containing EMG data
GR*3	BTS Bioengineering (Elite) binary file format containing force platform data
MOM4	BTS Bioengineering (Elite) binary file format containing joints' moments
MDF	Charnwood Dynamics Ltd (Codamotion) binary file format containing 3D trajectories, analogue data and force platform geometry
PWR	BTS Bioengineering (Elite) binary file format containing joints' powers
RAH	BTS Bioengineering (Elite) binary file format containing joints' powers
RAW	BTS Bioengineering (Elite) binary file format containing 3D trajectories
RIC	BTS Bioengineering (Elite) binary file format containing 3D trajectories
RIF	BTS Bioengineering (Elite) binary file format containing 3D trajectories
TDF	BTS Bioengineering binary file format containing 3D trajectories, analogue data and force platform geometry
TRB	Motion Analysis binary file format containing 3D trajectories
TRC	Motion Analysis ASCII file format containing 3D trajectories
XLS	Motion Analysis ASCII file format exported from the software Orthotrack
XMOVE	Charnwood Dynamics Ltd (Codamotion) XML file format containing 3D trajectories, analogue data and force platform geometry

One way to overcome this problem is to introduce anonymity to videos, so that faces can not be recognized any more in such materials. There are two approaches to this problem - editing videos before storing them, so that faces can not be recognized, or implement an abstraction layer, which would apply face blurring during the playback and grant original recording for video analysis (i.e. face micro-expressions). For our applications first solution is used - dedicated tools and algorithms blur people faces before storing video files.

Despite motion data, there can be stored also some additional personal data, especially when we consider medical patients. In this case even greater care must be taken to secure those data, as they might contain people names, surnames, birth dates, health security identifiers, addresses. In Poland main sources of knowledge about law regulations for personal, private data protection can be found in:

- Ustawa z dnia 29 sierpnia 1997 r. o ochronie danych osobowych - `http://isip.sejm.gov.pl/DetailsServlet?id=WDU19971330883`
- Rzecznik Praw Obywatelskich - `http://www.brpo.gov.pl`
- Generalny Inspektor Ochrony Danych Ososbowych - `http://www.giodo.gov.pl/`

Information provided by this sources and regulations they are imposing should be strictly obeyed to ensure required data security and prevent any legal charges against stored informations.

When such data are collected and managed in any way, a good idea is to separate motion data and personal data from each other as much as possible (connected only through some artificial indexes). Access to both of them must be verified, however access to private, personal data should be limited to minimum number of users. Simple information, however, might be shared publicly. This could be gender, age, height and weight of recorder people to allow basic statistical analysis of human motion, not violating private data protection regulations. To get more information about this topic a contact with presented organizations or specialized lawyer is suggested.

Table 2.5: Motion analysis tools comparison

	Application	
Feature	**Vicon Polygon 4**	**Mokka (Motion kinematic & kinetic analyzer)**
Platforms	Windows 7 (x86 and x64)	Cross-platform
License	Commercial	Free
Video browsing	●	●
EMG browsing	●	●
GRF browsing	●	●
IMU browsing	❍	●
Reports	●	❍
Time line and events	●	●
2D charts	●	●
3D scenes and models	●	●
Data import formats	ASF/AMC, C3D, on-line from Vicon systems	as B-tk
Data export formats	ASF/AMC, C3D	as B-tk
Custom user layouts	●	●
Motion database	❍	❍
Plug-ins	❍	❍ / ● (partial)
Data processing	❍	❍ / ● (partial)
Scripting	❍	❍
Translations	❍	❍
Custom style sheets	❍	❍

2.5 Available motion analysis tools

Currently there are only two applications in the field of motion analysis, mature enough and reliable that are worth mentioning:

- Vicon Polygon 4 (`http://www.vicon.com/Software/Polygon`)
- Mokka (`http://b-tk.googlecode.com/svn/web/mokka/index.html`).

Other solutions, generally small applications dedicated to particular problem connected with motion analysis, are not developed any more and their stability and usability are at a very low level.

Table 2.5 presents complete comparison of Vicon Polygon 4 and Mokka. Later we describe MDE according to the same criteria, to show its advan-

tages and improvements in comparison to those applications.

Comparing Mokka and Vicon Polygon 4 we were interested in a number and types of platforms they support, as different users might have various preferences in this field. We take into account capability to manage different motion data formats and motion data types. We are also considering browsing data at different perspectives (as 2D charts and 3D scenes), because many motion data might be presented in both ways. Very important aspect of motion analysis is efficient time data management and synchronization. Reporting features, data export and analysis results exchange between users are also verified. Applications flexibility in extending built-in functionalities with custom ones is taken into account. Software scripting capabilities are investigated, as they might speed up process of data analysis offering more general and flexible solutions. We score possibility of motion data processing, which we find to be one of the most important features. In the end application language translations and custom styling capabilities are verified as the least important features.

As it can be noticed both applications are oriented mainly on browsing motion data at different perspectives. Mokka offers some possibility to create a processing pipelines through the B-tk library, but it is hard to use it for a custom data processing, as it supports only strictly defined data format and requires programming skills. It also does not consider utilization of all available processing power, offering only sequential data processing. The lack of functionality supporting efficient data processing limits application of presented tools for complex research in motion analysis.

Vicon Polygon 4 provides some custom extensions, supporting their hardware for motion acquisition. Mokka however offers support for more data formats (including Vicon) and it is open-source, what makes it possible to modify Mokka to fit custom requirements. Vicon does not offer any plug-in system and it is impossibility to extend its functionality. Mokka on the other hand provides possibility to support new file formats for reading motion data. This might be useful, when custom format must be used, but as B-tk offers already a support for the most commonly used motion data formats, this it not so valuable feature. Team work and users data exchange is considered partially in Vicon as creating reports from loaded data and current browsing perspective. Mokka does not consider user data exchange at all despite simple motion data export. Although both projects are under heavy development, they offer very limited list of features supporting complete analysis procedure. None of them provides scripting capabilities. Only

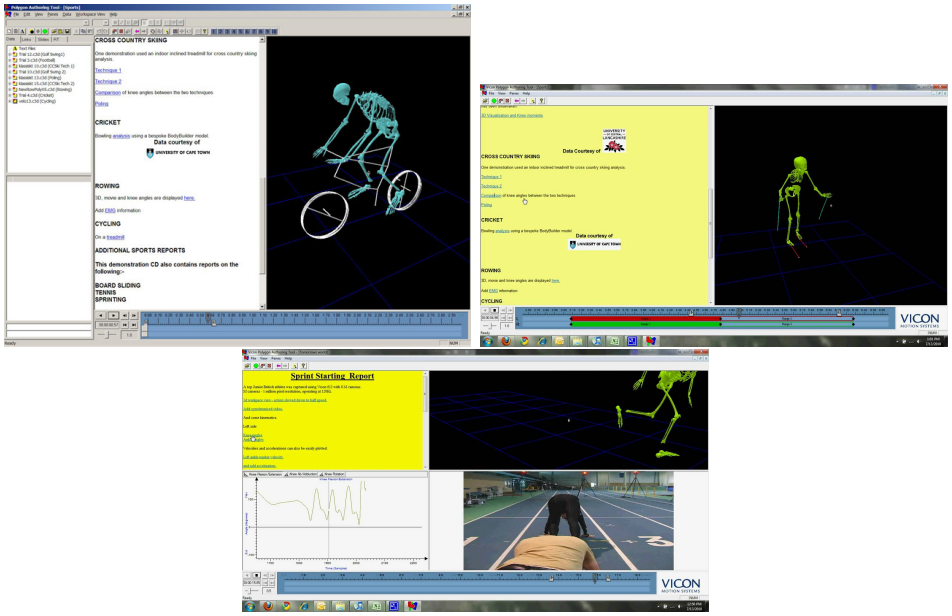


Figure 2.2: Vicon Polygon
sources:
`http://www.youtube.com/watch?v=TRYM8lCGFFU`
`http://www.youtube.com/watch?v=6g0mQpWkCdI`
`http://www.helmar-ms.pl/helmar-bis/biblioteka/polygon__helmar_vicon_motion-capture_video_system_mocap_przechwytywanie_ruchu01_lrg.jpg`

data files on local computer are handled, while non data streams and devices are supported by both applications. Both of them do not provide support for other languages than English. They provide limited support for customizing application appearance. Figure 2.2 and Figure 2.3 present screen-shots of Vicon Polygon and Mokka. For more information about those products please refer to provided producers web pages.

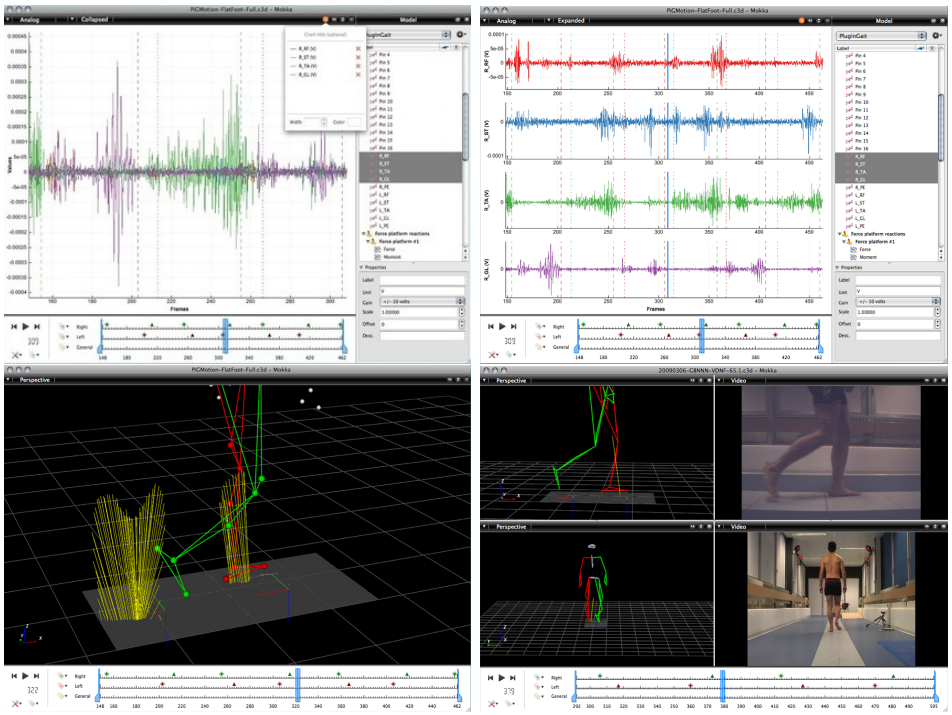


Figure 2.3: Mokka
`https://b-tk.googlecode.com/svn/web/mokka/index.html`

Chapter 3

Motion Data Editor

Motion Data Editor (MDE) is a software developed at PJWSTK since 2008. Application was supported by the Polish National Science Centre, grant agreements number:

- *NN 516475740*,
- *NN 518289240*.

MDE was designed firstly as a medical application, supporting orthopedist in motion data browsing. Since then it has undergone quite a long way of re-designing and re-factoring to become a mature, general purpose data processing and analysis tool.

The goal of this chapter is to present main MDE concepts and make software developers familiar with the MDE underlying data processing framework. We limit technical details to minimum, as they are implementation specific and not relevant for presenting main application features. For readers interested in MDE application public interface (API) we encourage them to look at the technical documentation and software development kit (SDK) available on attached CD. We also want to encourage potential MDE users to give it a try as a base platform for their research and advance data processing. We show, that application is built with very simple, clear and well decomposed components. They provide users basic, but reliable functionality for data management and processing. After this chapter it should be clear, how it is possible to utilize application capabilities to own research needs and how to extend it with users custom solutions through a simple plug-in system. Later, in Chapter 5, a simple example of an implementation for MDE data processing framework is given, presenting how to wrap custom solutions to fit this general framework.

Table 3.1: MDE features

Feature	**MDE**
Platforms	Cross-platform (Windows, Linux)
License	Educational
Video browsing	●
EMG browsing	●
GRF browsing	●
IMU browsing	● (custom IMU sensors and costume for motion acquisition)
Reports	●
Time line and events	●
2D charts	●
3D scenes and models	●
Data import formats	as B-tk
Data export formats	as B-tk
Custom user layouts	●
Motion database	● (dedicated plug-in)
Plug-ins	●
Data processing	●
Scripting	❍ / ● (partial, dedicated plug-in, in development)
Translations	●
Custom style sheets	❍ / ● (user interface (UI) still in re-design phase)

Similarly to previously presented tools, supporting motion data analysis, the MDE characteristic is presented in Table 3.1. In Section 3.2 grounds for presented features are given in form of system architecture and logic description.

Skipping most of implementation details, it has to be mentioned here, that application is written in C++ programming language, which provides high computations performance, when used correctly, but has also per-design limitations for uniform data management. We present a novel concept for general and efficient data storage in Section 3.2.1, that we have developed to address strong C++ variables typing making management of variety of data types difficult. Presented solution simplifies efficient data management significantly, what is one of the core tasks for data processing application.

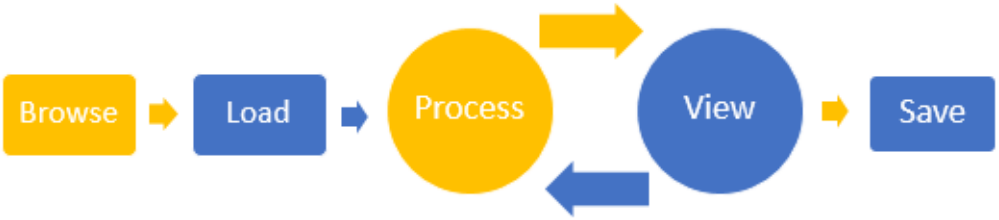


Figure 3.1: Common data processing pipeline

Before we describe in more details system logic and architecture, it is necessary to explain the process of data analysis, that is supported by MDE.

3.1 Data processing pipeline

Each data processing can be described as a sequence of specified set of operations repeated every time the data analysis is performed. It is possible to enumerate different processing stages, which are presented in Figure 6.2. Developed solutions are addressed to such a data processing scheme, decomposing it even more, to smaller, independent steps.

3.1.1 Browse

This step usually covers browsing local hard drive for particular files. In more complex cases it might require browsing Internet resources, FTP, querying database or web services, connecting to different devices. At this stage user is getting familiar with data available for further analysis. Some sources might require user authentication or additional settings (web page address, IP address, port number, transfer protocol) to gain access to requested data description.

3.1.2 Load

Data loading is a complex task. It covers data downloading (when required) from different sources, unpacking data and normalization procedure. It is also responsible for efficient data management.

3.1.2.1 Acquisition

Data can be delivered in different containers by various sources. Those are usually files or some kind of connections and streams. Usually, obtaining

data can be as simple as pointing out files on a local hard drive or more complicated, like downloading files from web pages or FTP. Different data formats and technologies are used to support data transfer and have to be handled to retrieve this data completely and securely.

3.1.2.2 Parse

This stage extracts essential data for analysis from previously obtained containers. Having connection (file path, buffer address, query result) with obtained data we have to unpack encapsulated data. For local files it can be simple XML parser, for connections with web cameras it might be conversion from stream representation (buffer) to a sequence of images.

3.1.2.3 Conversion

When specific data have been parsed to their custom formats, some standardization must be done to have ability to compare and analyse this data together. This is required as different sources can deliver logically data of the same type but in different representations. This is usually hard to find a common representation for a large amount of data types, but still they should be as uniform as possible. Special care should be given to data indexed with time, keeping in mind their time synchronization and resolutions. This stage is very important, as it delivers to the user data in their standardized format for further processing.

3.1.2.4 Management

Standardized data must be loaded to application for further processing. Efficient data querying should be ensured for fast data retrieval. It has to be noted that intelligent memory management is required, when large data, like videos, are used. Usually some centralized application data storage is provided to perform those tasks.

3.1.3 View

Visualization is required to present data to the users. They usually watch the data to have an idea about data structure, making their mind about further actions in analysis. Observing data allows also to compare data visually, when obtained numerical results are hard to interpret. It must be pointed out, that some data types can have various viewing perspectives, and users might require to go see all of them. As a simple example a 3D vector for a point position in time can be given. One perspective would be simple 2D

chart with all 3D position components plot independently, on the other hand 3D scene with point trajectory can be given as a curve or point movement in time can be animated.

3.1.4 Process

This is the core step in the whole data processing pipeline. User defines what operations are performed on particular data, providing optionally different sets of parameters to those operations. Data operations are chosen with respect to the data types they are going to be applied on. Input data is picked up from a set of loaded data. Based on basic operations users may create more complex algorithms, which they may want to save for further usage to limit analysis time and share those algorithms with other users.

3.1.5 Save

When processing has been finished, data must be saved for further analysis and possibly to share it with other users. Data serialization should be delivered allowing to store and load processing results. Although this step seems to be trivial, it can be hard to realize, as different data might require different storage formats. In general, creating one, universal format for all data types can be impossible. Also processed data serialization might be computationally very difficult task (i.e. video encoding, data compression).

3.2 System architecture

To address presented processing pipeline, a dedicated system architecture has been designed and developed. Proposing MDE logic we were following two main rules:

simplicity We wanted to decompose processing pipeline to fine grained, independent modules with very limited functionality and responsibility. Based on such components we wanted to propose simple, yet complete API for data processing. This allowed to create architecture with very limited dependencies between logic elements, making application maintenance and extension easy.

stability As described in [49], one of the most important software features is its API stability. Users prefer stable APIs more than continuously and rapidly changing ones, offering old functionalities in a new form, forcing users to upgrade their code with every release. Our goal was to

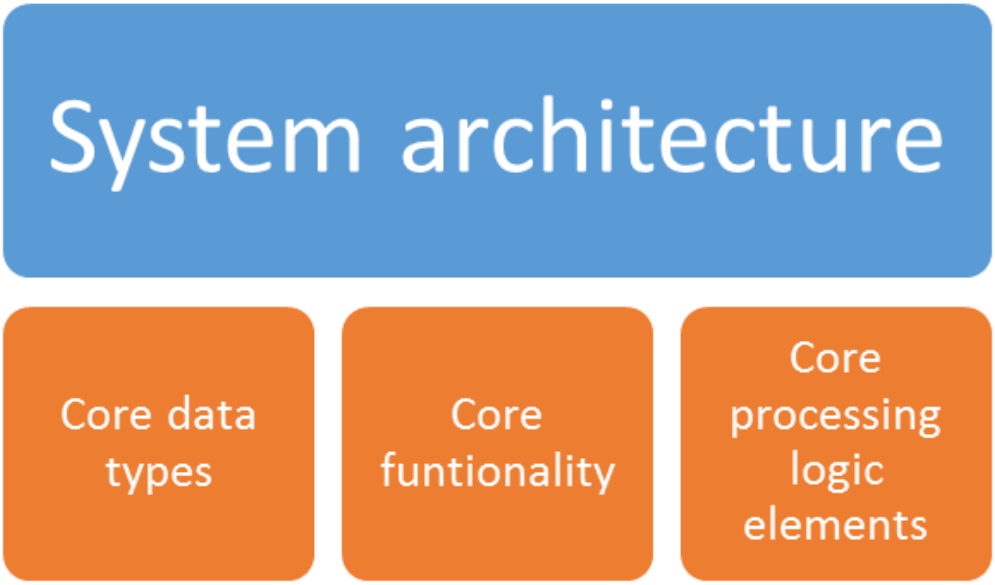


Figure 3.2: System architecture overview

provide since the beginning a stable and reliable API, not limiting users in any way in data processing and creating custom functionalities.

To achieve both those goals it was decided to provide very limited access to particular logic functionalities, only where it is strictly required, based on processing logic elements responsibilities and destination. There were proposed two levels of accessibility to core functionalities:

public globally available to any framework and client code,

private specific only for particular elements, which are eventually responsible for sharing them further only when required.

Such approach clearly defines what are particular logic elements responsibilities and capabilities. It keeps system logic simple with minimal number of rules, making it easy to familiarize with application architecture. This topic is elaborated in Section 3.2.4, where application core logic, organized with different managers, is presented. Figure 6.1 presents general architecture overview. Three components can be pointed out:

core data types basic MDE data types, offering fundamental functionalities like efficient memory management and standardized, generic time data representation,

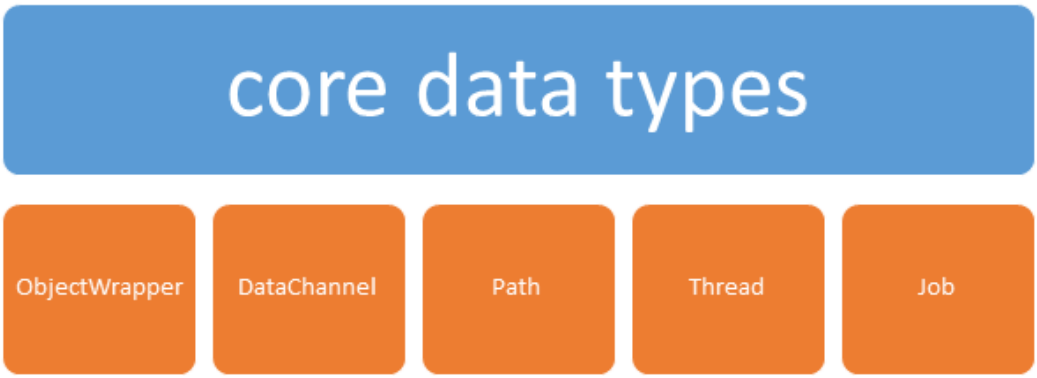


Figure 3.3: Core data types

core functionality provide abstraction layer for platform specific operations, they create base for data processing and other utility tools,

core processing logic elements objects in application responsible for different stages of data processing, joined together with simple rules create logical data pipeline and data flow in the system.

3.2.1 Core data types

Starting description of MDE in more details, presentation of the most important data types must be given first. Figure 6.3 presents five basic types. Now only two of them are presented, as they are crucial for data processing logic and whole system design is based on them. Other core data types are described later, when particular built-in, general purpose functionalities are presented.

The most important type in the whole application is ObjectWrapper (OW). It is developed to support unified data storage and management for any kind of data type in strongly typed C++ programming language. It might be thought, that encapsulation with help of simple template programming in combination with Real-Time Type Info (RTTI) might be enough, however we will show, that our solution has a greater potential. It follows a concept of the common base class for all types known from languages like *Java* or *C#*. Second core class, or rather group of classes, is dedicated to manage in a uniform way any kind of time indexed data.

3.2.1.1 ObjectWrapper

C++ programming language is strongly typed. During compilation static type checks are done. Based on a well defined Plain Old Data (POD) types it is possible to define custom types according to specified rules. Although a *void* pointer could be used to store any data, it looses information about data type. This makes impossible to access data, when more types are stored with such an approach. To address this problem a simple functionality is offered by *boost::variant* type and other similar template based constructions using *typeid* functionality. They however do not offer comfortable type information and data management, as only exact types can be extracted. Applying modern C++ programming techniques a novel solution to this problem has been developed. It is based on template programming [70] with behaviour policies [4], allowing flexible functionality customization at compile time for any types.

Main OW features are now pointed out, with their short descriptions:

type hierarchy information Provides information not only about wrapped data type, but also about its wrapped base types (currently only linear hierarchies are supported, for multi-base inheritance single base classes can be used)

meta-data Each data beside its raw values (scalars, images, ...) might contain some additional information in form of $[key \rightarrow value]$ string pairs (i.e. source of data in form of a source file path, web page address, label).

lazy initialization Object does not have to contain any data to provide complete information about its type. An initializer object can be given, which is used to deliver OW data. It is used when OW is requested to unpack wrapped data and it is empty (not unpacked and initialized already). Figure 3.5 presents general concept of lazy initialization procedure on data query.

easy data wrapping and unpacking Simple methods for storing and unpacking data are provided, with automated types conversion through wrapped class hierarchies.

customizable behaviour through policies Wrapped data types differ in two behaviours. This are: pointer policy, defining how data is stored in OW, and cloning policy, providing appropriate data copy functionality.

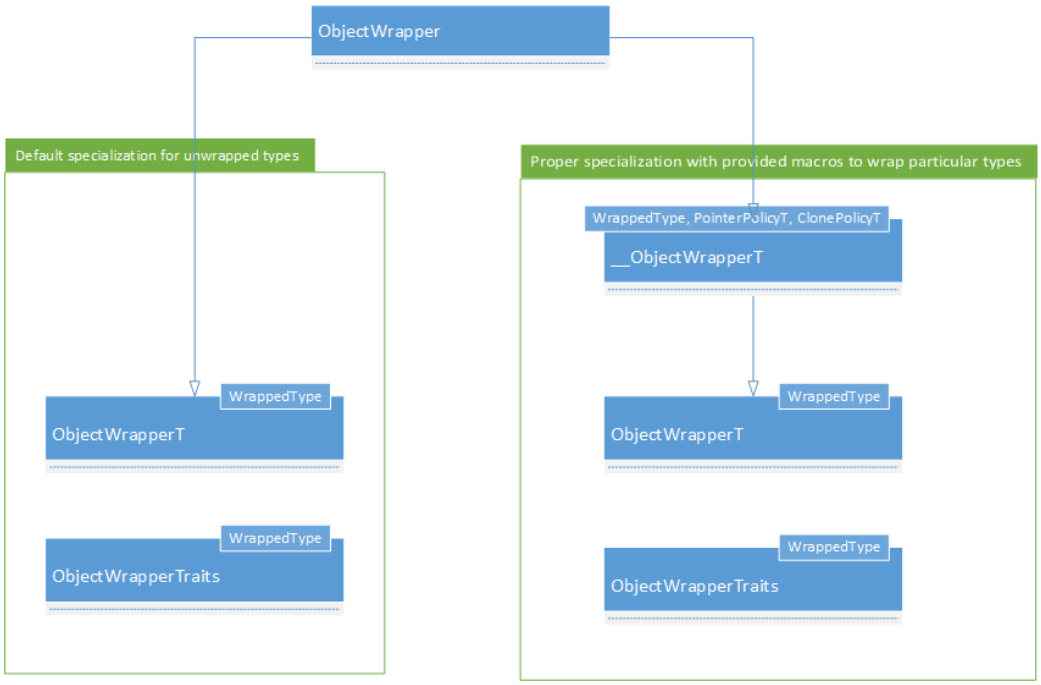


Figure 3.4: OW class hierarchy

compile time guards and traits At compile time verifications are made against proper usage of OW with different types. They allow to detect potential problems with usage of OW as soon as possible. Complementary checks are made on run-time.

Figure 3.4 presents OW class hierarchy. The main idea is to have a base class, providing common interface for accessing data type information and data itself, for any kind of properly wrapped data type. There are no limitations in data types, which can be handled by OW mechanism, making it possible to apply for both C++ built in types (PODs) and custom data types. Differences can occur in realization of particular operations and properties for those types, but they are covered with policies described in Table 3.2. Using OW data handling abstraction layer causes memory and data access overhead, but from data analysis and application design point of view, those penalties are completely acceptable, as there are much more valuable benefits from using so proposed data handling. Additionally, we have found out those overheads to be negligibly small in comparison to maintained data size and data operations complexity in real applications.

To make using OW simpler, a dedicated trait class is given, providing the most important information about particular, wrapped data types:

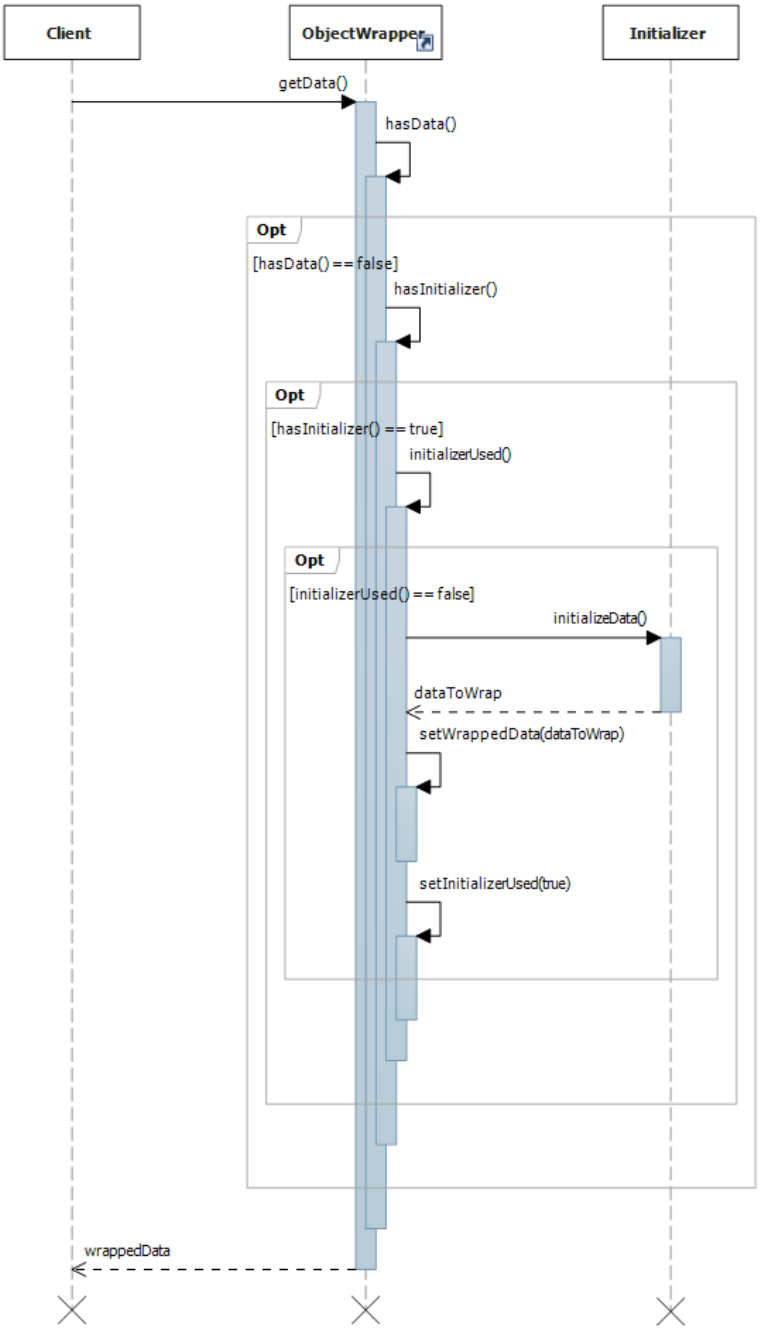


Figure 3.5: OW lazy initialization on data query

Table 3.2: OW policies

Policy	**Description**
Pointer policy	Defines what kind of pointer is used to store particular data type. In general two types of pointers can be used: raw, built-in pointers and smart pointers. Smart pointers might have two different schemes: intrusive and non-intrusive, where mixing them should never take place. Also mixing raw pointers with smart pointers is very dangerous operation. Depending on type implementation proper pointer policy must be used, fixed for particular type. It is suggested to use smart pointers policies, as they guarantee safe memory management.
Clone policy	Different types are design to offer their copy in a different way. Some use copy constructor, some have virtual *clone* method, some does not provide copying at all. This property is also specific and fixed for particular type. It is important for ensuring copying capabilities for OW, to make them behave similarly to the wrapped types.

- if type was wrapped with OW,
- what types of pointers are used to store the data,
- what clone policy does it use.

One disadvantage of such approach is, that whenever we want to use particular OW type traits their definition must be visible, therefore it might be required to include many headers files defining wrappers for particular types. This is however solved in the system architecture, so that type traits can be skipped, if some information about wrapped types is required. It is described in more details in Section 3.2.4.2, where data management functionality is presented.

Beside template implementation of OW functionality in class $\_\_ObjectWrapperT$ for properly wrapped types, a default implementation for any other type is given, preventing this mechanism usage with not properly wrapped types. It causes static assertions at compile time, when given type would be used with OW.

Sometimes, additionally to raw data some its more general description must be given. It is possible to create global mapping between different data and their description, but it would be affecting query performance, when more objects appear. To address this problem OW was designed to handle meta-data through a simple mechanism of storing strings in form of $[key \rightarrow value]$ pairs. Now, additional data description is strictly connected with the data itself within OW, with an immediate access to both of them.

In contradiction to simple implementations of *boost::variant* type, based on *typeid* functionality, OW allows to trace wrapped types hierarchies. There is a dedicated template specialization, where derived, wrapped types are extended with information about their wrapped base types. This allows to extract from the wrapped derived type its wrapped base types by automatic down-casting. In contradiction, for *boost::variant* type implementation exact type must be queried first and then downcast to the base type. Additionally, OW gives a possibility to query if particular type can be extracted from wrapped data. This is very useful when filtering for particular data type. Similar mechanism is used for assigning data to a wrapper. Tracing hierarchy information is one direction only feature - derived classes have information about their base classes, but base classes can not be extended with information about their derived classes in current implementation.

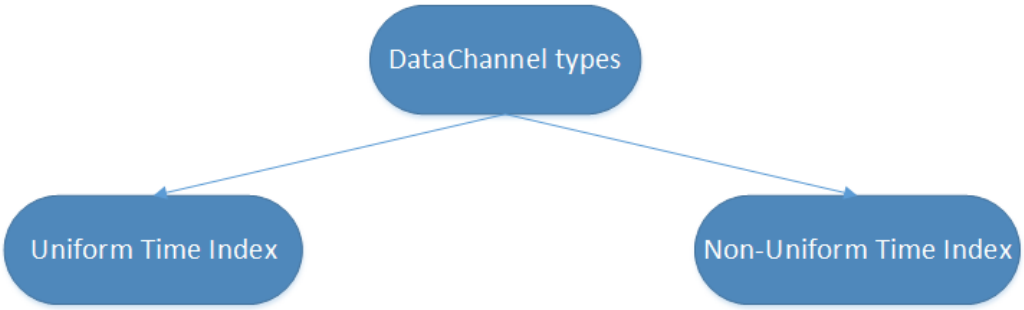


Figure 3.6: *DataChannel* types

For simple data query Return Type Resolver (RTR) idiom is used (`http://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Return_Type_Resolver`). Static types verification is done at compile time with methods call paths optimizations, based on the type traits. Analogically, run-time checks are performed, throwing exceptions, when necessary (i.e. when OW is still empty after initialization).

It has to be mention very clearly, that application implementation is completely based on OW mechanism. One drawback of this mechanism is that it is still based on *typeid* functionality, which for different compilers might provide different results. Using cross-compiled libraries or libraries built by different compilers might cause fatal errors. We, however, decided not to perform and support such builds and therefore this solution is perfectly functional and reliable under those terms.

3.2.1.2 DataChannel

Motivation for *DataChannel* development was to handle time indexed data in an uniform way. Its design addresses mainly time based physical measurements, but we wanted to provide general functionality for also other combinations defining custom indexing type and corresponding value type.

Figure 3.6 presents two types of *DataChannel* according to indexing values:

- with uniform indexes (constant distance between indexes),
- with irregular indexes (variable distance between indexes).

This property is used to provide efficient index based data access (constant for uniform indexes and O(log(N)) for irregular indexes). Data load

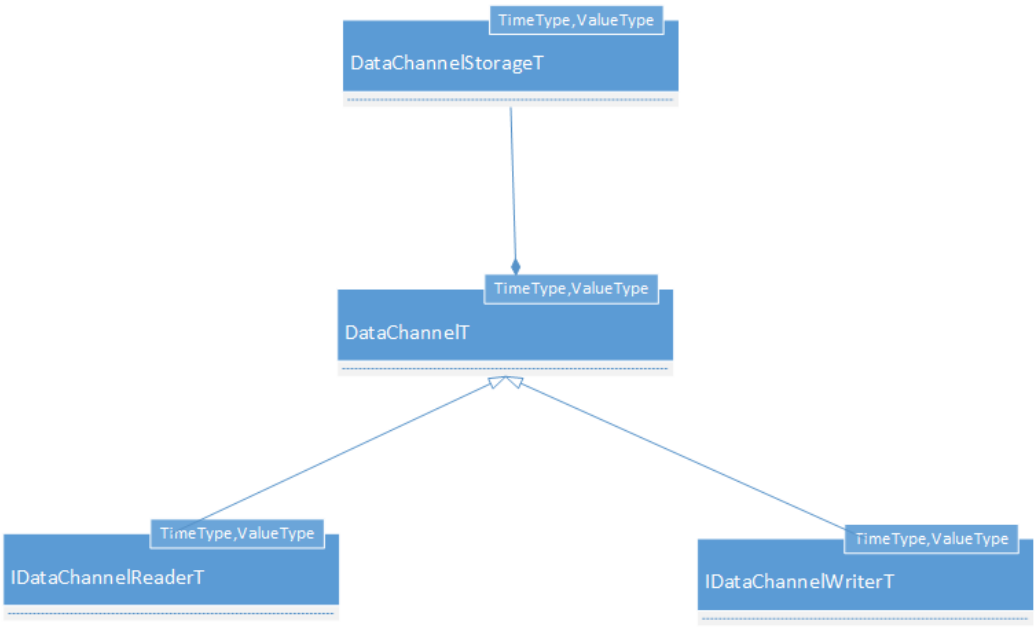


Figure 3.7: *DataChannel* concept

to *DataChannel* must be done in the ascending order for index values. *DataChannel* does not allow to modify index values, but gives full access to associated data values. Data can be accessed with various, intuitive methods:

- by indexing type values (time),
- by internal data sample indexes (samples indexing with non-negative integer values starting from 0 up to $number\_of\_samples - 1$)
- through iterators - Standard Template Library (STL) approach.

DataChannel mechanism is developed completely with templates using mix-ins of abstract interfaces and their proper functionality implementation. Figure 3.7 presents basic structure of *DataChannel*. There are two independent implementations covering both *DataChannel* types, based on common storage functionality - *DataChannelStorage* . Such approach allows to create easily adapters for reading data, making them behave as *DataChannels* with different data type without copying their content. Template approach allows automatic value passing optimizations based on the type traits to give *DataChannels* maximal efficiency. Table 3.3 provides complete description of *DataChannel* properties and requirements.

Table 3.3: *DataChannel* properties

Property	**Description**
Template based	Generic approach for any pair of index type and value type (some special functionality for index type is required). Strong compile time type check with value pass optimizations based on type traits.
Mix-ins implementations	Abstract interfaces for data access, separated from their implementation offer simple possibility for changing data representation (one 3D time based signal can be represented by three independent 1D time based signals operating on the same data).
Intuitive data accessors	Data access through internal integer samples indexes, defined index type value or iterators.
Ascending index type values	While data loading only ascending order of index type values allowed to ensure efficient data storage and access. This enforces index type to be comparable.
Unique index type values	Index type values must be unique. This property was decided to be introduced, as in most physical measurements it never happens, that for the given index (usually time) two different measurements are collected for particular observation.
Constant index type values	Only corresponding values can be modified, order of samples in channel must be unchanged.
State-less	*DataChannel* beside contained data values does not have any additional state.
Simple time queries	Methods for *DataChannel* samples duration and whole channel length according to indexing values are provided.

Data accessors Idea behind data accessors objects arise because of discrete data representation, when sometimes their continuous interpretation is required (i.e. when browsing data in a greater resolution than provided in raw data). Additionally, some algorithms might require querying for data outside of *DataChannel* (for integer and time index values range). *DataChannel*s provide only fixed number of samples and query for index values not present explicitly in data, although fitting its index value range, fails. Idea of data accessors is to provide abstraction layer, allowing to treat any kind of discrete *DataChannel* (although sometimes with index values with very high resolutions) as continuous. Data accessors can be provided for any *DataChannel*. Those objects behave like interpolation and extrapolation operators. For extrapolation several built in models are delivered:

exception Exception is thrown when querying for index value outside of the channel (default behaviour),

border Always first or last sample is returned depending on index value outside of *DataChannel* range,

periodic Index value is clamped to *DataChannel* range creating periodic signal impression.

For interpolation operator several implementations are provided:

piecewise constant interpolation the closest value is always chosen for given index,

linear linear interpolation is done between two closest covering values (difference and division operations must be provided for *DataChannel* value type),

polynomial missing value is interpolated with the given polynomial function (multiplication operation must be defined for *DataChannel* value type).

Timers *Timer*s introduce time state (for example during a playback) for *DataChannel* objects. This allows to save memory, when using the same data with different time modifications (offset, scale). This is very important feature of *DataChannel*, as some data types (videos, audio) might require great amount of memory. Further, it is possible to join time based data with *Timer* object to provide easy extraction of data from *DataChannel* according to the current time in *Timer* object.

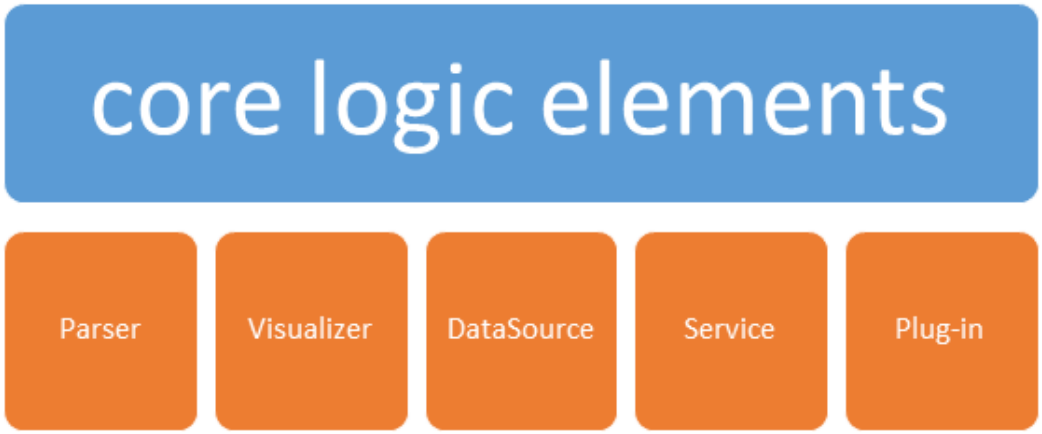


Figure 3.8: Core processing logic elements

3.2.2 Core processing logic components

Keeping in mind the idea of addressed data processing procedure and application fundamental data types, it is possible to present now the MDE logic. Figure 6.4 presents basic application processing architecture elements. Five fine grained elements, providing simple, independent data processing functionality were proposed as a base of MDE. Additionally, one of the most important application modules is a dedicated plug-in system. It allows MDE users to extend application functionality to fit their requirements, based on provided logic. Now description of core logic elements, loadable by plug-ins, is given.

3.2.2.1 Parser

Description The most important role for parsers is to unpack data from particular data containers and deliver them in an uniform format. This cover conversion of processed data and wrapping them with OW mechanism. Parsers must deliver:

- a description of supported data sources (files, Internet protocols, computer devices),
- possible to extract data types from supported containers.

Source of data is represented as a string, therefore regular expressions are used for source verification and description. As an example `"file://path"` pattern is used, similarly to Uniform Resource Locator (URL)s, for file paths

or `"hw://interface/deviceID"` pattern for local computer hardware devices. It has to be noticed, that different data types from the same source can be delivered by different parsers, supporting particular source. For example, some parser might provide only video from motion sensor for pattern recognition, while other would provide raw depth map for motion detection. As those two approaches are independent for various applications, probably non of them would implement the other functionality, because it is not required. However, both can work with the same source, extending its set of available data types in application. On the other hand, it would be probably a better solution to develop parser providing complete set of data types for particular data source, to keep its functionality complete and encapsulated in one place.

Although parsers have a common interface for querying unpacked data and supported sources, they can be divided into two categories based on their capabilities in processing particular data sources:

- stream based - considering STL data streams,
- custom I/O based - reading data source in a custom, specialized way, performing all low level operations on data.

Valid parser should support functionality of minimum one of those groups. In case of files as data sources, this gives the application opportunity to decide, which parser behaviour to use for particular data and context, if both functionalities are provided. This mechanism allows to increase application performance, by loading files once to memory and operate on them many times in form of streams, despite performing independent I/O operations on files by different parsers.

Logic accessibility and instancing In terms of processing logic elements accessibility, parsers do not require any additional access to core functionality beyond publicly available logic. Parsers hoverer must provide clone functionality, because when loaded through the plug-in system, they are treated as unmodifiable, prototype objects. They are created each time, when new data is loaded to application to unpack supported sources.

3.2.2.2 Visualizer

Description The goal of visualizers is to deliver a uniform way of presenting data to the users. As application support many different data types, viewing mechanism must also support this types diversity. Each visualizer

therefore provides a description of supported types it can handle. Presented data are organized in visualizers in form of data series objects. They offer simple description for shown data:

- name - descriptive name for the data, providing information what does it represent,
- data that is presented, wrapped with OW,
- desired data type representation, as particular data might support more perspectives based on their class hierarchy.

Visualizers might limit the amount of data series they can handle because of particular data type characteristic. Visualizers can support:

- single data series (for video playback, as playing two video streams on one scene is not a common approach for video players),
- finite number of series (to maintain scene view clarity),
- unlimited number of series (in the end limited by the range of integer types, but still it is hard to imagine more than 10 curves in one plot).

Visualizers introduce a concept of active data series, representing data currently managed by the user, when some additional operations for this data might be provided. Visualizers offer different operations for managing data series:

- addition (with respect to potential limitations in supported series number),
- removal,
- cloning (creating identical series from already created to manipulate its parameters and compare both of them),
- setting active series (turn on some additional functionalities for associated data, that visualizer might provide).

Visualizers are expected to provide functionality for making screen-shots of their current scene state. This is required for reporting mechanism, based on currently viewed data. This mechanism is described later in Section 3.2.4.7.

Despite such simple data management in form of data series, there is also an extended representation for data series considering time. When particular

data has time indexed representation and visualizer can handle it, then time aware series should be provided by visualizer for this data type. Such series in comparison to the basic series representation offer additional properties connected with time management:

- setting current time of the series,
- scaling the data in time,
- setting time offset to data.

This functionality is used for time indexed data time synchronization and manipulations for all visualizers in an unified manner with a dedicated *Timeline* service (Section 3.3.5).

Graphically, each visualizer delivers its own window, where it is presenting the data. Application only choose the place, where this window is initially placed, and from where users can start managing it. Such window might provide additional visualizer functionality, which is handled by visualizer, independently from the rest of the application logic responsible for presenting the data.

Logic accessibility and instancing Visualizers, similarly to parsers, do not require any additional access to the logic elements, as they provide simple, closed functionality supplied with all required data on run-time. They are also considered to be prototypes when loaded with plug-ins, therefore they must provide cloning functionality.

3.2.2.3 Services

Description By a service a new application functionality is considered, that users might introduce according their needs. As services can depend on each other, a simple mechanism of their initialization is provided. This is a two step operation, where firstly services are initialized independently, not knowing if other, required services were already initialized. In the second pass services are guaranteed, that all other services have already been initialized and they can start interacting with each other. Similar mechanism is used for finalizing services in application, but the order of operations is reversed - in the first step a service is asked to finalize its interaction with all other services and later it is finalized alone.

Services can provide three graphical windows to:

- present users their current state,
- allow users to control their behaviour,
- set-up their configuration.

Not all windows must be provided, only those the service can utilize. They are handled in a well defined manner in GUI, offering users well organized access to all application windows.

Logic accessibility and instancing As services provide new functionality, they have access to all logic functionalities. This is required, so they do not limit any potential features that users might want to introduce. This makes services the most privileged processing logic elements. They can provide access to complete application logic further for other objects, but they are responsible for offering minimal access to the private functionality, only when it is really required. Application was design to limit such situations, or ideally - they should not occur at all. Services do not need to provide cloning functionality, as they provide unique new features to application.

3.2.2.4 Data sources

Description Data sources are delivering new data to application. They offer users possibility to browse available data, provided by a supported storages, and load them to application. Data sources must provide information about data types that they can deliver. This feature is used for filtering, when user is interested in processing particular data type. Each data source is responsible for two operations:

- load - deliver data from various containers and load them to application,
- unload - remove data from application and potentially free containers resources (remove file i.e.).

Those operations should be realized in terms of provided system logic responsible for memory management and data loading, presented later in this chapter.

As sources may also cooperate together, they are using the same two stage initialization and finalization procedures as services. They can provide the same set of windows as services. Data source is a very general logic concept and an example of a data source can be found in Section 3.3.2.

Listing 3.1: Example plug-in definition with provided macros

```
CORE_PLUGIN_BEGIN("ExamplePlugin", ↩
    core::UID::GenerateUniqueID("{3C0CD7AF-9351-46CC-A5FE-52AA182E1279}"));
    CORE_PLUGIN_ADD_PARSER(ExampleParser);
    CORE_PLUGIN_ADD_VISUALIZER(ExampleVisualizer);
    CORE_PLUGIN_ADD_OBJECT_WRAPPER(ExampleDataType);
    CORE_PLUGIN_ADD_SERVICE(ExampleService);
CORE_PLUGIN_END;
```

Logic accessibility and instancing Similarly to services, data sources also do not have to provide cloning functionality and can deliver well defined windows. Because data sources are delivering new data to application in different form they have access to private application managers responsible for loading data (Section 3.2.4.2).

3.2.2.5 Plug-ins

Description Plug-ins are also considered to be the core processing logic elements, as they allow to deliver described so far core objects to application in a standardized way. They are used as containers allowing to load dynamically to MDE the following elements:

- data types - new data types supported by application,
- parsers - providing data from particular resources,
- visualizers - presentations capabilities for particular data types,
- services - allowing to extend application with custom functionalities,
- data sources - providing new data, that are managed by application.

Data types are represented as empty OWs. They are treated as prototypes and creating their new instances is done with cloning functionality of OW objects.

Plug-in objects are created with provided macros making it easy to construct plug-in with desired new features. They can be queried for particular object types. Each plug-in is uniquely identified with dedicated identifier. Plug-ins might have some additional description in form of a name and information, what functionality do they provide. They are used to present

users, what extensions are currently loaded. Plug-ins are compiled as independent shared libraries. By dedicated macros plug-ins are extended with maintenance informations, required for their proper management and verification (Section 3.2.4.6). Listing 3.1 presents example plug-in created with delivered macros. First line begins a new plug-in with the given name and unique identifier, based on sixteen characters literal string. Next, presented set of macros are used to embedded example parser, visualizer, service and data type to the plug-in to be registered in the application, when plug-in is loaded.

Logic accessibility and instancing Plug-ins are providing public MDE logic functionalities to all delivered elements. They are also initialized with a dedicated application context (Section 3.2.4.6), uniquely identifying plug-in and its components.

3.2.3 Core functionality

Figure 6.5 presents built-in system features, providing abstraction layer for common, platform specific operations. This are also various helper utilities supporting data processing logic. Beside described core data types and processing logic elements, this general purpose operations were developed to support efficient data management, data processing, data loading and conversion, application initialization and plug-ins handling.

3.2.3.1 File system

Path To unify local file paths representation and deliver common path operations, a dedicated data type *Path* was developed. Due to a different representations and conversions between various literal paths representations, we decided to standardize it in the application. *Path* provides simple decomposition of path to a list of the following directories and disks in their hierarchical order. It allows to query for a file name, extension, parent folder and the root folder for the given path. Additionally, path creation is intuitive and independent from file system - user does not have to worry about using slash (/) or backslash (\) characters, as simple operator for concatenating paths is given. In the end conversion to and from different literals is provided, verifying path correctness.

Operations As most application perform different file system operations, which are platform specific, we decided to uniform also this functionality.

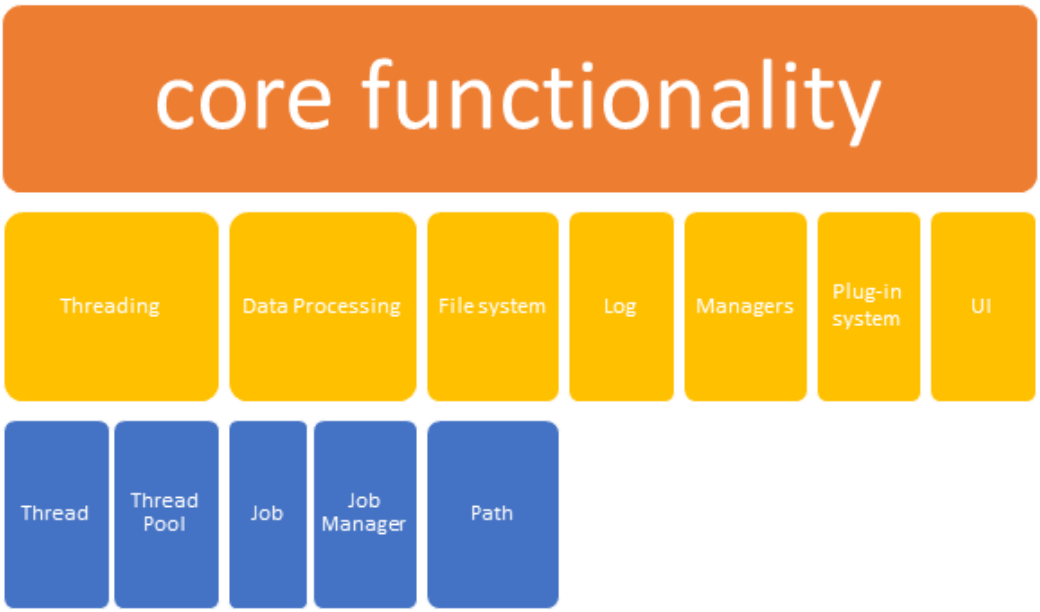


Figure 3.9: Core functionality

Dedicated abstraction layer is proposed, offering basic files and folders manipulation methods. Simple file and folder management is provided (create/delete) with advanced folders content listing and filtering. It is also allowed to query particular file system properties, as time of creation for file/folder, last modification or disk remaining space and capacity with respect to the current user rights and quota.

3.2.3.2 Application paths

Most of applications manage some well known file resources and paths. Despite unifying file paths representation and file system functionality, also the most important paths from application point of view are delivered in a compact way in form of a *ApplicationPaths* object. Table 3.4 presents all MDE specific paths.

3.2.3.3 Log

Since the beginning of MDE development, we found it very important to provide a simple to use, flexible and efficient logging mechanism. We proposed a dedicated *Log* that allows to present users application specific messages and information. We considered it also as a support for application debugging tool at runtime. Logged data can have different, independent destinations:

Table 3.4: Application paths

Path	**Description**
User data	Path to user data, where its results, settings and other data are stored. This path is relative to the currently logged system user on particular machine.
Application data	Private application space. MDE configurations and files requiring edition privileges are stored here.
User application data	Application private user space. MDE writes here user data, which are private for application and user does not have to know about their existence. Such approach guarantees user public space clarity and provides higher stability, as important data are secured from accidental deletion and modification.
Resources data	Application resources. Mainly resources for GUI: images, translations, additional static, read-only data.
Temporary data	Directory, where application temporary data can be stored. It is guaranteed, that it would be cleaned on MDE exit or failure (as much data is deleted as possible).
Plug-ins	Root directory, where user plug-ins are placed. This directory is browsed during application launch to provide additional plug-ins. This way user can easily manage custom plug-ins.

GUI, file, console. Our log system gives flexibility in choosing desired outputs for log messages. Moreover, logged information can have different information levels, where user can globally define minimal information level to be notified about. Informations with lower levels are automatically ignored. Logging system allows to create hierarchies of loggers. This provides greater clarity of information, as complete, detailed message context can be presented with detailed message source description. Figure 3.10 present different logger outputs presenting basic diagnostic data when starting application and loading data. Listing 3.2 presents messages logging for plug-ins with default plug-in logger.

Listing 3.2: Plug-in logging macros examples

```
PLUGIN_LOG_DEBUG("Plug-in debug message logged in default plug-in logger");
PLUGIN_LOG_ERROR("Plug-in error message logged in default plug-in logger");
PLUGIN_LOG_INFO("Plug-in information message logged in default plug-in ↩
    logger");
PLUGIN_LOG_WARNING("Plug-in warning message logged in default plug-in ↩
    logger");
```

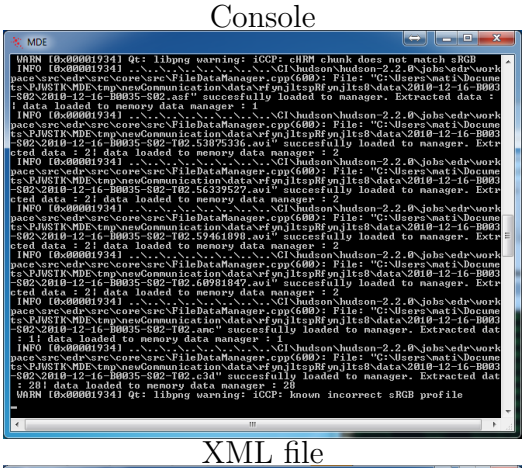
3.2.3.4 Threading

Efficient computational power utilization and management are the the most important aspects of data processing applications. On a modern hardware a processing unit can be represented with a traditional Central processing unit (CPU) or, more often this days, by a high-end Graphics processing unit (GPU), offering huge processing power with massive hardware parallelism. Nevertheless, as a CPU is organized in independent physical cores, which provide basic computation resources, their efficient utilization must be ensured. This is achieved by creating specific number of threads, usually equal to number of physical CPU cores multiplied by factor of two (current CPU producers declare, that with different technologies their hardware can process two or more threads simultaneously on one physical core). As we are providing cross-platform solution and threading is platform specific, we provided a dedicated abstraction layer for threading and synchronization.

Thread First step was to create a custom representation of a thread. We developed a *Thread* object allowing to:

- start,
- join,

Console



XML file

GUI

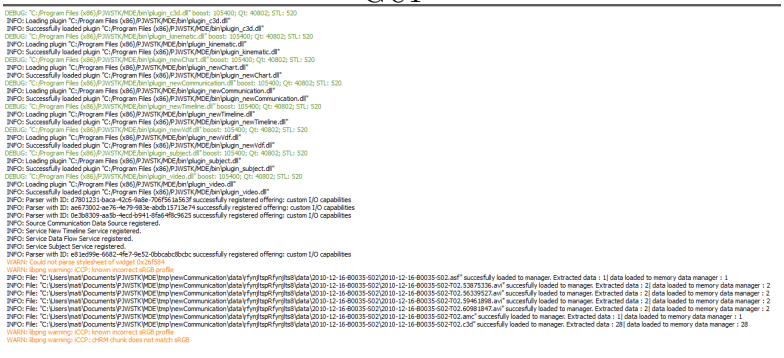


Figure 3.10: Various logger outputs

- detach,
- cancel

thread execution. Moreover, it allows to set the following thread properties:

- priority (operating system specific),
- stack size.

Computations performed in a *Thread* object must be encapsulated in form of *IRunnable* interface (abstract class in C++ nomenclature), which has a single method, taking no parameters, called *run*. This method is executed by newly created thread when it is started.

While writing this thesis, the new C++ standard, called C++11, already offers language native threading support, but not all compilers are offering its complete implementation and functionality. Therefore, presented custom solution has been proposed, but in the future it would be probably exchanged to language built-in features, when most compilers would support it. This would limit application external dependencies and necessity of maintaining additional functionality.

ThreadPool Standardization of thread representation at application level is just a first step in efficient thread management. Using too many threads may cause application to work slower in comparison to sequential code execution. This is caused by an overhead, that is connected with thread context switching on CPU and system level [75]. To address this problem, we have decided to control number of threads used in MDE by providing *ThreadPool* object. Its main roll is to deliver new threads when required and when specified maximum number of threads has not been reached yet. If limit is reached no more threads can be created until some already created threads will finish their work and free computing resources. Despite controlling maximum number of threads available in the application, *ThreadPool* preserves some minimum number of threads as a buffer, to eliminate necessity of creating new threads, as this system specific operation is usually time consuming. To save this computation time for data processing, threads are reused, when possible, to save as much resources as possible. Algorithm 3.1 and Algorithm 3.2 describe logic of acquiring new threads and releasing thread after it has finished its job.

Provided threads are assumed to perform simple operations and to sleep most of the time, handling some special events in the logic they are executing. Only some of them are used for data processing, utilizing maximum available computational power, but this is explained in the following sections.

Algorithm 3.1: Requesting ThreadPool for threads

```
Data: requestedThredsCount
Result: collection of threads
if requestedThredsCount + currentThreadsCount >=
maxThreadsCount then
    throw("Not enough thread resources");
else
    collection ret;
    int useFreeCount = min(requestedThredsCount,
    freeThreads.size());
    ret.insert(freeThreads.begin(), freeThreads.begin() +
    useFreeCount);
    freeThreads.erase(freeThreads.begin(), freeThreads.begin() +
    useFreeCount);
    int diff = requestedThredsCount - useFreeCount;
    for i=1 to diff do
        ret.push_back(threadFactory.getThread());
    end
    currentThreadsCount += requestedThredsCount;
    return ret;
end
```

Algorithm 3.2: Releasing unused thread

```
Data: releasedThread
if freeThreads.size() < minThreadsCount then
    freeThreads.push_back(thread);
end
currentThreadsCount -= 1;
```

3.2.3.5 Data processing

Efficient data processing is a separate and wide topic. It covers issues from low level code optimization for specific hardware, through numerical stability

of calculations, up to optimal problem decomposition into smaller, independent parts running possibly in parallel. As those optimizations are problem specific, we decided to provide easy to use, general tool to manage processing of already decomposed problem pieces. This is how concept of *Job* and *JobManager* was developed. Its main role is to manage available computational resources efficiently for processing smaller jobs, solving various problems.

Job *Job* is supposed to be reasonably small part of the problem, that can be processed independently from other *Jobs* solving this or other problems. Similarly to threads, reasonably small means, that overhead caused by such abstraction layer should be relatively small to the performed computations. Otherwise, poor performance can be met, as with too many threads. *Job* can be also thought to be a small, independent thread, with limited creation cost penalty. It must also provide a functionality of *IRunnable* interface, where whole processing must be done. *Job*s can be queried about their status of execution to check if they are:

- pending,
- being processed,
- finished.

It is necessary to provide such information, as *Job* delivered for processing need not to be processed immediately, as there might be not enough processing resources, when it was pushed to the *JobManager*.

JobManager As already mentioned, when describing threads and the thread pool, they are not considered for heavy computations explicitly. To handle such functionality *JobManager* has been developed. Its main role is to control processing of all delivered *Jobs*. It is responsible for collecting *Jobs* for further processing and scheduling them, as soon as processing resources are available.

Figure 3.11 gives and overview of *JobManager* functionality. For providing great efficiency *JobManager* is implemented in terms of threads and thread pool. Specific number of threads are automatically reserved for *JobManager* at application launch from the *ThreadPool*. Their number is equal to the maximal threads count CPU can handle effectively, where one of them is not realizing *Jobs* processing. This specific thread is responsible for monitoring other processing threads and restarts them, if they have failed or been

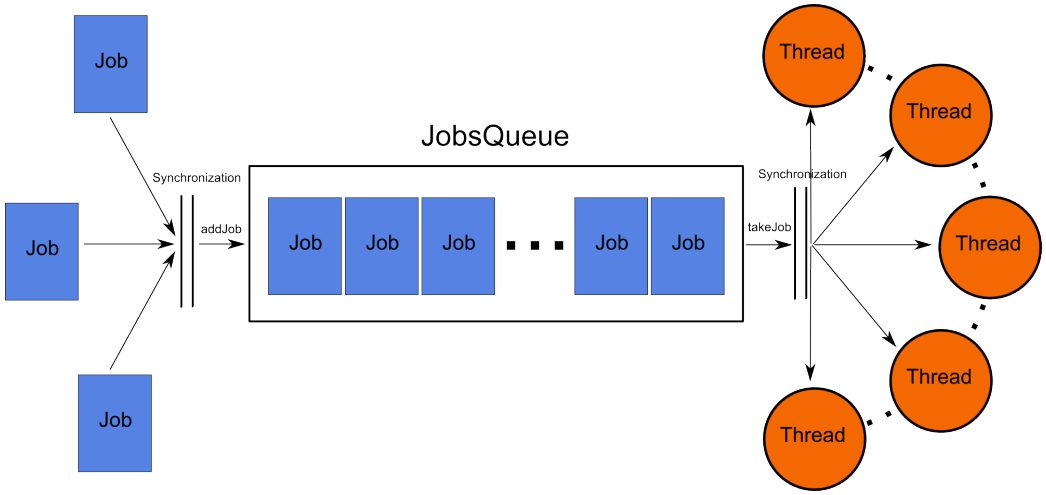


Figure 3.11: JobManager overview

terminated unexpectedly. Such approach leaves always resources for providing GUI minimal processing time and responsiveness. *JobManager* provides two main operations:

- public *addJob*, allowing to push new *Jobs* for processing,
- private *takeJob*, used by processing threads to query for next *Job* for processing.

Both those operations are synchronized, as *Jobs* might be added for processing from different threads and queue consistency must be guaranteed, when inserting and taking next *Jobs*.

3.2.4 Managers

Now it is explained, how particular logic elements are managed in MDE. Figure 3.12 present core managers, providing system functionality and support for data processing. Generally, for each presented architecture object a dedicated manager is designed, where some of them were already presented. Most of managers have their functionality divided in two separate functional groups:

- reading - mostly publicly available for other component, not modifying state of the manager
- writing - private, provided only to components requiring such functionality, modifying state of manager.

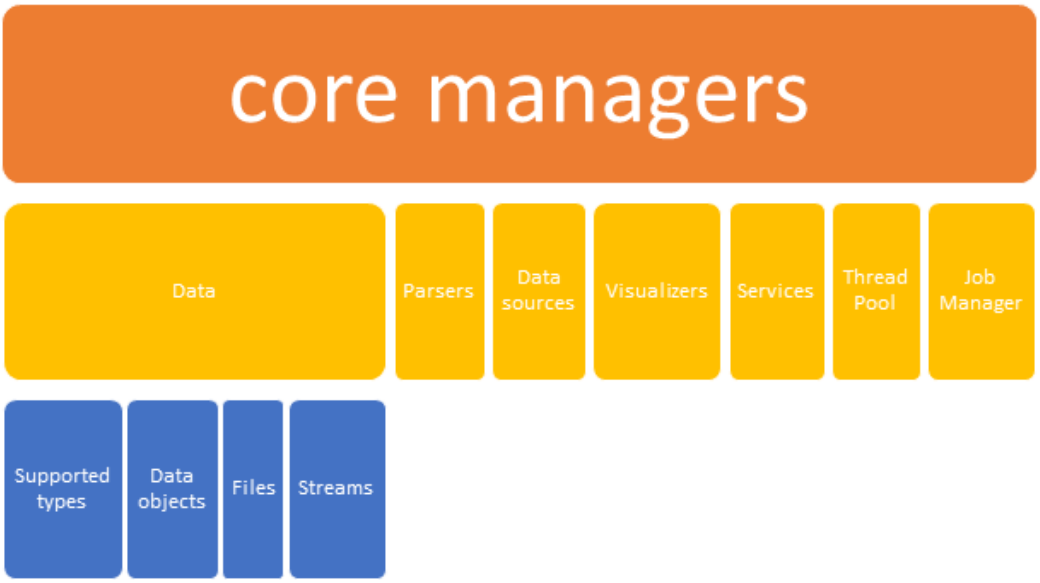


Figure 3.12: Core managers

This is the realization of discussed in Section 3.2 concept of providing particular core logic functionality, where it is required according to system architecture design.

3.2.4.1 Parsers

As parsers can deliver various data from different sources, their efficient management is required. Parsers manager offer functionality for registering parsers in application and querying for parsers providing particular data type. It allows querying for parsers supporting particular data source. Additionally, querying for information about all possible to extract types from particular source is possible. In the end, detailed queries can be made for parser supported functionality - if it supports stream parsing or custom I/O operations. This is useful to perform optimizations while data are initialized. Resulting parsers are always delivered as prototypes in unmodifiable form. They must be cloned to be used. Querying functionality is publicly available, but registration operations are hidden in private implementation for core application components use only, when plug-ins are read and unpacked and for other data specific managers, presented in the following sections.

3.2.4.2 Data

Smart data management is the key to efficient data processing application. This makes data managers one of the most important part of the MDE. As data in application might be loaded from different sources and dedicated parsers are used to extract the data, data managers are divided in four categories:

- types hierarchy manager,
- memory manager,
- files manager,
- streams manager.

Types hierarchy manager provides basic functionality for registering and querying supported data types, wrapped with *ObjetWrappers*. Memory data manager stores the data for processing in form of OW. It is MDE central database, providing efficient data access and management. File data manager offers functionality for simple loading files and extracting their data to memory data manager. Streams data manager provides analogical functionality to files data manager.

Types hierarchy This manager is responsible for registering supported by application data types, wrapped with OW. Moreover, on registration additional data is extracted for new types, investigating already registered types, to provide complete types hierarchy information based on OW functionality. This procedure is dedicated to the lack of such functionality in OW mechanism for types, for which derived types were registered. Figure 3.13 presents new data types registration procedure.

With help of types hierarchy manager it is possible check at runtime, if two types are in hierarchical relationship and what is the direction of this relationship without use of dynamic casting each time relationship is tested. Provided requests are based on OW mechanism and might be limited in contradiction to real class hierarchies because of provided OW hierarchy description. Types hierarchy manager functionality helps to organize and manage supported data types. All querying operations are public, while registering operations are reserved for private implementation, responsible for unpacking data types from plug-ins.

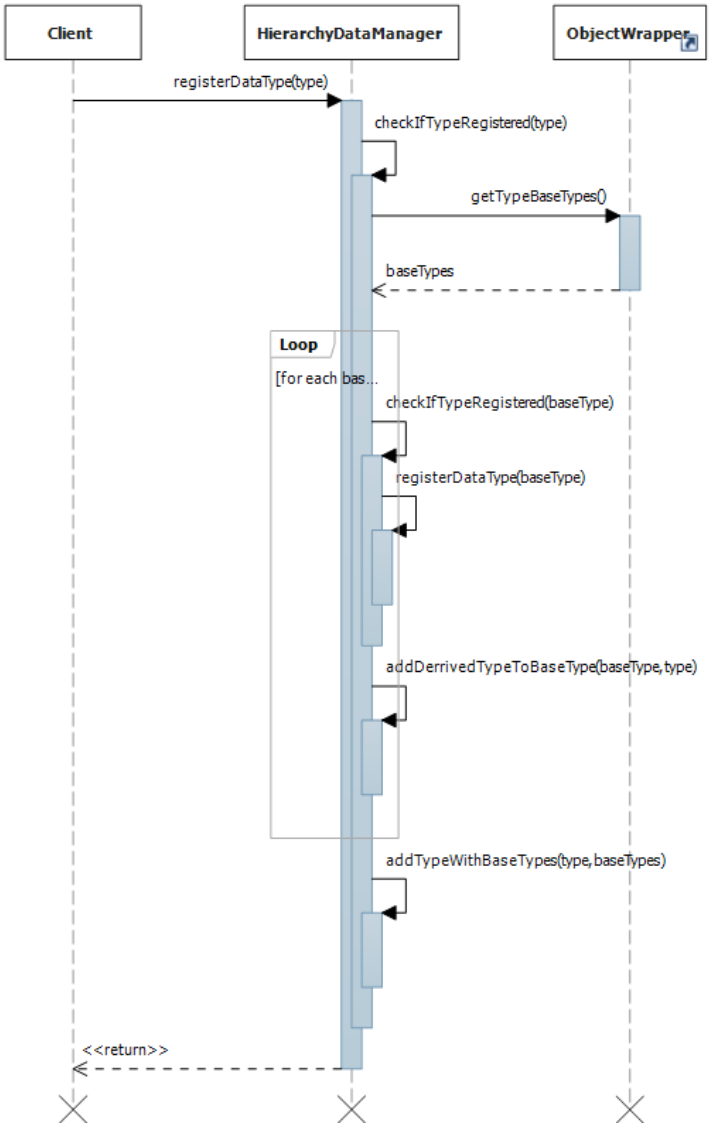


Figure 3.13: Registering new data type

Memory Memory data manager is a central data storage in the application. It stores data wrapped with OW. Provided functionality allows to:

- add and remove data,
- query, if data is managed by memory data manager,
- query for data of particular type,
- query for data supporting particular type.

As application supports multi-threading, all operations are guaranteed to be thread-safe. Moreover, when lots of queries and operations on data manager are done, transactions are introduce to limit synchronization overhead to minimum, providing isolation for all performed operations. Read transactions are always considered as correct and complete, but write transactions may fail. Transaction which caused failure is rolled back, returning memory data manager its initial state before the transaction has been started (adding and removing particular data modified during the transaction).

Data stored in memory data manager are delivered as read only data. They are treated as prototypes for further processing. It might be however necessary to modify such data, which is also possible through a dedicated functionality. Such editions should, however, occur very rarely, because of efficiency reasons - each edition is guaranteed to be thread-safe. Moreover, to allow capturing data edition by other application elements, and in general, monitoring memory data manager state, a notification system is introduced. It is possible to subscribe to memory data manager for it state changes, but it has to be ensured, that no edition is done from within the notification, as this would cause application to break (recursive, infinite notification). Very frequent data editions would cause many notifications to be emitted, what could drastically lower overall application performance.

Data querying functionality is offered publicly, but data management is available only to *Services* and *DataSources*. Also private implementation of other data managers have access to those operations, what is presented in next two paragraphs.

Files As most of the data comes from different types of files stored on local hard disk, we found it necessary to provide mechanism for simple file loading, their data extraction and management. For files manager operations of files addition, removal and querying are possible. Procedures of addition

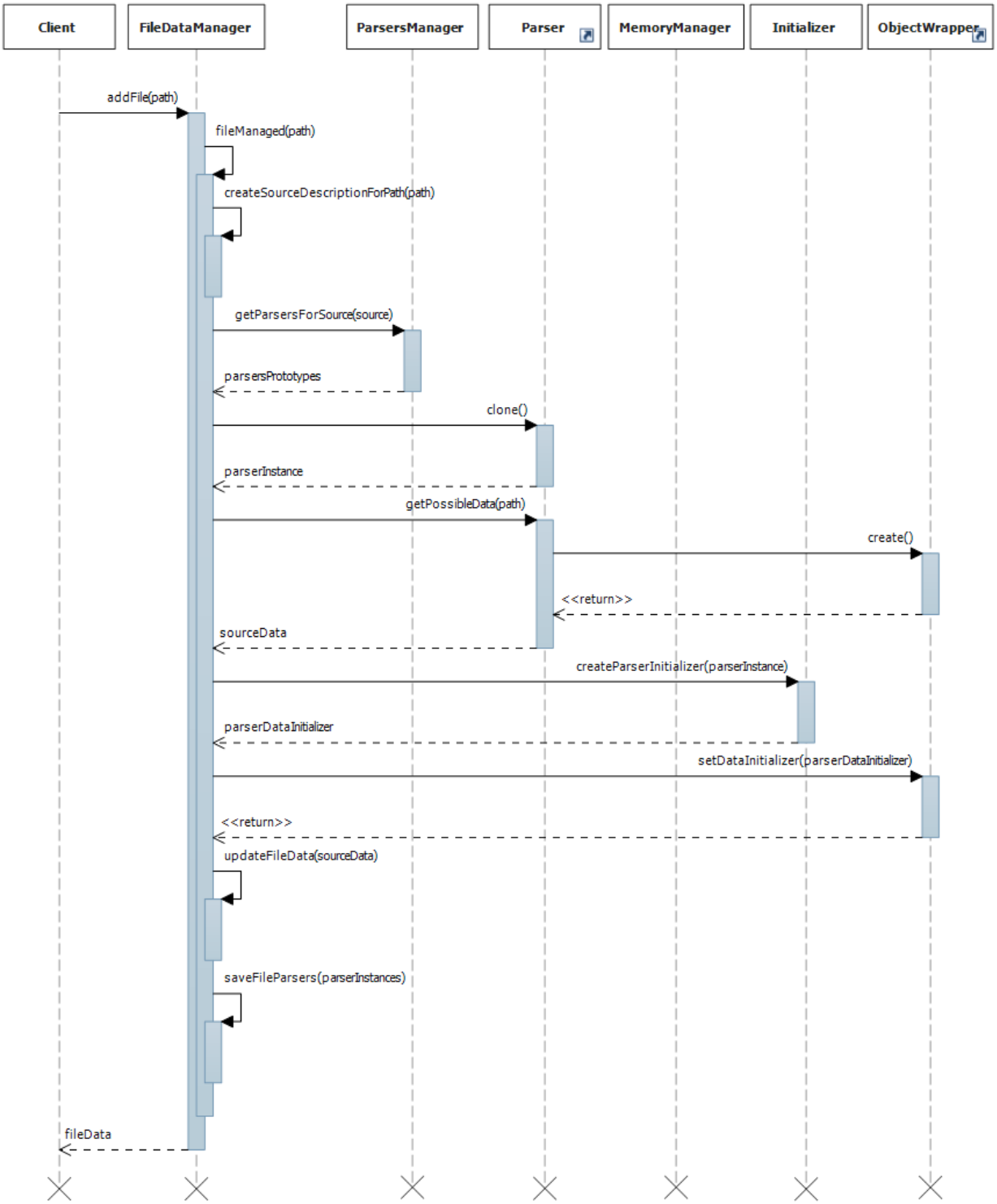


Figure 3.14: Extracting data from file

and deletion are strongly connected with parsers, where parsers manager is extensively used.

File addition procedure overview is presented in Figure 3.14. Firstly, simple check is done, if particular file was already managed, as there is no need for any file to be processed more than once by file manager. If file is not already managed, parsers are collected, that can possibly extract data from this file. It has to be recall, that parsers can provide different files handling through streams and custom I/O. File data manager always prefers stream access over custom I/O, where file read optimization can be done. Also, if parser supports both file handling techniques, only one of them is used, as they both should provide the same set of possible data types. Having all parsers that can potentially extract data from the file, they are queried for uninitialized data for this file. This is part of data lazy initialization procedure, where we do not provide data immediately, only when user explicitly query OW for encapsulated data. Empty OWs are supplied with dedicated initializers, causing data parsing on data query, when data is not available yet. Such data might be released later to save memory, if not used. Next query will cause initializer to launch once again initialization procedure. Figure 3.15 presents data lazy initialization with parsers.

Question arise, what happens, when parser can not deliver particular data type from the parsed file. OW, which should deliver this data returns empty pointer (comparable to null pointer) on extraction. It would not cause any further data parsing and it would be removed from both memory and files manager, as not providing any data.

File removal causes deletion of all connected with file data from memory manager. Next, all associated parsers are removed and destroyed and file is removed from file manager. Similarly to memory manager, all operations are guaranteed to be thread-safe. Also similar transactional system is provided with notification mechanism. It is possible to observe changes in managed files. Additionally, file data manager traces management of delivered to memory manager data and response on them. In specific, it removes loaded files, if all their data have been removed from memory manager. Such file is treated as unused, as no references to it exist any more. It must be added to files manager once again if we want to use its data.

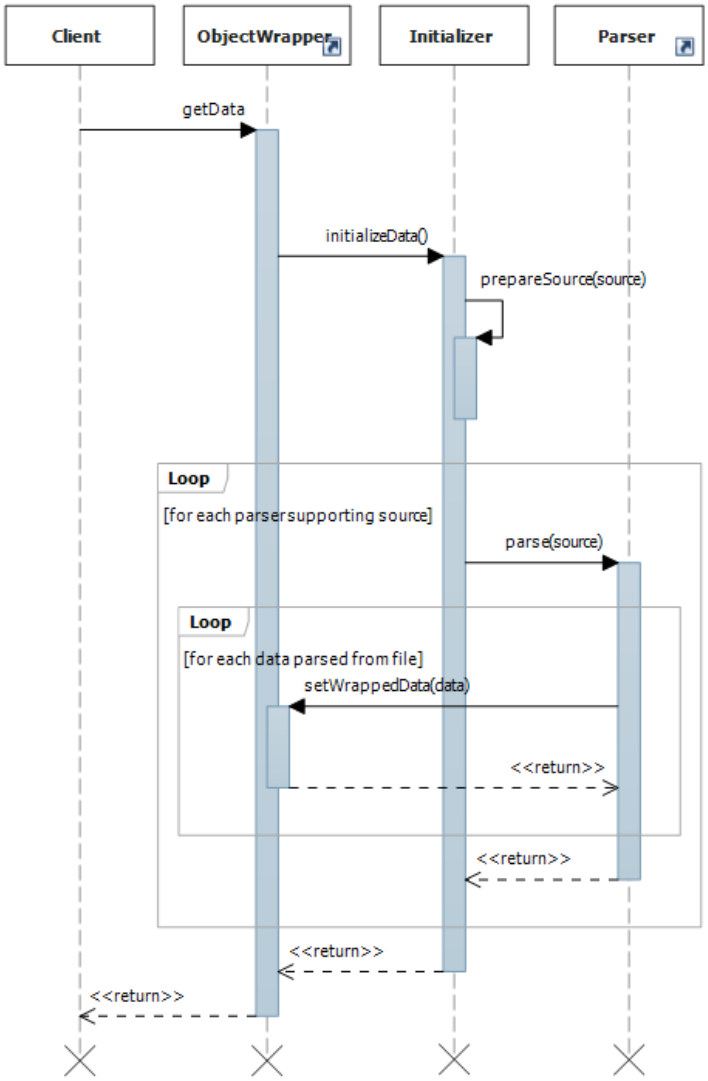


Figure 3.15: File data lazy initialization

3.2.4.3 Visualizers

Browsing various data types in different perspectives can be difficult to handle. Visualizers manager is design to support this type of operation. Similarly to parsers manager, it is possible to query for visualizer handling particular data type. To support GUI functionality, methods for observing visualizer manager are provided. Notification mechanism was proposed, notifying subscribers each time new visualizer is created or destroyed. This functionality is useful for finding visualizers with particular data type. Visualizer manager always return visualizers prototypes. Thread-safety of all operations is guaranteed. During notifications it is impossible to create or destroy new visualizers. Last, but very important functionality realized by visualizers manager is updating all visualizers instances to refresh their scenes according to a defined constant time intervals. This behaviour is required by some visualizers, not having their own source of time. Visualizers with custom source of time and scene refresh mechanism can ignore this update calls.

Visualizers registration functionality is private, available only for plug-in system. All querying methods are publicly accessible.

3.2.4.4 Service

As services are designed to provide new functionality to the application it is hard to define any common behaviour for theme. Therefore, their manager is limited to very few and simple operations. First of all, service manager can register new services in the application. Secondly, it is possible to query for services. Querying can be done in two different ways:

- query by index, according to number of registered services,
- query for particular service by its type (interface).

In the second approach, each service is tested against support of particular type with dynamic casting and returned, if such casting is successful. If there are more services supporting particular type, querying by index must be done to obtain their instances, as query by type returns first service matching the given type. In contrast to visualizers or parsers, returned results contain requested services instances instead of their prototypes, as they are treated to be unique, standalone functionalities in application. Once again, service registration is a private operation, used for plug-in unpacking, and querying is publicly available.

3.2.4.5 Data source

Sources are very similar to the services, therefore their manager is behaving in a similar way as manager for services. For additional information please refer to the previous paragraph.

3.2.4.6 Plug-ins

Plug-ins manager control process of loading plug-ins to application. It is a fundamental MDE element, as plug-in system is thought to be one if its most valuable features, giving users flexibility in extending application with their own solutions.

Figure 3.16 presents procedure for loading plug-ins. Plug-ins in MDE are thought to be dynamic libraries providing well defined methods for application extensions extraction and plug-in verification. Verification of plug-ins is crucial for application stability, as external libraries might be built against different libraries than application itself. Exchanging data of the same type, but with different application binary interface (ABI) may quickly cause application to be unstable and crash, as stack objects might have different sizes. To address this problem, plug-ins are automatically extended with information about their build type (debug or release) and used libraries versions. During plug-in load procedure those values are compared with the same information embedded in the application. Plug-ins are processed further only if this data match.

When plug-in dependent libraries verification is successful, plug-in is tested against its interface compatibility with application. Whenever MDE public API changes, a value describing public interface version is increased. To ensure that provided plug-in interface is compatible with the same interface at application side those values are compared. Plug-in can be loaded only when both interfaces versions are compatible.

Plug-in unpacking is based on looking for a well defined exported function, which creates plug-in defined class wrapper for all new functionalities. When expected entry point for plug-in is not found, such library is skipped from further processing. Loaded plug-ins can be queried further for unpacking core processing logic elements and register them in application within dedicated managers. To make delivered elements easier to identify plug-ins must have their unique identifier and optionally a name with a short description. During initialization, plug-in is supplied with required data to work. This is described in the next paragraph about application context. Plug-in

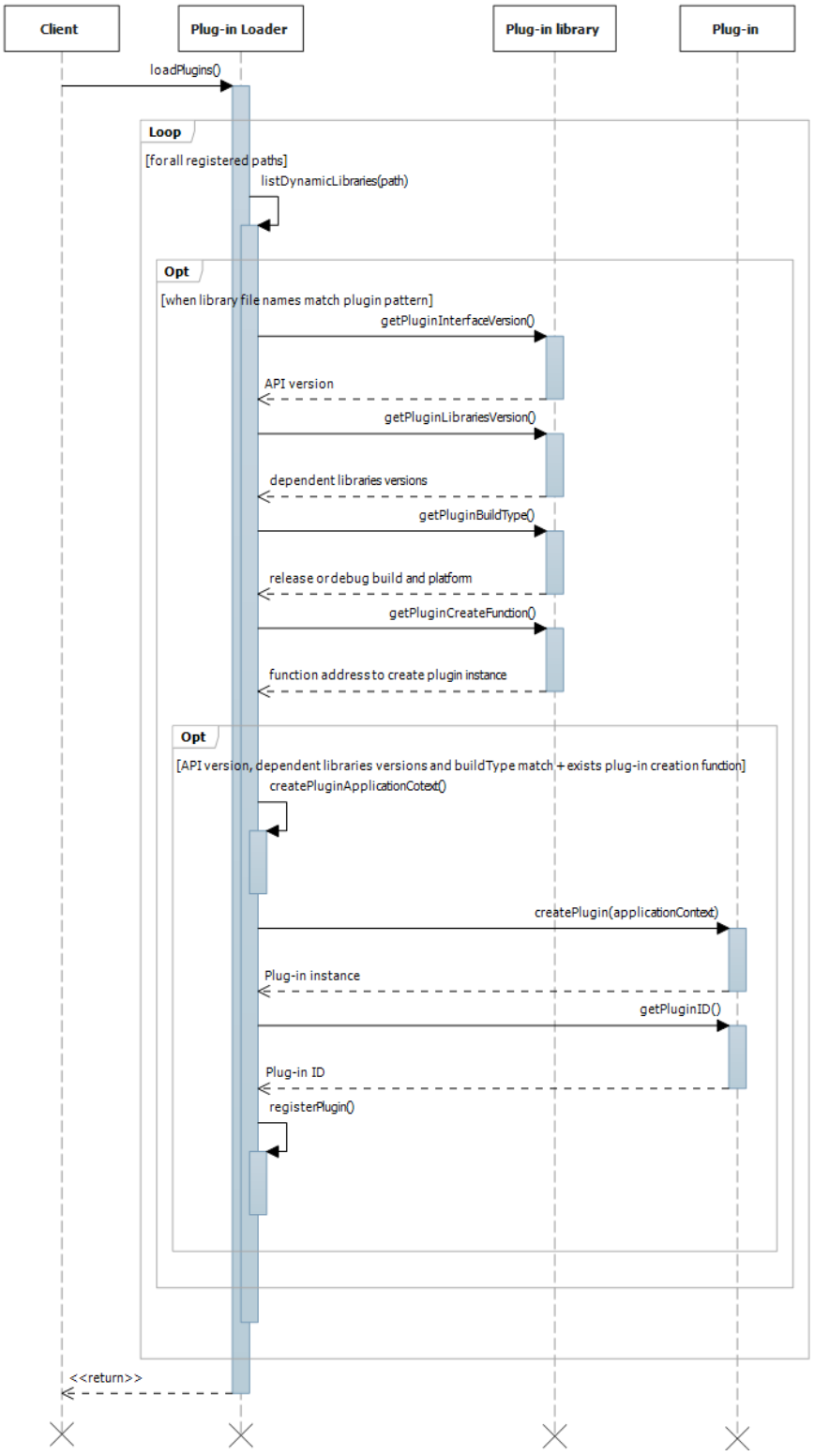


Figure 3.16: Loading plug-ins

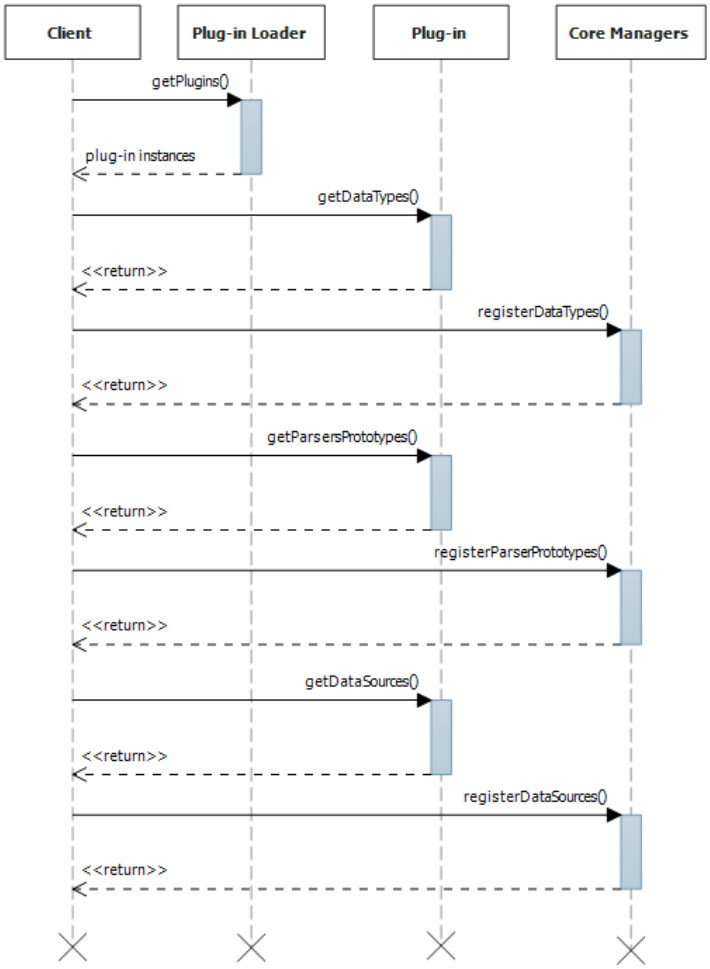


Figure 3.17: Unpacking and registering plug-in content

extraction procedure is presented in Figure 3.17. For clarity, only new data types, parsers and data sources are presented, as visualizers and services are handled in the same way.

ApplicationContext Application context provides description of current and logically complete application state. It delivers all so far defined managers and functionality with application paths to initialize plug-ins. This information must be provided at plug-in level, so that log system, all managers and other core functionality are available to elements delivered with plug-ins. Each plug-in obtains its custom *ApplicationContext*, supplied with dedicated logger and customized paths. Delivered logger is already extended with plug-in information (name, identifier, description), so it is clear, which

plug-in is providing particular log entries, even if different plug-ins have the same internal loggers hierarchies. Each plug-in has its own folder for resources and other data, where it has full read/write rights. It has to be noted, that delivered managers in *ApplicationContext* are publicly available at any place in a plug-in and, according to presented design rules, most of them are limited in their functionality for read only operations. Write functionalities are delivered only to elements strictly requiring them according to already presented architecture design rules.

3.2.4.7 UI

As application is designed to provide simple data processing functionality, it must also provide intuitive GUI for users. For this reason different UI utilities classes are provided, from basic docking windows, through their managers (organizing windows in logical groups), dedicated console window (presented when different forms of message logging were discussed), simple actions, buttons, tool bars with many other useful GUI functionalities standardizing UI creation and development. We want to present here only one specific GUI class - text editor, which is a core part of proposed reporting functionality, as UI layer is being currently re-design to offer general purpose mechanism for managing GUI, according to simple rules.

CoreTextEditWidget and reporting mechanism As MDE was firstly design to support orthopaedics in browsing patients data, it was clear, that reporting functionality would be very useful. It should allow medics to create custom documents with fragments of viewed data and attached comments to exchange knowledge with other colleagues or save their current work state for further analysis. This is how a concept of standalone, rich text editor have been developed. Figure 3.18 presents example editor window with a short report based on viewed data.

Provided editor offers most of the functionality available in classical text editing tools. Despite text, users can extend documents with screen-shots from visualizers. Text can be easily formatted - size, colour, spacing, font, style all customizable. Moreover, users can create hierarchies of paragraphs with enumerations. For reports predefined document layouts can be used in form of document templates. Users can also apply custom style sheets to documents, automatically formatting text. Documents can be printed or saved directly from editor. Two main export formats are supported - PDF or ODF, as they provide good document portability and for ODF also further

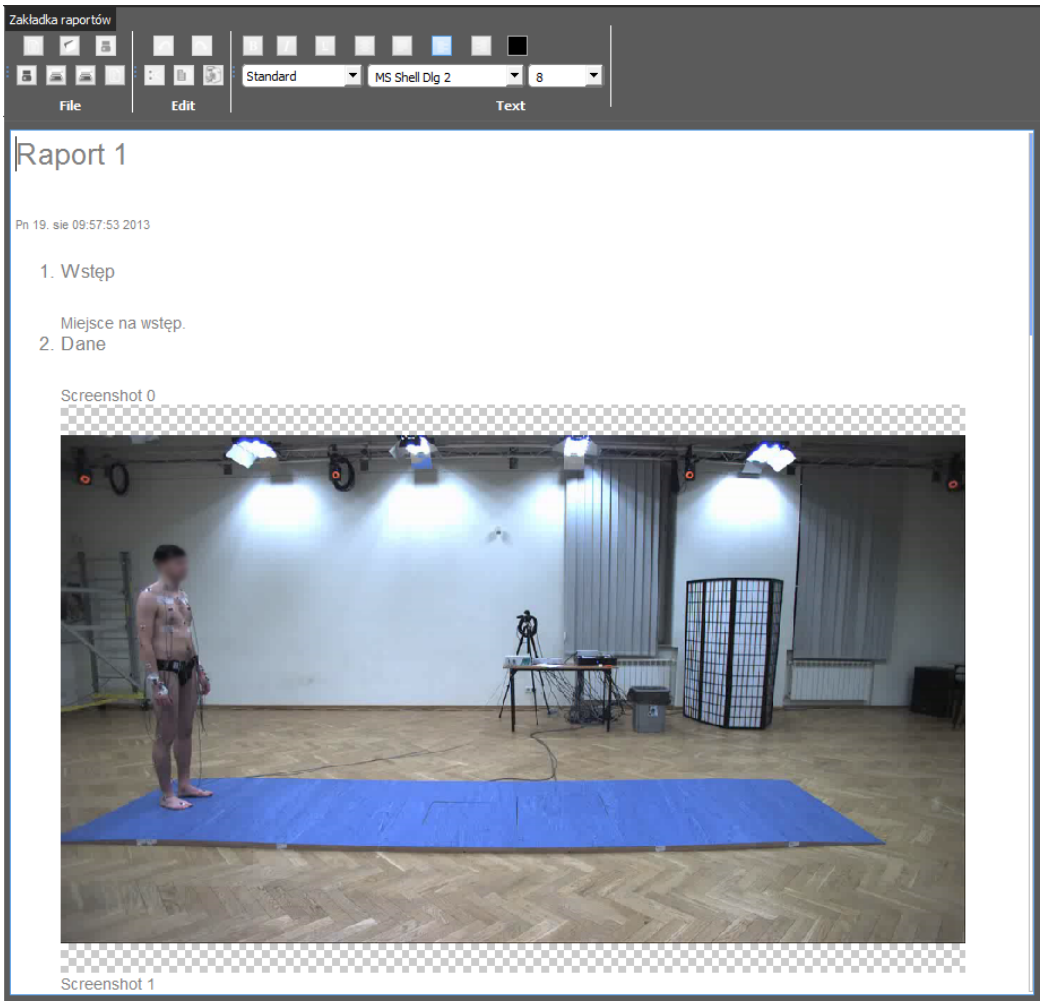


Figure 3.18: Text editor and reporting

edition in other supporting it tools.

Reporting mechanism in MDE is very simple. User browsing the data in visualizers create their screen-shots. They are captured in a dedicated area, presenting thumbnails of so far saved visualizers scenes. User can manage them if required - in particular remove them, when they are no more needed. When user collects all required images for a new report, he simply chose which report template to instantiate, or just creates an empty document. Now he can load images directly to the report and extend it with additional text. This utility allows simple work state description exchange between users.

3.2.5 Code organization

Many different data types and functionalities were introduced so far. Last step step to familiarize developers with provided solutions is to present, how those concepts are organized across different libraries. Generally, four groups of application logic can be determined:

- private core logic (implementation),
- public functionalities and API (abstract interfaces, utility classes and general purpose functionalities),
- UI utilities,
- application views.

3.2.5.1 corelib

This dynamic library provides all public MDE API and some general purpose functionalities. Interfaces for all enumerated logic elements are given. List of all classes is presented in Table 3.5. Most of class names are consistent with presented system architecture concepts.

Table 3.5: corelib exported classes

Classes and Interfaces	Description
ObjectWrapper	Unified data storage and management
Path, File system and IPaths	Complete file system management with application paths access
IApplication	Interface for ApplicationContext providing all publicly accessible logic elements
I{X}Manager with IThreadPool	Group of interfaces for accessing different managers and data processing logic
IJob, IThread and IRunnable	Interfaces for processing logic elements
ILog	Interface for message logging system
ITransaction	Abstraction layer for different transactions provided by managers
IParser	Interface for delivered parsers
IDataSource	Interface for delivered data sources
IService	Interface for delivered services
IVisualizer	Interface for delivered visualizers
Visualizer	Wrapper for IVisualizer providing basic data series functionality implementation
Other	Interfaces for common functionality responsible for example for data serialization, description and identification

3.2.5.2 coreui

This dynamic library contains dedicated, general purpose tools for MDE UI creation. Specialized classes for managing user actions, data presentation and organization of available operations in a simple, intuitive way are given. Table 3.6 presents complete class list exported in *coreui* library. We are skipping UI design description, as it is still developed. It is worth mentioning that UI is based on the *Qt* framework as it offers multi-platform support, also mobile devices with touch screens, which we consider to support in the near future.

Table 3.6: coreui exported classes

Classes and Interfaces	Description
CoreAction	Helper class for delivering windows custom operations to manage them in an uniform way, usually represented by different buttons
CoreWidgetAction	Similar to CoreAction, but custom widget can be provided instead of action, text and icon
ICoreActionSection	Additional interface for actions providing information about actions organization within various actions groups
CoreTitleBar	Custom title-bar implementation with actions on the left and the right side. It can be used vertically and horizontally
CoreConsoleWidget	Implementation for widget presenting logger messages. Messages are styled depending on their type
CoreTextEditWidget	Widget allowing custom documents creation and text editions. Supports styles and templates. Base for reporting functionality
CoreDockWidget	Implementation for widget that user can grab and position in application main window according to his needs
CoreDockWidgetSet	Widget for many CoreDockWidgets. Limits their maximal number to maintain clear view
CoreDockWidgetManager	Manager for groups of CoreDockWidgets. Creates new groups and stack them if current groups do not have any free space left
CoreFlexiToolBar	Implementation for flexible toolbar where actions can be grouped together. Groups organized in sections can be moved around toolbar
CoreFlexiToolBarSection	Implementation for sections managing groups of action in FlexiToolBar
CoreSplittableDockWidget	Implementation for widget having capability to split their content either by cloning or other specific behaviour

Continued on next page

Table 3.6 – *Continued from previous page*

Classes and Interfaces	Description
CoreVisualizerWidget	Dedicated class for managing Visualizer view. Presents all Visualizer actions in attached CoreTitleBar in an organized manner or in application CoreFlexiToolBar, depending on usage context
CoreCompoundVisualizerWidget	Widget managing many visualizers allowing to switch between them and reloading their context and actions
CoreMainWindow	Base class for any new view handling processing framework. Instance is initialized with *AppInitializer* on start-up

3.2.5.3 corelib

This is a static library, where private implementation for most of the core functionality can be found. All managers are instantiated from within this library with complete core logic initialization. Processing framework is controlled through dedicated *Application* class and its initializer with a template method for starting different application views. Initial *ApplicationContext* is created here at start-up. This static library is dedicated to be compiled with different, higher level processing logic with dedicated views, presented in the next section. For clients only *AppInitializer* class is exported to initialize data processing framework. Starting logic with a new view is as simple as presented in the Listing 3.3, where *CustomMainWindowClass* represents class derived from *CoreMainWindow* realizing custom UI. It has to be mentioned, that this library must not be linked with any other libraries, except new views (executables), as this would introduce ambiguities, which compiled-in set of managers instances should be used.

Listing 3.3: Starting data processing framework with a dedicated UI logic

```
int main(int argc, char* argv[])
{
    return core::AppInitializer::runApp<CustomMainWindowClass>(argc, argv);
}
```

3.2.5.4 views

Based on core functionality different views can be proposed. They are dedicated to particular user needs, having common, standardized low-level modules. This are main application executables being compiled with *corelib* library and initializing application with specific UI. Currently we are supporting two views of applications:

- MDE as our leading project for medicine,
- so called *OldView* for tests, where very little higher level logic is introduced to verify core functionalities.

View logic must be developed by creating classes derived from *CoreMainWindow*, which is a bridge between user and data processing logic. It introduces graphical windows context (Qt) to data processing architecture.

3.3 Built-in plug-ins

Beside core logic presented so far, based on plug-in system, we are developing built-in plug-ins, giving users some out-of-the-box additional functionality. This section presents provided MDE plug-ins.

3.3.1 c3d

As already mentioned, some of our solutions are dedicated to motion data analysis. One of the presented motion data formats was *C3D* and a specialized plug-in was created to support it. Plug-in provides various motion time based data in *DataChannel* representation:

- EMG,
- GRF,
- markers positions,
- moments,
- forces,
- joint angles,
- powers,

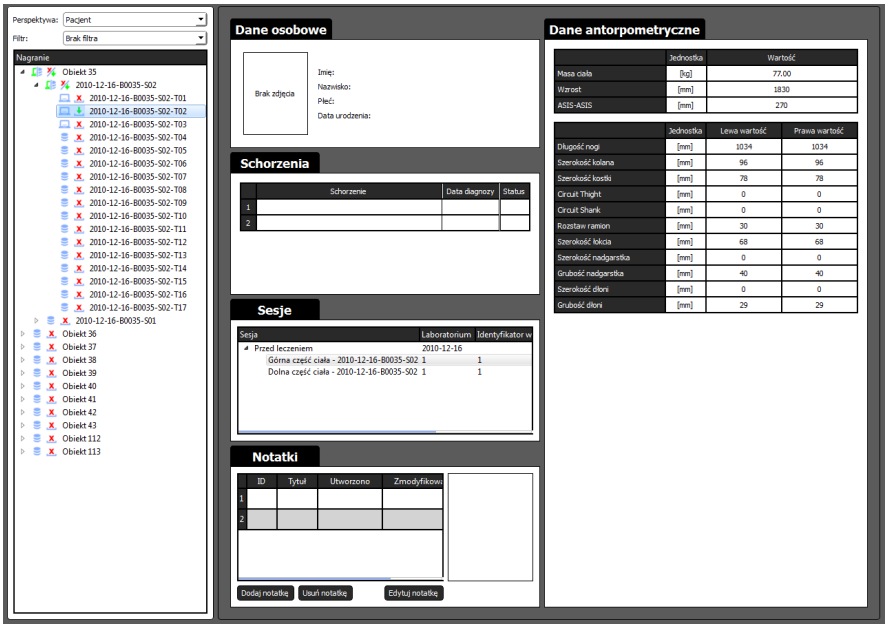


Figure 3.19: Communication data source

- video time offsets according to other measurements time,
- ground reaction force plates positions,
- events.

Provided events describe motion in terms of gait analysis, where swing and stance phases can be pointed out. Plug-in registers presented data types and a dedicated parser for a *C3D* files.

3.3.2 communication

We have mentioned at the beginning of this chapter that despite providing MDE as a data processing software, we have also developed additional technologies supporting motion data storage. Plug-in communication is dedicated to provide motion data from HMDB. It provides a dedicated data source, which main role is to give user an easy access to the centralized motion database. *Communication* data source allows browsing stored data according to granted data privileges and download data of interest to user computer. Later, downloaded files are passed through the core functionality of data loading procedure with help of files and streams managers. Loaded data are organized in a well defined hierarchy proposed by plug-in *Subject*, presented later in this section. *Communication* hides completely network

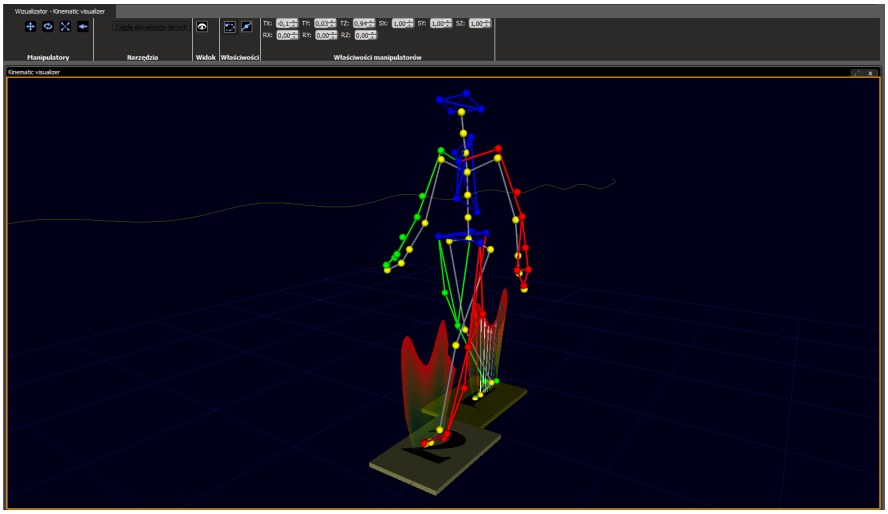


Figure 3.20: Kinematic visualizer

communication layer and web services querying. Its base functionality for motion database communication is delivered as an independent, standalone library, giving users possibility to access motion data also from their own applications. Despite provided library and a dedicated SSL certificate for a server authentication, a simple registration procedure must be completed to subscribe for a new account. It has to be mentioned, that stored data have large volumes and their download may take longer time. Figure 3.19 presents *Communication* data source view in MDE. Provided data can be vied at different perspectives, allowing to filter motion data according to various criteria.

3.3.3 kinematic

Plug-in kinematic provides dedicated data types for representing motion in form of a hierarchical skeleton model. For each joint motion data (rotations/orientations) are represented in form of quaternions or Euler angles time series encapsulated with *DataChannel*. Skeleton structure and data are separated, allowing to supply different motions of the same person to its dedicated skeleton model configuration. All mapping between joints data and skeletal model is done automatically, based on joints description. Plug-in delivers also a specialised visualizer for 3D scenes, which has the following features:

- complete 3D scene manipulation,
- skeleton model manipulation (rotation, translation, scaling),

- customizable skeleton ghost mode, presenting its previous, current and future states in time domain,
- support for many skeletons on the scene (multiple data series),
- marking currently selected scene element (skeleton, markers, force plates),
- ground reaction force plates visualization,
- GRF vector visualization at position where feet touch the plates,
- joints and markers trajectories visualization,
- predefined skeleton perspectives (left, right, top, bottom, front, back),
- selective data presentation (chose which joints, segments and markers to present),
- data time cropping (limiting viewed data to a particular time range).

Figure 3.20 presents example of kinematic visualizer.

3.3.4 chart

Chart plug-in provides a visualizer for scalar, time based data. It exports a simple *DataChannel* interface for supported data. Beside simple data plotting it has many additional features:

- managing visibility of data series,
- simple data statistics,
- value preview (displaying data value under current cursor position),
- value marker (displaying data value in particular position for chosen data),
- vertical range marker (displaying time difference of two chosen data samples),
- horizontal range marker (displaying value difference of two chosen data samples),
- data manipulation (scaling, translating),
- data series legend,

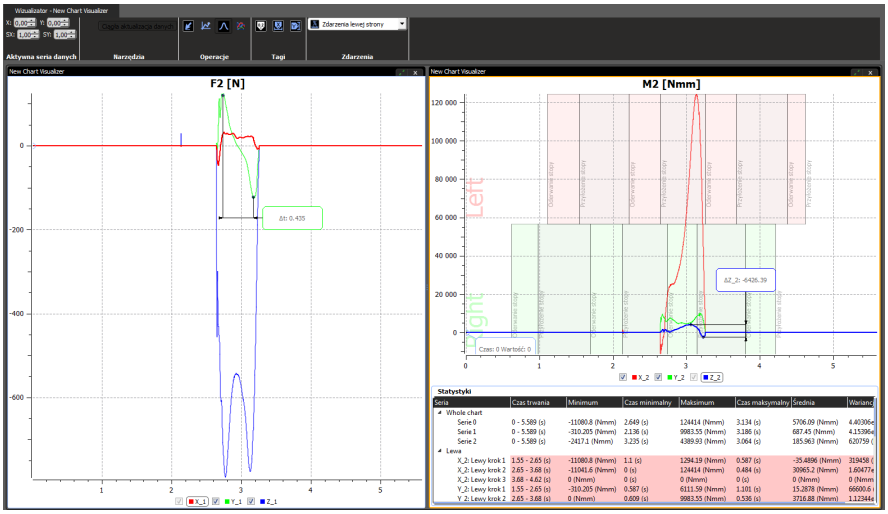


Figure 3.21: Chart visualizer

- axis description,
- plot description (title with values units),
- gait analysis perspectives (for left and right leg),
- time events presentation (stance and swing phases),
- automatic plot fit to data range,
- automatic gait perspectives switching on data series time change.

It is planed to extend chart giving users access to chart scene to introduce custom graphical chart elements and functionality. Figure 3.21 presents chart visualizer example.

3.3.5 timeline

Analysis of time based data requires time synchronization, or more generally, time domain to be normalized, as sometimes data with different frequencies and time resolutions have to be processed. To ensure uniform time based data handling in the time domain, plug-in *timeline* was developed. Most of its features are time operations on delivered signals, where original data are never modified explicitly. Dedicated abstraction layer was proposed, handling all time manipulations. *Timeline* is implemented as a dedicated MDE service. Time based data have to be wrapped with provided time channel interface.

Figure 3.22: Timeline playback controller

Timeline allows to organize time data in a hierarchical structure, combining particular data in virtual time channels with two simple operations:

- add channel to hierarchy (introducing virtual channels in hierarchy when required),
- remove channel from hierarchy (removing empty virtual channels in hierarchy).

It allows to apply time modifications to each channel separately automatically propagating their results through the channels hierarchy:

- offset - shifting channel in time,
- scale - changing channel time resolution,
- split - creates two separate virtual channels from one channel in a given splitting point,
- merge - joins two separate virtual channels to a one channel.

Timeline offers possibility to present graphically time events for particular channels. This is very usefully when user wants to analyse time data in some specific time range defined with particular events. *Timeline* supports data playback. It can be used as a base for visualizers to manage their time aware data series in a synchronized manner. With such approach chart plot, kinematic scene and video can be viewed in parallel for the same time. Data can be played forward and backward. Dedicated slider with current time input widget offer users manual time manipulation during playback and when *timeline* is stopped. Quick jumps to the end and the beginning of the managed time range are provided. *Timeline* automatically controls, if current playback time is within data channels range (after applying all external and local time modifications in hierarchy). If this time is in the channel time range, then it is applied to the channel, else, when channel data range has been reached, its time modification is not required any more, until time returns to the covered range. Figure 3.22 presents *timeline* visual playback controller and time manipulators.

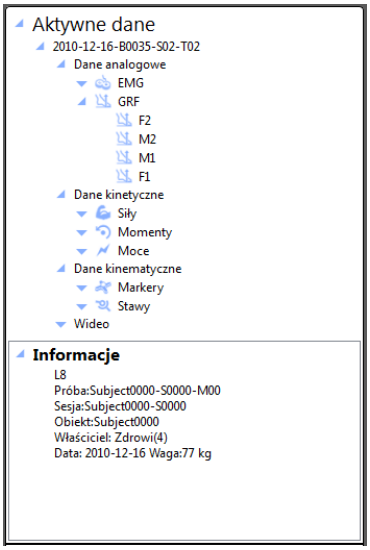


Figure 3.23: Subject hierarchy and data grouping

3.3.6 subject

The idea of plug-in *subject* it to organized motion data in a scheme provided by the HMDB. Data is grouped into motions and sessions for different actors (subjects). This service provides a general functionality for other MDE modules, which may also deliver data in a similar scheme or can be reorganized to fit it. Introduced grouping ensures, that all provided subjects, sessions and motions have unique identifiers, which in general are independent from identifiers delivered by the HMDB. Such data organization allows to present data clearly to users. Also data processing should be easier, as data is already grouped and logically consistent. Plug-in *subject* provides therefore new data types and service, handling (creating) particular instances of group categories with proper identifiers. They are extensively used in MDE view for data presentation, querying and filtering. This is presented in Figure 3.23, where data are grouped according to *subject* hierarchy and data types.

3.3.7 video

Video plug-in functionality is self-explanatory. Simple *DataChannel* interface with time indexed images is provided. It can be used to deliver video data to application from various sources. A dedicated visualizer is provided to watch loaded videos. This visualizer supports single data series per visualizer, as managing more videos at once is not a standard behaviour. Additionally to mentioned interface and visualizer itself, a dedicated parser is given, that can

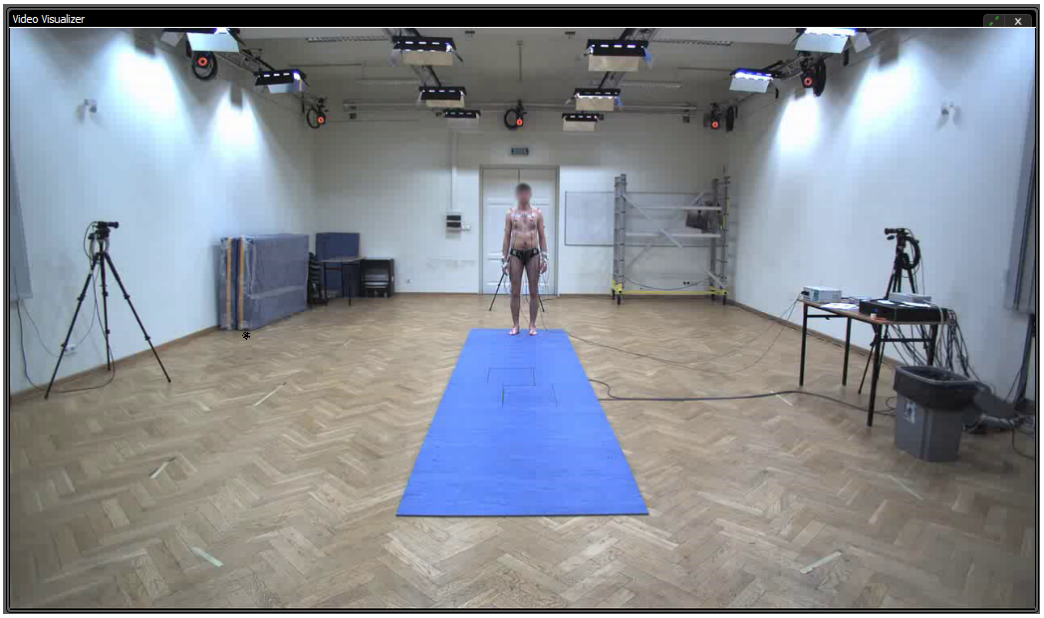


Figure 3.24: Video visualizer

handle many different video formats efficiently. This parser behaves slightly different then already presented parsers, as it extracts new data from attached video, when data channel is queried for particular time values. Parser reads from particular file a fixed size buffer, containing data for the given time and several nearby frames. Next queries do not perform any I/O operations, as long as queried time is still present in the current buffer. This optimizes sequential video reading for playback purpose. This approach is implemented to save memory, as video files usually use lots of storage space, depending on their quality, duration, format and compression. Figure 3.24 presents video visualizer.

3.3.8 python

To give users a flexible way of interacting with the application independently from GUI, we decided to provide scripting languages binding. In the firs step only simple python binding is offered. It is integrated with system console allowing to pass custom scripts to python interpreter. Currently only plain python environment is available, as plug-in is still developed and python wrappers for core elements are being prepared. Plug-in system gives us an easy way to provide also other scripting languages bindings when required, but as this is an additional feature, not crucial for data processing, we decided to reschedule it for a later time. Figure 3.25 presents python environment

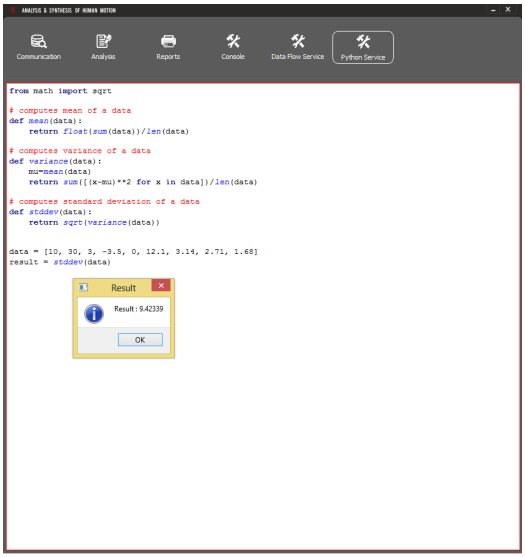


Figure 3.25: Python environment

inside MDE.

3.4 Implementation and development

Now we shortly describe process of application development. We want to show, how geographically spread team can efficiently communicate, plan further work and verify project overall progress. We want to present developed CI process, that supports application development. In this section also several other tools are presented, that support us in code development. We start with a short project description.

Project overview Major part of presented solutions were developed within a period of two years - from 2010 till 2012. During this time different application concepts were tried and developed in parallel. Their various parts were combined together and reorganized to become a mature and reliable processing logic presented in this thesis. MDE was developed by varying number of developers, starting from 3, up to 7 in total. There has been always one team leader, coordinating all other members actions and overall project progress. As involved people presented different programming and engineering skills, some steps had to be taken to ensure developed application quality. Also this steps have evolve in time, bringing us to the present time, when we think the right habits and methodology were achieved and introduced to the process of software development.

Work scheduling and team cooperation It was decided to use the Assembla (`https://www.assembla.com`) as an on-line project space, which provides a great support in project leading and management. To provide globally an information, how project is advancing, each developer is reporting his everyday work with short message, called *stand-up*, containing three simple informations about:

- what he did particular day,
- what he plans to do next day,
- what are potential pitfalls, that might delay his work and may require wider team discussion.

With such simple approach each team member is always aware of the current work done by other colleagues. To organize and schedule application development stages, each job is divided into smaller tasks, attached to particular programmers. Tasks are called *tickets*, and responsible developers are accepting next tickets, when they finish their current work. Any knowledge exchange, connected with particular *ticket* is done in the tickets space, providing comprehensive history of different concepts and topics, giving an overview how particular module design has changed in time. Assembla offers also a possibility to create project *wiki*, where fixed, general information about the project and future plans can be shared in an organized manner. Additionally, one of its services is hosting Source Version Control (SVN) repositories for team projects. They are fully backed up in the cloud, providing secure, continuous access to the repository.

Cross-platform projects As project is written in C++, since the beginning it was considered to be a cross-platform application. To handle project solutions under different operating systems and Integrated Development Environment (IDE)s, CMake (`http://www.cmake.org`) was chosen. It provides simple scripting language to describe project configuration, delivering generators for most platforms and development environments, from simple makefiles to complex Visual Studio or Eclipse solutions. As its basic syntax requires creation of large, complex and in general similar projects descriptions, we have developed custom framework to handle our projects on CMake basis. This framework offers many advantages over built-in CMake capabilities, providing methods for simple project configuration, external libraries handling and inter-project dependencies management. This allowed us to minimize time spend on defining new, or extending existing projects

with CMake basic syntax. Additionally, our solution handles process of artefacts management and installers generation. It is worth mentioning, that this framework is treated as a completely independent project with its own space on the Assembla. It is a base project for all other projects. CMake provides also testing tool, called CTest which we also use. Installers are handled by an integrated with CMake module called CPack.

Developer tools For different platforms variety of developer tools and compilers are available. For Linux platforms open-source Eclipse IDE (`http://www.eclipse.org`) is used with gcc and g++ compilers (`http://gcc.gnu.org`). For debugging gdb (`https://www.gnu.org/software/gdb`) was chosen with simple graphical layer. Under Windows operating system Microsoft Visual Studio 2010 is used. Windows platform is the main development platform. Programming under Linux is done, when some tests fail for this platform or application is crashing in this environment. Such approach was dictated by simpler and more efficient development tools for Windows. To increase developers productivity and code quality plug-in for VisualStudio is used - VisualXAssist (`http://www.wholetomato.com`) - which speeds up overall code development, especially code browsing and re-factorization.

External libraries To prevent developing common functionalities from a scratch for different platforms and maintaining them to ensure their stability and functionality, it was decided to use well tested and commonly used external libraries and frameworks. As project covers wide range of programming problems, such approach allowed us to build MDE core as an integration and composition of already existing solutions. This is how such relatively small team was able to develop so complex product. When the project started, C++0x and C++11 standards were not finished and available in modern compilers. *Boost* (`http://www.boost.org`) library was used to provide missing functionality. Firstly, it was only memory management with smart pointers, but currently whole file system and unique elements identifiers are based on its components. Next important decision to make was choice of GUI framework. As it had to be a cross-platform solution written in C++, the Qt tool-kit (`http://qt-project.org`, [59, 8]) was chosen, as the most user friendly and mature framework. Today we know it was a good decision as new versions of this set of libraries extend supported platforms on mobile devices, which we start to consider as a next target for our upcoming products and solutions. Last such a big choice has been done choosing OpenSceneGraph (`http://www.openscenegraph.org`) library for managing 3D scenes. It is rapidly developing, but stable, general purpose 3D engine,

supporting whole rendering process with many optimizations and support for common scene operations and modern hardware. Moreover, it is very intuitive even for users not keen in computer graphics, with great community, offering additional support, when needed. Other libraries worth to mention here are:

- FFmpeg for video decoding (`http://www.ffmpeg.org`),
- TinyXML for xml data processing (`http://www.grinninglizard.com/tinyxml2/index.html`),
- log4cxx as logging framework base (`http://logging.apache.org/log4cxx`),
- cUrl for handling network data transfers (`http://curl.haxx.se/`),
- OpenSSL for secure connections handling (`http://www.openssl.org`),
- Sqlite with SqlCipher as application one-file local database (`http://www.sqlite.org`, `http://sqlcipher.net`),
- Qwt as base for chart functionality (`http://qwt.sourceforge.net`),
- B-tk as parser for most motion data formats (`http://code.google.com/p/b-tk`),
- cppunit as a unit testing framework (`http://sourceforge.net/projects/cppunit`).

Continuous integration Providing such a large number of libraries for different platforms it was difficult since the beginning. As their number was growing in time because of different external dependencies like:

- zlib (`http://www.zlib.net`),
- jpeglib (`http://www.infai.org/jpeg`),
- libTIFF (`http://www.libtiff.org`),
- giflib (`http://giflib.sourceforge.net`)

and many others, it was clear to us, that some automated system is required to handle their build process. From that point of time it was quite a long way to reach the present time, when we have developed mature, fully automated system for obtaining external libraries, building them for different platforms and delivering build artefacts to developers. Additionally, it

provides functionality for managing also our projects, making their periodic builds with respect to the delivered external libraries. Any failures in external builds or custom projects are immediately reported to developers, so potential problems are caught and resolved as soon as possible. Having such a great tool, it was just matter of time to extend it with different code quality measure features, reporting quantitative, well defined code status of developed projects, allowing to keep the code clean, safe and efficient. It is also used for scheduling re-factoring tasks for too complex modules, which could cause many problems in the future with their maintenance and extending their functionality. In the end, whole procedure of projects testing is handled, which starts generation of product installers, if all test have passed. Any testing failures are reported immediately to developers. For all those functionalities historical progress can be traced to detect bad trends in the software quality. Presented functionality is covered by two tools:

- Hudson (main CI tool, external libraries and custom project builds, testing, installers and documentation generation, `http://hudson-ci.org`),
- Sonar (code quality measures, `http://www.sonarqube.org`).

Figure 3.26 presents general schema for CI. Figure 3.27 shows how complicated dependencies can occur between different external libraries and custom projects with intermediate build stages.

Development standards As modern functionalities becomes more and more complex, despite a good architecture design is it important to maintain high programming standards. In our development work we use different kind of design patterns for well known and defined programming problems, starting with Private Implementation (PIMPL) idiom, going through Resources Acquisition Is Initialization (RAII) pattern, ending on visitor patters and advanced template meta-programming. Combination of all those techniques allowed us to provide high quality code, based on best programming practices [61, 60], ensuring us in the right development direction. They also allow to maintain application great flexibility for new features, keeping its high performance. Such approach limits debugging process time, decomposing problems to individual behaviours, making bugs easier to trace and fix. Despite usage of different design patterns we have introduced internal coding standards. They cover detailed code comments, dedicated mainly for public headers. Files are commented in an hierarchical order - firstly general overview of file content is given, with optional examples and explanations for more complex algorithms. They might contain author information and date, when

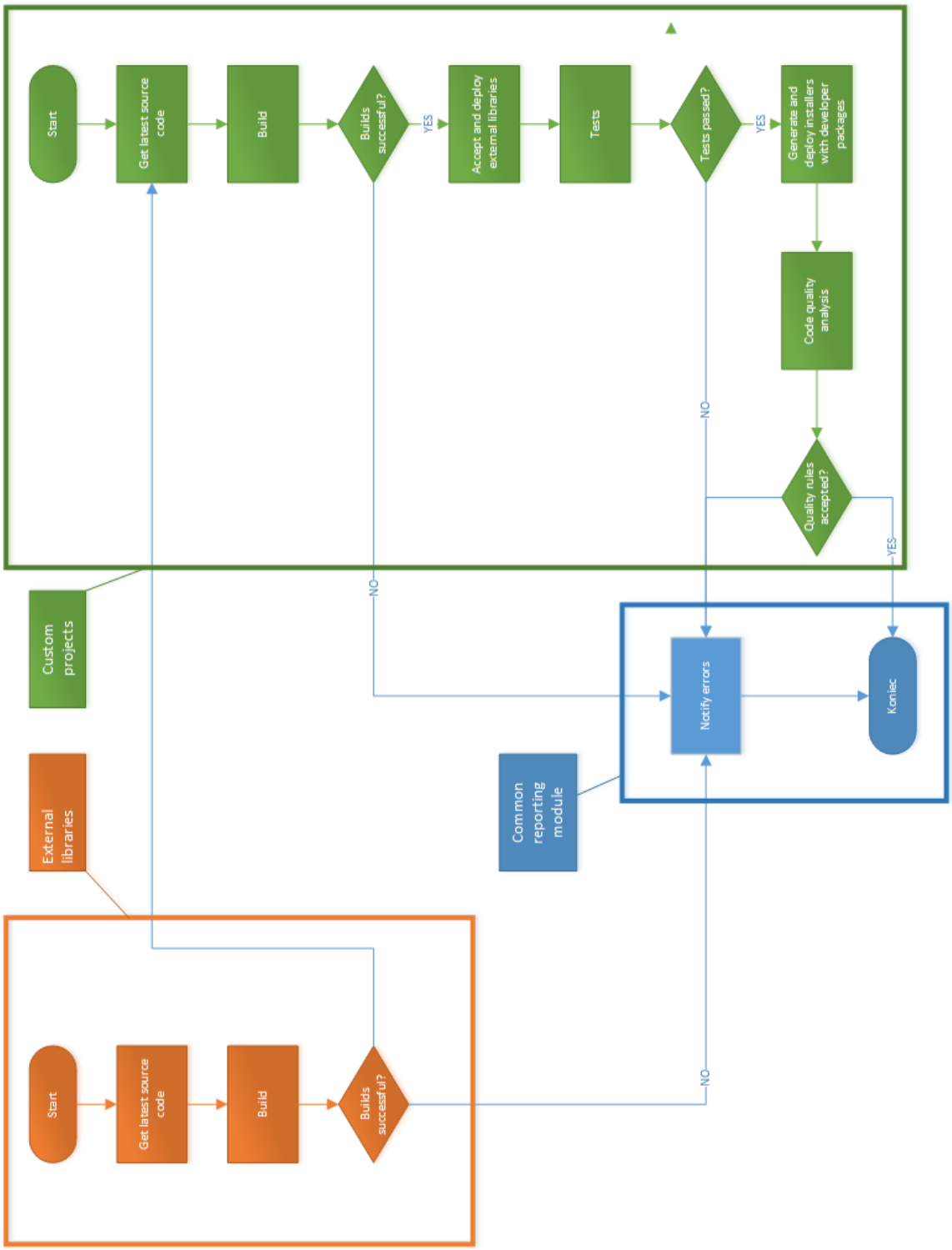


Figure 3.26: CI process overview

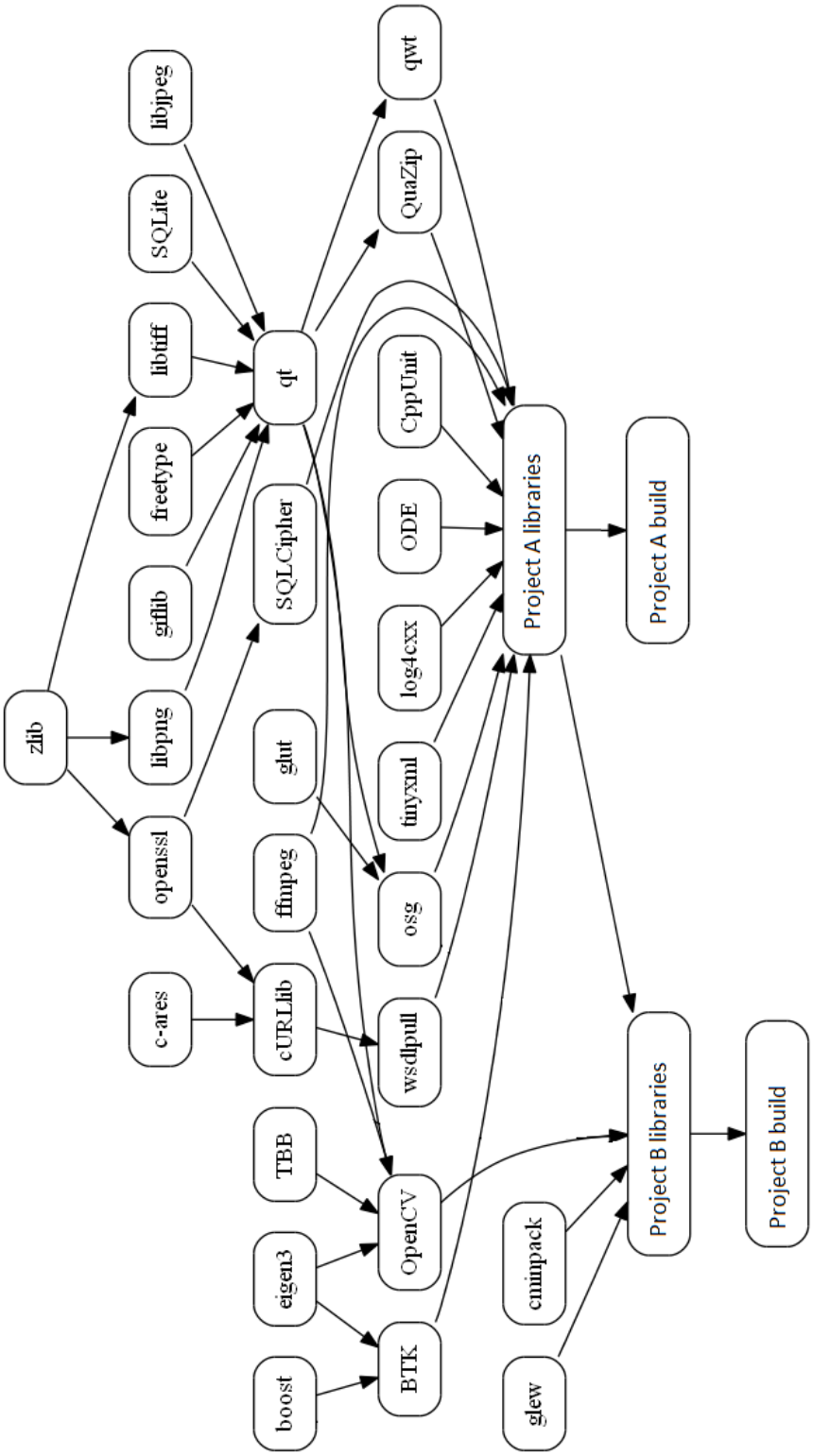


Figure 3.27: External libraries hierarchy with custom projects dependencies

particular file was created and for what purpose. Later, particular classes, methods, free functions and new data types are defined and described. Doxygen (`http://www.stack.nl/~dimitri/doxygen`) style is used for comments with Doxygen tool itself for generating technical documentation based on developed code. Documentation generation is automated as a part of CI process.

3.5 Future work

There are two areas, where we have scheduled new features and additional work. First one covers presented CI process. Second one is connected explicitly with new application functionalities and improvements.

CI For continuous integration procedure we want to introduce a bug tracking tool. We believe, that this is the last element, missing in developed automated procedures of building, testing and developing custom projects. Our goal is to expand number of users working with MDE and testing it, giving them tool to report application errors. Additionally, we are thinking about automated reporting module for MDE, which on different application errors would try to send application log for verification. Such notifications could be later scheduled with other development work, allowing to plan better new activities.

Moreover, we still see several improvements for the current CI process. Now separate servers are maintained for Linux and Windows platforms, building almost identical set of external libraries. We want to join both those servers into one general solution to schedule various CI procedure stages work to dedicated machines in a uniform manner. Currently each of those servers updates external libraries source code independently. This have lead us several times to a situation, where on one platform libraries with the same version number were build from different code revisions, causing application to crash on the other platform. As we are now aware of this dangerous situation, we can quickly solve it by forcing particular library refresh on both platforms manually. Our goal is to have one dedicated machine for updating code and scheduling builds on other machines based on the same, common source code.

MDE For MDE several improvements were already mentioned. This are in the first step scripting languages and utilities for general UI management and design for data processing. New functionality for *chart* plug-in is planned,

allowing users access plotting scene and extend charts general capabilities. Similar operation is thought for kinematic and video visualizers, where access to the 3D scene and current video frame is considered.

The greatest changes are planned for plug-in *communication*. We want to offer additional services supporting team work and knowledge exchange. As we have gained lot of experience with providing motion data through a dedicated HMDB, we want to extend its functionality to allow users to store their processing results specific data in centralized database. This would be done through a dedicated user space, where users might upload their work results. Later on, they could share those results with other users, by explicitly sharing particular resources to defined users and groups of users. Some data might be marked to be visible to all system users, for example to give a short overview of research work they do. This would allow other users to contact them, if they are also interested in those particular topics. Despite knowledge exchange we would like to introduce tools for group management which will support organizing users in particular teams. Most of our efforts would be focused on this functionality in nearby future, as we claim that data processing is generally a team work task and efficient work results sharing and knowledge exchange have the highest priority.

For *JobManager* it is considered to implement more advanced *Job* scheduling system, where synchronization delays for processing threads would be limited by independent work queues and work stealing approach, as presented in [9, 1, 23, 15]. It has been proven that this solution improves calculations performance, as logically connected jobs can better utilize CPU cache, when similar data are processed. It would also introduce an order, where it is explicitly known, which thread is processing particular *Job*, allowing more flexible jobs management.

As data management was standardized with OWs and dedicated data managers, a functionality similar to garbage collection can be proposed, based on OW lazy initialization features. Memory data manager can be extended to trace the time of last data access during various data queries. Based on such information it is possible to release resources not used for a longer time, when there exist a dedicated initializer object, for example for data extracted from files and streams with parser specific initializers. Such data would be release only if no more references to it are present in application (investigating reference count for smart pointers). This would allow to save application memory.

Chapter 4

Data flow processing and visual programming

4.1 Introduction to data flow processing

We want to describe data processing as a concept of data flow, which we find to be very intuitive. It allowed us to develop very simple, yet efficient tool for designing and realizing general purpose data processing. Firstly, we introduce in details data flow idea with all logic elements it covers. Additionally, data flow independent functionality layers are pointed out. We discuss required operations, that need to be provided to execute so defined processing. Later, we present developed library for general purpose data processing based on data flow concept. We give examples of abstract interfaces, that need to be provided to use developed data processing. In the end, we describe, how this general purpose processing library is wrapped to deliver complete data processing functionality for MDE, based on provided core functionalities.

4.1.1 Data flow graph concept

Generally speaking, data flow structure might be compared to a graph. The following logical mapping between graph elements and data processing can be given:

- vertexes - represent operations performed on provided data,
- edges - describe data propagation structure between various operations, define the direction of data flow.

Such description must be however extended, as it is not accurate enough for data processing. In data flow nomenclature vertexes are called *nodes* and

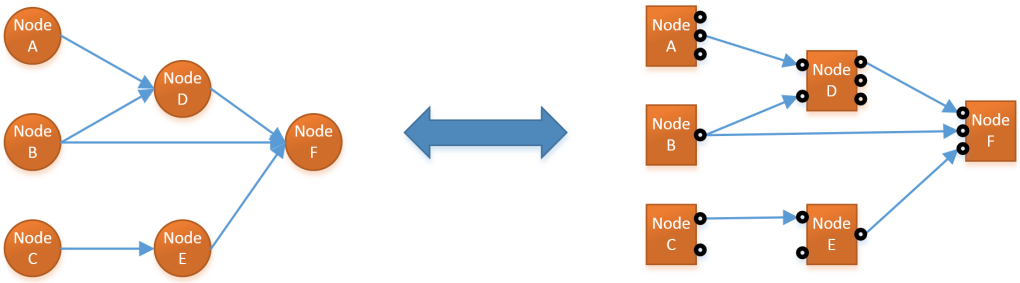


Figure 4.1: Example data flow structure for a given graph

edges are called *connections*. In contradiction to graphs, where vertexes are explicitly connected with each other, in data flow logic additional elements are introduced - *pins*. They are integral part of nodes, allowing nodes to connect to each other. Moreover, three types of nodes can be pointed out, according to data processing logic they provide:

- source nodes - providing new data for processing,
- processing nodes - working on delivered data and providing processing results,
- sink nodes - consuming processed results for further usage, ending data propagation.

Pins are also divided in two categories, depending on their role in nodes:

- input pins - delivering data to node,
- output pins - propagate processing results to connected nodes.

Figure 4.1 presents comparison of a simple graph and corresponding example of data flow model.

In graphs edges might be either one direction or bi-directional links between vertexes. In data flow connections are represented as strongly ordered pairs of pins:

- source pin - output pin,
- destination pin - input pin.

where presented order defines connection direction, consistent with data flow nature - from sources to sinks. Such approach forces basic rule for connecting pins - only pins of different type can be connected, as linking two input pins or two output pins makes no sense, because they do not provided complementary processing functionality.

Thinking about data processing with data flow, two independent functionality layers can be pointed out:

- designing data flow processing graph,
- executing data flow processing.

Building processing graph defines, how data is going to be processed in terms of sequence of executed operations on the data. It can be a simple linear pipeline or a complex graph with recursions. Executing data processing defines, how this procedure is going to be realized in terms of:

- data passing between particular operations,
- when new data is delivered for further processing,
- when whole processing should be finished.

4.1.2 Data flow model design

For building processing graph a general purpose functionality for model creation can be given. It is responsible for managing nodes and connections. Model manages well defined nodes instances and connections between them, based on nodes pin configurations. It might additionally introduce some connectivity rules, limiting possible connections combinations to a set of valid connections in term of a dedicated problem. Model provides functionality for querying about managed nodes and connections, allowing to traverse model structure from particular node or connection in both directions, according to fixed order of pins, handled by connections. Now particular model elements, with model itself, are presented in details, to explain how graph structure management can be realized.

4.1.2.1 Nodes

Nodes provide simple functionality for querying the attached pins configuration:

- number of input/output pins,

- input/output pin at the given position (unique internal index).

Only processing nodes allow to query for both pin types, other nodes offer querying for one particular pin type. What is very important, nodes after creation must not change their pins characteristics. Without such rule maintaining model consistent would be hard and complicated logic will have to be developed to handle such functionality. Additionally, such behaviour does not seems to be natural for graph elements. Node provides additionally information about number of connected pins and if it is connected at all. It can be queried about its type for easier management in the model.

4.1.2.2 Pins

Pins provide functionality for checking their connectivity status:

- if they are connected,
- for how many connections they are used,
- connection for given internal index (position).

This operations are sufficient for node to define its own connectivity status. Each pin is aware of node, to which it is attached, making it possible to follow data flow graph structure from input pin, through node to output pins with well defined connection direction. Additionally, information about pin type is available, to make connecting procedure easier. This functionality is however private and reserved for model only, as it must have an opportunity to verify particular connections with provided connectivity rules, disallowing connections violating those rules.

4.1.2.3 Connections

Connection allows to query for linked pins: source pin and destination pin. It can not be reconnected to another output or input pin explicitly - firstly old connection must be removed and new one must be created to verify its correctness. Based on those simple operations reconnection logic might be easily developed, if necessary.

4.1.2.4 Model

Model manages nodes and connections. Its main operations are:

- node addition,

- node removal,
- connecting nodes,
- disconnecting nodes.

Model provides also its state description in form of managed nodes, grouped according their types, with the list of current connections. It must guarantee that node management operations are consistent, that is node removal will also remove all node connections. Additionally, adding new node with unknown connections is not allowed, as model did not verified those connections according to custom connectivity rules.

Different models can introduce new rules for connections, for example allowing or disallowing recursion in their structure, depending on a logical structure they are going to describe and problem which would be solved. It can be noticed, that introduction of pins to graph structure maintained graph functionality, making its representation more fine grained and well suited to describe data processing.

4.1.3 Data flow model processing extensions

Designing data flow model for processing requires some extensions to pins and model concepts. Model elements must handle minimal set of operations allowing to:

- deliver new data for processing,
- process data,
- store processed data,
- pass data between nodes.

To describe them clearly, we want to introduce an analogy for data flow processing to a procedural programming, where well defined paths of execution are given. They cover not only order of execution for particular operations, but also operations signatures, defining their input and output data. Keeping that in mind data flow processing can be defined as a sequence of function calls, where some of them:

- do not have any input data and provide output data,
- require specific input data and providing output data,

- require input data, but do not produce any output data.

This analogy gives a clear mapping between data flow nodes and function signatures. It is consistent with nodes types and their responsibilities. We discuss now extensions to pin elements, allowing data flow structure to describe function calling mechanism with model additional verification rules for data processing realization.

4.1.3.1 Data flow pins

According to introduced procedural programming analogy, pins can now be compared to function parameters and their returned values. Such interpretations makes it also clearer, why nodes must not change their pins configuration after creation, as most of programming languages do not allow to change function signatures after compilation. For data processing purpose pins are extended with additional properties:

- input pins can be marked as required or optional,
- output pins might depend on input pins.

First property allows nodes to represent functions with default parameters, where not all input parameters have to be provided as their default values will be used instead. Second property allows to describe internal data transition inside of the function, defining what kind of outputs can be expected, when particular inputs are provided. Both those attributes are used by a model to verify model correctness.

4.1.3.2 Data flow model verification

Having extended pins properties, model can verify those additional connectivity rules. First of all, in a valid data flow all required pins must be connected. Secondly, when particular output pin is connected and it depends on some input pins, all those input pins must also be connected. This ensures that described data processing pipeline is correct in terms of functions signatures, their internal data transitions and call paths. Additionally, for data flow model input pins are limited to be a part of a single connection, while output pins can create any number of connections. This might be compared to calling a function, when a fixed, well defined argument must be provided as an input, while resulting data might be used or copied by any number of functions. In the end, all model nodes with input pins must have at leas one input pin connected. Also source nodes must have connected at least one output pin. Otherwise, there could be some free functions in the pipeline,

that would never execute their functionality or executing it would not affect overall data processing, as either there are no inputs for them, or produced data would not have any recipients. Model keeps structure clean and simple, forbidding existence of unused and inactive nodes.

4.1.4 Data flow processing logic

Having properly constructed model describing data processing steps, understood as function call paths, data flow processing logic can be elaborated. Before proposing custom approach to processing logic, several important questions have to be answered about its requirements:

- what processing scheme should be used,
- which data propagation model to apply,
- what are the rules for loading new data to data flow,
- how data is going to be exchanged between nodes,
- when data flow finishes data processing.

Each of this points in details defines data processing realization for provided data flow model. They all explicitly affect data flow efficiency and implementation, therefore we want to give their overview in next paragraphs, before we present our solution for this task.

4.1.4.1 Processing scheme

Two processing schemes can be pointed out, according to parallelism of nodes work:

- sequential,
- parallel.

Sequential data processing is simpler in implementation. Nodes process the data one by one in the well defined order, starting from sources, going forward in the model structure as long as required data are available. When there are missing some data, the processing returns to previous nodes on the given path, continuing from a place, where data is available and have not been processed yet. This however limits drastically utilization of modern hardware computational power, where many tasks can be processed in parallel. To ensure efficient processing power utilization, parallel data processing

approach should be chosen. Different nodes can work in parallel, when they have all necessary data. Their ancestor nodes in the model obtain required data faster for processing, making overall data processing more efficient. This solution however is harder to implement, as complex processing logic requires careful synchronization between particular data processing stages, avoiding potential dead-locks between different tasks in the processing logic.

4.1.4.2 Data propagation

Two basic operations, despite data processing itself, in the data flow can be pointed out:

- load - delivers new data to node for processing,
- propagate - processed data are delivered to ancestor nodes.

Those features describe, how processing logic is scheduling execution of ancestor nodes tasks in the model. Independently from parallel scheme, two approaches are possible:

- prefer new data to be loaded to model and then process data, if model nodes are maximally loaded with data,
- prefer data to be processed as far as possible in model, before loading new data.

In the first approach nodes closer to data sources can be preferred in the first place to process the data and then processing logic approaches forward to the ancestor nodes, so they are always guaranteed that all data is already present for processing. Such approach tries to load new data to the model as soon as possible, keeping data flow pipeline maximally loaded with data to process. Second approach is based on opposite behaviour - going as far forward with data processing in the model as it is possible and returning back to nodes at previous levels, only if there are not enough data for further processing at current level. This makes loading of new data delayed until is is required to continue further processing because lack of data. With this approach potential results should be obtained earlier.

4.1.4.3 Loading new data to model

Next question to be answered is, how long data is delivered to the model with the source nodes. It can happen that sources may provide different number of data for processing. Two solutions can be chosen:

- as long as any source can provide new data,
- as long as all sources can provide new data.

First approach tries to process data in the model as long as any of the sources can deliver new data for processing. It gives a chance for more data to be processed, although at some processing stage it might occur, that further processing is impossible. It can happen because of lack of required data being delivered by a processing branch starting with other source, that could not provide more data. Second approach finishes data loading, when any of sources can not provide more data for processing. This also introduces synchronization among data sources to verify at each data load stage, that sources still can provide more data. In this approach only complete input data sets are introduced for further processing, allowing to process the data through the whole data flow model. Second approach seems to be more reasonable and natural to choose, but there might be some specific applications, when first approach would fit better.

This property is also important as it explicitly affects stopping criteria for data flow, when decision is made, if data processing should be finished. This is elaborated in one of the following paragraphs.

4.1.4.4 Data exchange between nodes

It has to be defined very clearly, how nodes exchange their input and output data. Strict rules for pins, connections and nodes must be given. Data passing is a very important stage of data processing pipeline, as it affects explicitly efficiency of data processing. Data passing logic must fit previous decisions about processing schema and data propagation. Different dedicated data passing functionalities must be provided for data flow elements, as they represent different levels of a model (model-node-pin-connection hierarchy). Exact moments of data copying and processing must be defined. Objects initializing data exchange must be pointed out. Set of requirements and conditions, allowing data exchange, must be specified. Data consumption and data sharing create unique communication protocol between nodes.

4.1.4.5 Finishing data processing

Previously made decisions define, how processing logic should monitor its execution to finish correctly data flow processing. There must be defined a set of rules that will verify if:

- model is still processing data,

- there are still data in the model, but their further processing is impossible,
- all data have been processed.

This topic is quite simple in case of the sequential execution, as there is a well defined path for processing logic, which itself introduces some stop criteria, based on chosen data loading scheme. This is however harder to provide and monitor data flow finish for parallel execution, when different tasks are processing data independently, with others waiting for their resulting data. Simple check, if thread is still working might be not enough, as it may sleep on some synchronization object, waiting for new data to come or to be consumed. A dedicated mechanism, monitoring amount of data delivered to the model and data that have left model, must be given, to provide reliable information about execution status of parallel data flow and its correct finalization.

4.2 General purpose data flow processing library

Now we present our solution to general purpose data processing based on data flow concept. We present simple interfaces as a ground for easily customizable data processing framework, presented in the previous sections. Presented logic is based on two layer approach, where first layer defines processing model and second one runs data processing based on the model structure.

4.2.1 Processing logic characteristic

We provide processing logic supporting utilization of all available computational resources by implementing parallel processing schema for nodes tasks. Sources are delivering new data to the model according to the rule *all-or-none*, stopping new data to be load as soon as one of the sources can not provide more data. Model prefers always to load new data first, before passing processed data forward for further processing. This approach allows to keep model maximally loaded with data to utilize effectively all available computational power. We do not force any mechanism for data passing, as it is specific for the problem that is solved with the data flow processing. Users are free to implement specialized and optimized data passing functionalities fitting their internal requirements, as long they follow general data flow concept and rules. Defining data passing mechanism requires from users

Listing 4.1: Input pin interface for data processing

```
//! Input pin interface
class IDFInput
{
public:
    //! \\param pin Output pin with provided data to copy
    virtual void copyData(const IDFOutput * pin) = 0;
};
```

to implement methods for binding particular data to input and output pins, and later to unpack this data from pins for processing. All other processing logic, with threading and synchronization is already provided, therefore users do not need to implement and control them manually. Data flow automatically handles proper processing finish, observing current state of data flow execution. Next sections present very simple interfaces that particular data flow model elements must provide to process the data. Additionally, logic of each element is described in details with attached state charts. We do not provide any internal implementation details as this is not relevant for getting familiar how to use so proposed data flow library. With presented detailed descriptions it is possible to develop custom data flow processing library.

4.2.2 Public interfaces

Now we present interfaces that have to be provided additionally to particular data flow elements functionalities to allow data processing.

4.2.2.1 Pins

We decided that input pins are responsible for copying data from output pins. Listing 4.1 presents input pin interface that has to be implemented. With such approach users can customize mechanism of data passing between nodes, making it optimized to their requirements and data representation. This solution does not force any data types nor mechanisms used for data exchange.

4.2.2.2 Nodes

As different nodes provide various data processing roles, three interfaces are proposed, each for particular node type, equivalent to its functionality. Listing 4.2, Listing 4.3 and Listing 4.4 present required nodes interfaces. This

Listing 4.2: Data source node interface

```
//! Data source interface
class IDFSource
{
public:
    //! \return True if source has no more data to provide
    virtual const bool empty() const = 0;
    //! Method providing new data for processing by setting up output pins
    virtual void produce() = 0;
};
```

Listing 4.3: Data processing node interface

```
//! Data processing node interface
class IDFProcessor
{
public:

    //! Method processing data from input pins and setting up output pins ↩
        with processed data
    virtual void process() = 0;
};
```

Listing 4.4: Data sink node interface

```
//! Data sink node interface
class IDFSink
{
public:

    //! Method processing data available in input pins (possibly data ↩
        serialization)
    virtual void consume() = 0;
};
```

Listing 4.5: Input pin wrapper internal data copy

```
// get wrapped input pin from the model
auto wrappedPin = getWrappedPin();
// extract attached connection (only one is avaiable according to data ↩
    flow model constraints for input pins)
auto connection = wrappedPin->getConnection();
// call copying functionality delivered by clients for attached output ↩
    pin, being data source now
wrappedPin->copyData(connection->getSourcePin());
```

high abstraction level allows to implement various mechanisms for data loading, processing and storing. Users are not bound to any specific data types and can freely customize this behaviour to the nature of the problem they are processing. Presented methods must however ensure proper data update in output pins, so that further data copying to input pins can be done successfully. Similarly, before data processing, users must unpack data from attached input pins. This can be easily implemented with various techniques, as data flow node offers required methods for querying about attached pins.

4.2.2.3 Model

As various data types might be processed with this universal data processing framework, it might be required to provide additional connectivity rules for a model, to verify particular data types compatibility with provided operations. It should ensure model logical consistency, because some general data operations might not be applicable to particular data types (i.e. squaring image instead of scalar values). This can be achieved by extending basic model functionality responsible for connections verification.

4.2.3 Processing logic details

Processing logic is based on dedicated wrappers for data flow elements. Based on model structure and nodes configurations, complete wrapped model clone is created with injected processing logic. Now we present wrapper objects for the model elements.

4.2.3.1 Pins wrappers

Input pin wrapper Input pin wrapper is responsible for delivering new data to the node from attached output pin. It starts data processing in its initial state - waiting for new data to be delivered. When output pin wrapper

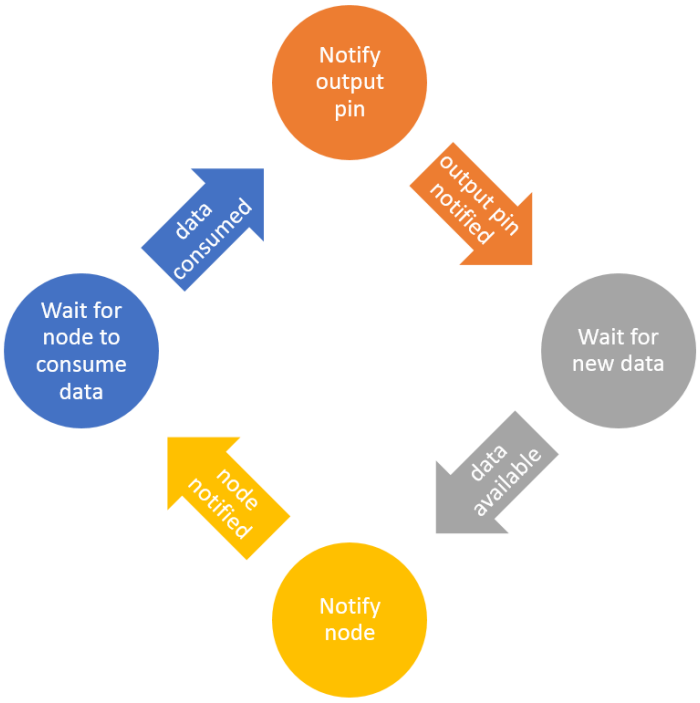


Figure 4.2: Input pin wrapper state chart

notifies input pin wrapper about availability of new data, input pin wrapper informs attached node, that it is ready to deliver new data. Next, input pin wrapper waits until node consumes provided data for further processing. This operation internally extracts wrapped input pin from the model with attached connection and linked output pin, calling public API, responsible for user defined data copying operation. Listing 4.5 presents pseudo code for this operation. When data is copied, input pin wrapper informs output pin wrapper that data is already consumed and starts waiting for new data. This allows predecessor nodes to deliver new data for processing, while current ones are being processed. Figure 4.2 presents state chart for input pin wrapper.

Output pin wrapper Output pin wrapper is responsible for propagating new or processed data for further processing in data flow. Its initial state is waiting for new data to propagate. When node delivers new data, it is his responsibility to set them to output pins. When data is already available to output pins, output pin wrappers notify all connected input pin wrappers about data availability. This is different than for input pin wrappers, as they are limited to a single connection and output pins can deliver new data to

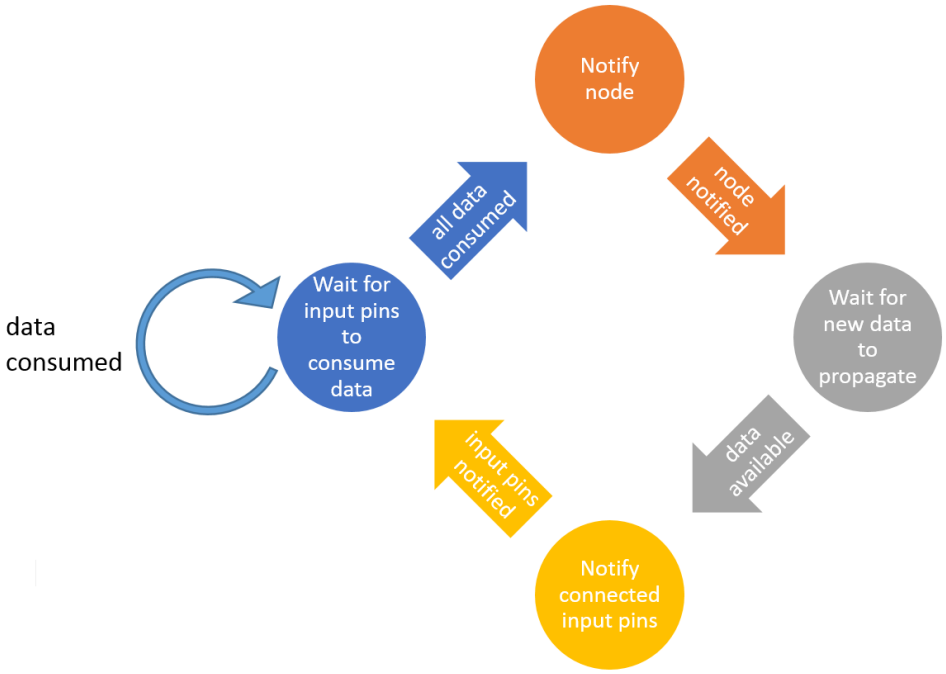


Figure 4.3: Output pin wrapper state chart

potentially unlimited number of input pins at once. This allows to optimize data sharing procedure (i.e. by using smart pointers instead of real data values copying) and to observe data changes at some intermediate steps of data processing. When all input pin wrappers are informed, output pin wrapper waits for all of them to consume the provided data. If all connected input pins have copied the data, output wrapper pin notifies node, that it is ready to provide new data. Next it starts waiting for new data from the node to propagate them and repeat the cycle. Figure 4.3 presents state chart for output pin wrapper.

4.2.3.2 Nodes wrappers

Source node wrapper Source node wrapper states are defined with a previous decisions about the way we load new data to the model. Such approach enforces source nodes to work synchronously, sharing information about their available data to deliver. Assuming all source nodes in the model can provide data at the beginning of data flow run, their corresponding wrappers start with producing new data. At this stage new data are delivered to the model and they have to be set up to output pins, so they can be copied further for processing. Next, each source node wrapper is waiting until all

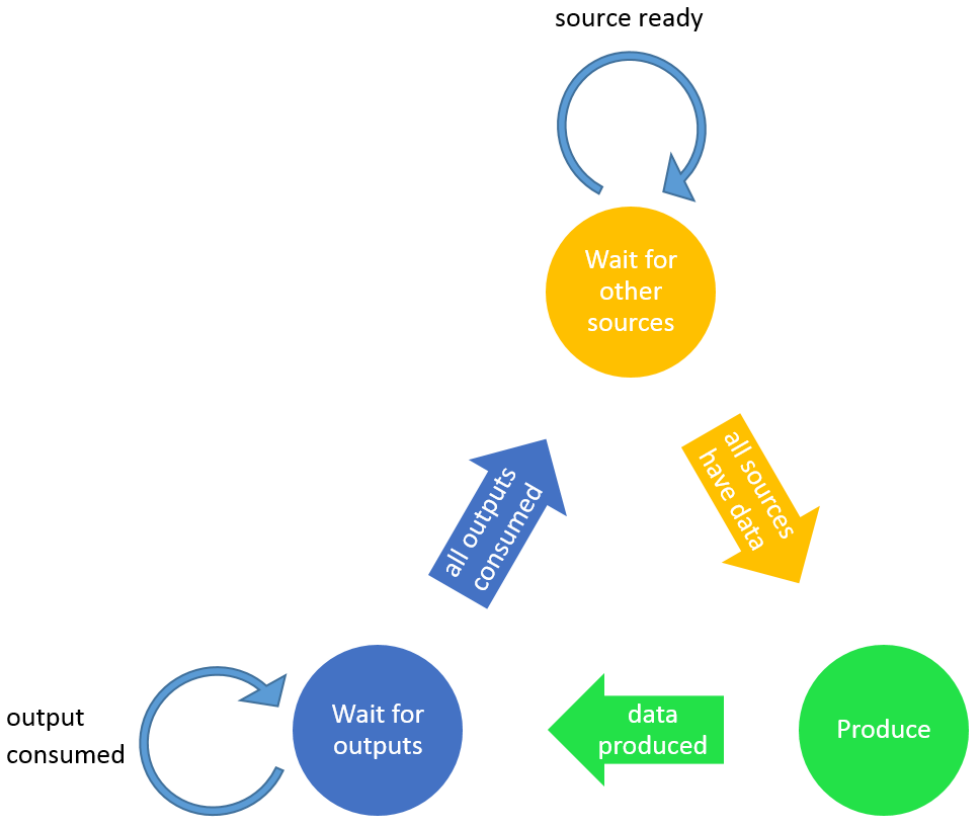


Figure 4.4: Source node wrapper state chart

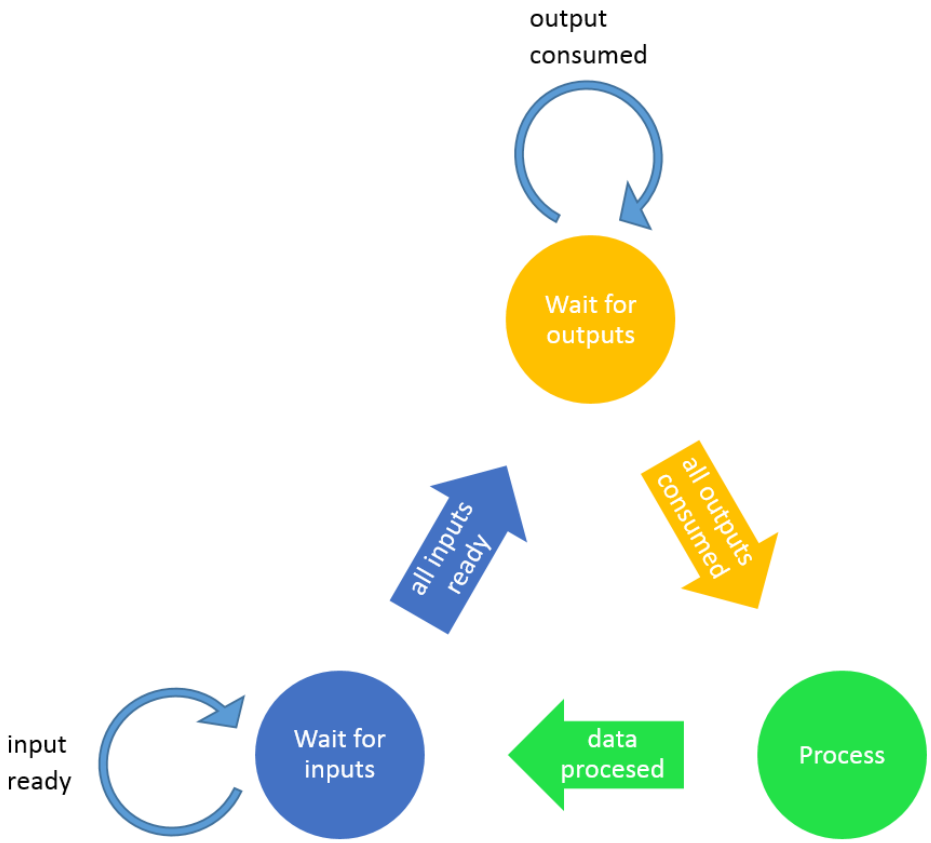


Figure 4.5: Processing node state chart

attached output pin wrappers notify, that the data they have been providing was successfully consumed. Data source wrapper starts to wait for other data sources to finish propagation of their provided data. If all sources wrappers reach this stage, they check if all can provide new data. If yes, they move to their initial state, producing new data. Otherwise, process of loading new data to data flow is stopped and processing logic starts to monitor for data flow to finish data processing. Figure 4.4 presents state chart for source node wrapper.

Processing node wrapper Processor node wrapper starts waiting for inputs to be ready to provide new data for processing. When all connected input pins notify about available data, processing node wrapper checks, if previously produced data have been already consumed. Processing node can not provide any new data, if previous one have not been copied by the ancestor nodes. Wrapper waits until all connected output pins notify, that

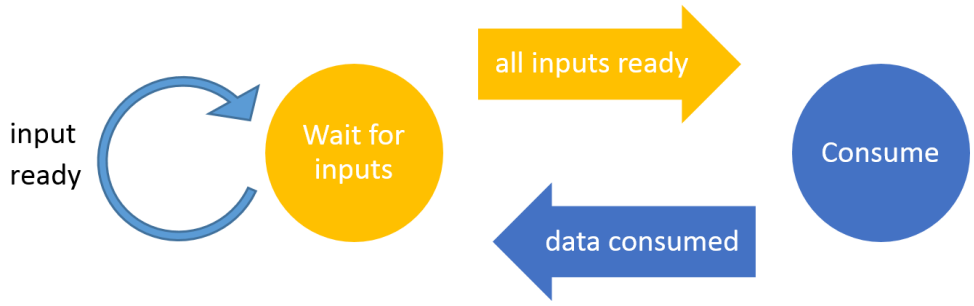


Figure 4.6: Sink node wrapper state chart

provided data have been properly consumed by all attached nodes. During the first run of processor node wrapper, output pins are already in their initial state, waiting for new data to propagate, therefore this step is immediately skipped forward and data processing can be done. In the next runs however, it might force processor node to wait with data processing, despite available input data, until output data is consumed. Figure 4.5 presents state chart for processing node wrapper.

Sink node wrapper Sink functionality is the simplest among all nodes. Sink node wrapper starts in an initial state, waiting for all input pins to be ready to provide new data for consuming. When all input pins notify that data is available, sink node process provided data and returns to its initial state, once again waiting for input data. Figure 4.6 presents state chart for sink node wrapper.

4.2.4 Executing processing logic

Executing defined processing logic is done through a dedicated *ModelRunner* class. It delivers methods for starting, stopping, pausing and resuming data flow processing. Additionally, it allows joining data processing, which stops current tread execution until data flow processing is finished. *ModelRunner* internally verifies data flow model structure according to its basic constrains for nodes and connections. If model is considered to be correct it is encapsulated with described wrappers to create its cloned version with embedded processing logic. Node wrappers offer functionalities to be run in separate threads. To start a model execution, it is required to deliver threads factory object. With its help *ModelRunner* tries to create required number of threads, one per node. Such simple approach might seem to be naive, as al-

ready explained in Section 3.2.3.4, because too many threads can slow down processing instead of accelerating it. As it will be shown later, this approach is completely acceptable, as it offers to manage processing power in an optimal way with just a small implementation improvements to processing logic.

When required number of threads has been obtained, nodes processing logic is launched within particular threads and data processing starts. Processing logic tracks continuously for data processing to finish, stopping threads execution and releasing their resources when suitable. Also possible failures in user implementation cause data flow to stop its execution and report errors. Threads joining data flow processing are waken up on data flow finish, to continue their execution. User can pause data flow execution, which would pause individual nodes processing logic execution after current data processing stage. User might resume processing, which will allow threads to continue executing their processing logic. Stopping data flow cause immediate stop of all processing threads and finish data flow in an undefined state.

4.3 Data flow plug-ins for MDE

Now we present how data flow processing is provided in MDE. We have developed two dedicated plug-ins. First of them provides basic data flow elements, model and processing logic in form of a service for MDE (*vdf*). Second one (*dfElements*) provides helper classes for universal sinks, sources and pins based on templates, which automatically translate information about accepted and provided data types to OW mechanism, description accepted by the MDE logic. Similarly to visualizers or parsers, data flow elements prototypes are registered within provided service to give centralized access to all processing elements. We present how particular processing logic has been implemented for delivered data flow model and runner.

4.3.1 Data exchange mechanism

To pass data efficiently MDE provides data representation in form of OWs. It is used in the processing logic to copy smart pointers to OWs encapsulating required data, instead of copying data itself. All output data is propagated in unmodifiable form (*const* modifier), therefore they have to be cloned explicitly, when particular functionality requires to modify them locally. Any processed data must be wrapped with OWs before propagation. Data processing requires unpacking encapsulated data from OWs. This mechanism

is also used for verification if particular pins can be connected, as each pin is now represented by one particular data type, described with OW. All processed types must be registered in application, so it is possible to verify different data types compatibility. When such mechanism is not sufficient for converting data from one representation to another, users can still provide custom processing nodes realizing required conversions, for example splitting 3D vector data to three independent scalar channels.

4.3.2 Optimal computational resources utilization

General purpose data flow processing library uses naive approach in threads utilization, allowing each thread to process particular node tasks and logic functionality. To address this problem and ensuring optimal processing resources utilization provided in MDE threading and job processing functionalities are used. Application thread pool is wrapped to behave as a threads factory for creating threads responsible for data flow processing logic execution. Instead of using all of created threads to process data flow operations, each node processing task is wrapped to a *Job* object and passed to *JobManager* for processing. Attached data flow processing logic is then suspended in an assigned to node processing logic thread, until particular *Job* is not finished. When *Job* is processed, attached data flow processing logic is resumed and continues executing data flow logic. With such simple improvement we have ensured, that data flow is utilizing optimally processing power and it process data in an standardized manner, along with other data processing tasks in MDE. One drawback of this approach is that creating complex data flows might require setting high value for maximal number of threads provided with *ThreadPool*, but as those threads are going to sleep most of the time it should not affect overall application performance.

4.3.3 Visual programming

Visual programming concept was proposed to simplify process of software development by manipulating graphically logic blocks, instead of writing their equivalent code. It was thought to speed up overall software creation process, but as it occur, changing programming languages and rising software complexity caused this idea to be abandoned. This approach has however been applied with success for other areas. Tools like:

- LabView (dedicated to scientist and engineers, `http://poland.ni.com/labview`),
- Blender (free image and video processing `http://www.blender.org`),

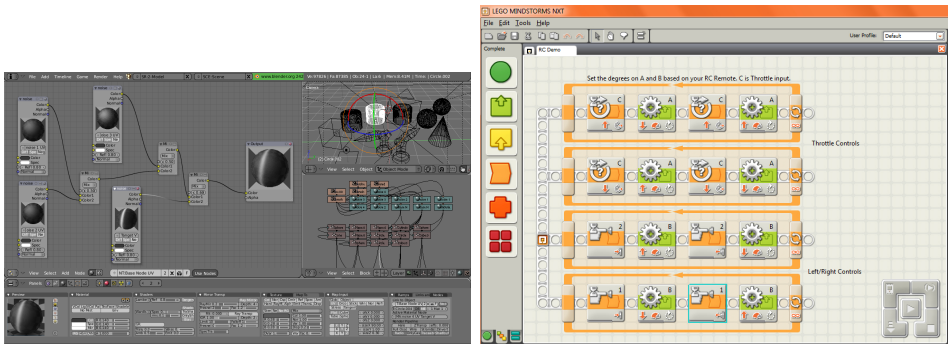
- Lego NXT-G (tools for programming autonomous robots, `http://mindstorms.lego.com/en-us/default.aspx`)

use extensively visual programming. This approach allows their users to create easily complex processing pipelines according to predefined model rules. With help of visual environments users not keen in programming can create dynamically specialized processing graphs realizing particular functionality (video rendering, robot steering) based on delivered solutions. This concept provides clear overview of processing schema, giving a good feeling, how data would be processed in the next steps. Figure 4.7 presents examples of different visual programming environments.

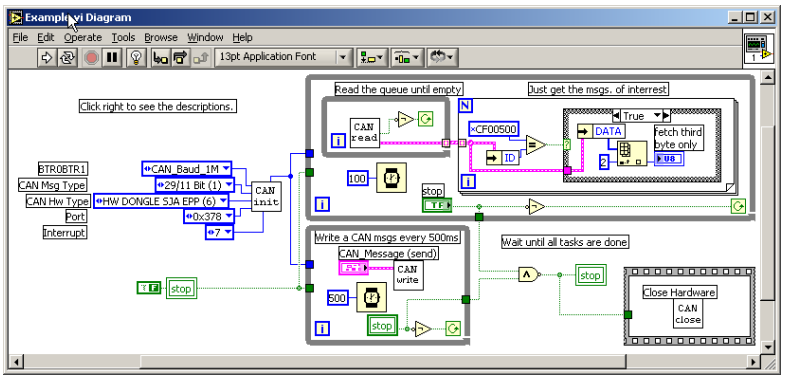
We decide to provide users with similar visual programming environment, despite processing model and logic, to simplify and speed up process of data flow creation. It allows users to dynamically create new data flows without need of writing new fragments of code and compilation procedure, based on delivered nodes functionality. They can now define and launch data flows without any knowledge about programming. Visual data flow environment supports user in creating data flow by presenting graphically available nodes. It guides users, how particular nodes can be connected according to basic model rules and data types compatibility. Required and dependent pins are marked graphically with different styles to point out user those places in the model, where connections are still required to make model complete. In the end any model verification failures are also presented to the user, with detailed description of elements and actions leading to fix those problems.

Visual programming introduces higher abstraction layer for model creation. New operations like merging and splitting nodes groups are proposed, allowing to create more complex algorithms and store them for further use, serializing internal groups connections and settings. Whole data flows can be stored and loaded to provided ready to use processing schema to run with different data sets. Visual data flow environment is oriented on users trying to process data based on the delivered operations with no programming skills at all. It does not limit however software developers who want to have full control over created model and execution procedure, introducing new, custom functionalities. Visual data flow is also perfect for scripting, making it even simpler to create and run models from a console with short commands.

For MDE visual data flow is delivered as a dedicated widget loaded with registered nodes, grouped according to their types for easier browsing. Simple drag‘n‘drop mechanism allows to instantiate particular node in a data flow model and position it on a scene. To connect nodes user have to click



(a) Blender (b) NXT-G



(c) LabView

Figure 4.7: Examples of visual programming environments
sources: `http://upload.wikimedia.org/wikipedia/commons/thumb/2/26/Working_with_Nodes_Blender.PNG/800px-Working_with_Nodes_Blender.PNG`
`http://www.letu.edu/opencms/export/sites/default/_Academics/Engineering/Media_Room/LEGO_toy_adaptations/programming.jpg`
`http://www.computer-solutions.co.uk/gendev/images/Example_LABview_diagram.png`

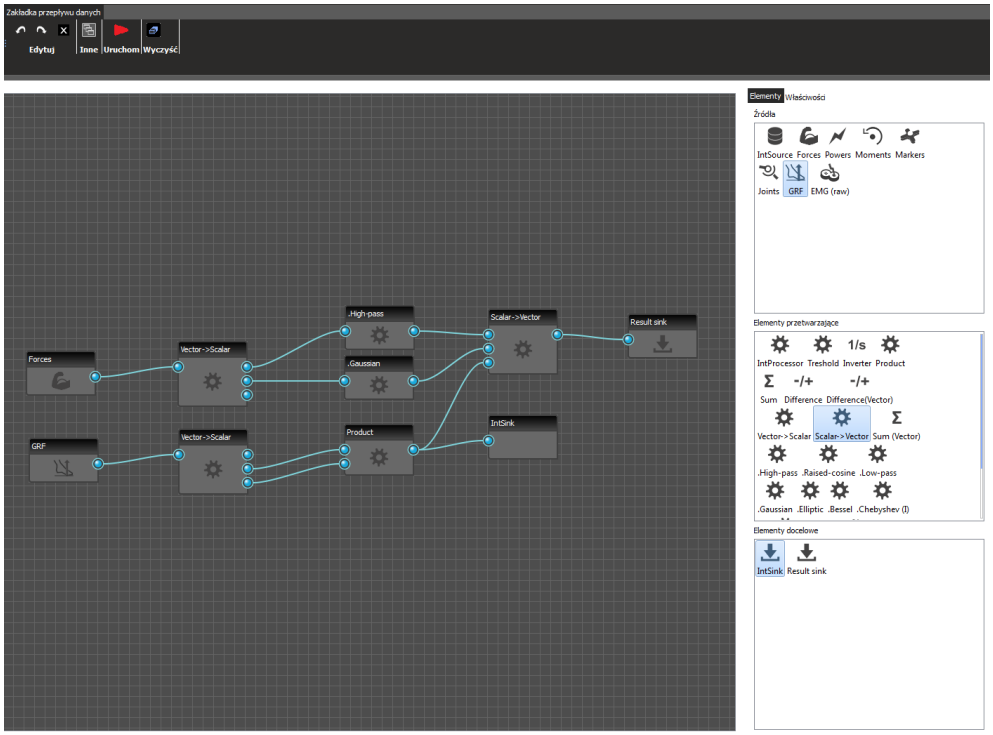


Figure 4.8: Visual data flow environemnt exmaple for MDE

Listing 4.6: Nodes configuration window interface

```
//! Node graphical configuration interface
class INodeConfiguration
{
public:
    //! Virtual destructor
    virtual ~INodeConfiguration() {}
    //! \return Configuration widget
    virtual QWidget* getConfigurationWidget() = 0;
};
```

particular pin, what starts connecting procedure. Proposed connection is draw from chosen pin to the actual cursor position on the scene. To finish connecting particular pins, user should move mouse cursor above desired pin to connect and release mouse button. If pins are compatible and no connectivity rules are violated a new connection is introduce to the model with selected pins. To remove connection, user must click chosen connection and press delete key on keyboard or press right mouse button and chose *Remove connection* option. It is also possible to select group of nodes and move them around the scene. Graphical connection representations are updating their appearance automatically. As there can be many nodes on a scene, user is supported in managing those nodes by disallowing stacking them, so that some of them might be covered by others, becoming invisible. When this situation occur, dragged node is moved back to its initial position and it is ensured, that newly placed nodes can not collide with each other. Service allows to create many visual data flows and manage them independently, but all of them are utilizing the same pool of computing resources. Running in parallel several data flows would extend their running time. Figure 4.8 present visual data flow environment for MDE.

For visual programming a simple interface can be implemented additionally for each node, to deliver its optional settings, that user might edit from the GUI. Listing 4.6 presents this configuration interface. Widget with settings is automatically handled by the visual programming environment in an uniform manner for all nodes, when user interacts with particular node.

Additionally, a simple node validation interface is proposed, that will be used to validate nodes configuration independently from model validation. Listing 4.7 presents this interface.

Nodes are registered with helper macros within particular plug-in. During registration client has possibility to deliver nodes extended description, used to present nodes to the user in a graphical manner in the nodes browser and

Listing 4.7: Nodes additional validation interface

```
//! Node validation interface
class INodeValidation
{
public:
    //! Virtual destructor
    virtual ~INodeValidation() {}
    //! \return True if node`s configuration is valid, otherwise false
    virtual bool isNodeValid() = 0;
    //! \return If node validation fails, short description can be given
    //! what is wrong
    virtual QString getErrorMessage() = 0;
};
```

on the scene (node instances). For each node an icon can be given, with a short node description. Listing 4.8 presents example node registration macro. It consists of two parts - first creates node group that is delivered with plug-in, later particular nodes are registered within this group. Dedicated macros for sources, sinks and processors are given. They verify if registered nodes provide required interfaces. Registering new node its type is given first, later name or a short description, then unique identifier and in the end an icon.

Listing 4.8: Macro for data flow node encapsulation within plug-in

```
VDF_SERVICE_BEGIN(ExampleNodesGroup, ↩
    "{C354FD3F-3559-4990-830D-57BA5E5BC813}")

    VDF_ADD_DATA_PROCESSOR(
        ExampleDataProcessor,
        "Example Data Processor",
        "{1143D447-2F76-458A-ADEB-4A669E487023}",
        QIcon(":/nodeIcon.png")
    );

VDF_SERVICE_END(ExampleNodesGroup)
```

4.4 Future work

We see a great potential in presented data flow processing, therefore many new ideas were introduced for its possible applications. The major development directions are:

- utilizing *JobManager* work-stealing scheduler,
- utilizing GPU computational power,

- scheduling and distributing work in clusters environment.

First solution is based on extended *JobManager* functionality, where new *Jobs* scheduler is proposed, allowing more flexible *Jobs* assignment for particular working threads. Based on data flow processing structure it would be possible to automatically schedule work in such a way, that a probability of optimal utilization of processor cache would be maximized, what should improve genera data flow performance.

Second idea requires extending functionality of *JobManager*, so that additional *Job* category could be introduced, representing GPU tasks. It could be easily done by adding internally second queue (or multiple queues) for this type of *Jobs*. Probably additional maintenance thread would be required to service GPU tasks queue processing, scheduling work to available GPU devices. Such *Jobs* might cover universal *OpenCL* (`http://www.khronos.org/opencl`) implementation or dedicated solutions for particular vendors like *CUDA* (`http://www.nvidia.com/object/cuda_home_new.html`) from *NVIDIA*.

Third improvement assumes to use data flow for initialization and control of computations in cluster environments. Such application might look as follows:

1. User connects to particular cluster.
2. Cluster configuration is read.
3. User builds processing graph.
4. Processing graph is translated to fit cluster capabilities and architecture.
5. All required resources are loaded to cluster and initialized on cluster nodes.
6. Data processing is started on cluster.

After the processing is finished, data might be collected back to user local computer to browse results and analyse them. There might be also some dedicated data source managing produced results at cluster side, not requiring to download them all to a local computer, as their volume might be too big. This approach is currently only a rough idea how data flow with visual environment might be used to automate and speed up cluster computing.

Chapter 5

Motion analysis

5.1 State of the art

Many multi-resolution tools were developed to support motion analysis. The background for those solutions are usually classical digital signal processing techniques. Great amount of those tools process orientation data, describing particular joints move as three uncorrelated signals represented by Euler angles. Quoted solutions consider usually data correlation in time domain between particular angles, not including joints hierarchical description in skeleton.

In [26] motion data smoothing with B-spline wavelet for unit quaternion is used. Similarly, in [27] based on the Daubechies wavelets (D10) smoothing of each component of a unit quaternion with soft and hard thresholding methods is proposed. Rotation smoothing is formulated as a non-linear optimization problem in [37]. A series of fairness functions defined on orientation data are used to derive smoothing operators. Spatial filters for orientation data are proposed in [38, 39]. Similar solution, based on the digital filter bank technique, can be found in [10]. In [18] a low-pass filter is applied to the estimated angular velocity of an input signal to reconstruct a smooth, angular motion by integrating filter responses.

Multi-resolution techniques are also applied for effective feature detection. Extraction of easily comparable features, unique and robust for motion data is one of the main goal of motion analysis. Such features are fundamental for motion phases description with set of reliable parameters, allowing to predict human motion based on previous observations. An example for such approach can be found in [7], where the Haar wavelet transform for the

extraction of appropriate features from kinematic data (joint trajectories) is used. Multi-scale smoothing of input data is combined with with an features extraction, which characterize transitions between various, well defined motion phases.

Motion data compression is another application for multi-resolution techniques. In general, presented solutions remove high frequencies details of the signals, which usually contains noise. This is done especially for the joints not important for particular move. The joints importance is defined based on a joint position in the skeleton hierarchy and motion type. To compress skeletal motion data lifting scheme blocks implementing the cubic interpolating bi-orthogonal wavelet basis are used in [6]. With such wavelet transform temporal coherence is exploited. In [40] it is proposed to combine forward kinematics and wavelet transform. The cubic B-Spline wavelets are used in [2]. Usage of an anisotropic diffusion process for smoothing all degrees of freedom of a motion windows, which are later divided into segments at feature discontinuities, is proposed in [41]. The cubic Bezier curves are used to approximate each degree of freedom for particular segments. Application of the anisotropic diffusion process smooths low-frequency parts of the data and preserve perceptually important high-frequency parts of the data. In [5] high level of compression comes at the expense of smoothing high frequency details in the motion. The short clips of motion sentences can be also represented as cubic Bezier curves and clustered principal component analysis can be used to reduce their dimensionality. This technique utilizes temporal coherence (fitting Bezier curves) and correlation between all degrees of freedom.

Our goal is to use multi-resolution tools for motion data in the quaternion representation, to capture rotation data correlations in the time domain, instead of analysing independent signals for three Euler angles. We believe that this is the right approach, as quaternions algebra better describes the nature of rotations. With multi-resolution techniques we want to decompose motion data to a simpler representation, which would allow reliable motion features description and extraction, being an input for further motion analysis tools.

5.2 Quaternions

5.2.1 Introduction

Quaternions [52, 58, 24] are the number system extending capabilities of the complex numbers. They describe vectors relations in 3D space - rotation and

scaling - as follows:

$$\frac{\vec{v_a}}{\vec{v_b}} = q_{ab} \rightarrow \vec{v_b} q_{ab} = \vec{v_a}$$

where q_{ab} denotes quaternion and $\vec{v_a}, \vec{v_b} \in \mathbb{R}3$. Quaternions are represented as a pair of a 3D vector (imaginary part, Img) and scalar (real part, Re) according to Equation 5.1. Also quaternion interpretation as a 4D vector is used.

$$q = [\vec{v}, s] = [(x, y, z), w] \tag{5.1}$$

There are several well defined quaternions operations and properties:

multiplication There are two methods for multiplying quaternions:

- First one uses a 4D vector quaternion representation and produces simple component by component multiplication as presented in Equation 5.2.

$$q_a * q_b = \vec{q_a} \cdot \vec{q_b} = [x_a * x_b, y_a * y_b, z_a * z_b, w_a * w_b] \tag{5.2}$$

  It does not have any geometrical explanation and in general should not be used.

- Second approach defines quaternion multiplication as an geometric operation, allowing to combine different vector transformations. It is done according to the Hamilton product presented in Equation 5.3.

$$\begin{aligned} q_a * q_b &= [x_a, y_a, z_a, w_a] * [x_b, y_b, z_b, w_b] = \\ &= [x_a * x_b - y_a * y_b - z_a * z_b - w_a * w_b, \\ &\quad x_a * y_b + y_a * x_b + z_a * w_b - w_a * z_b, \\ &\quad x_a * z_b - y_a * w_b + z_a * x_b + w_a * y_b, \\ &\quad x_a * w_b + y_a * z_b - z_a * y_b + w_a * x_b] \end{aligned} \tag{5.3}$$

  It has to be noted, that in this case multiplication is not commutative, so in general:

$$q_a * q_b \neq q_b * q_a$$

  This is a very important property of quaternions algebra. Quaternion division is realized with so defined multiplication and quaternion inverse (reciprocal) operation.

conjugate Quaternion conjugate is defined as:

$$q^* = [\vec{v}, s]^* = [-\vec{v}, s]$$

It represents a quaternion with a negated vector part and unchanged scalar value. Conjugate can be used to extract scalar and vector part by proper addition and subtraction, followed by a division with a factor of two. Applying conjugate twice will return the original quaternion:

$$(q^*)^* = [-\vec{v}, s]^* = [\vec{v}, s]$$

norm This property is usually interpreted as a length of a quaternion. Equation 5.4 presents norm definition. It is a square root of quaternion dot product with its conjugate.

$$||q|| = \sqrt{q * q^*} = \sqrt{q^* * q} = \sqrt{x^2 + y^2 + z^2 + w^2} \tag{5.4}$$

reciprocal (inverse) With help of this property it is possible to introduce quaternion division. Equation 5.5 defines quaternion reciprocal - a normalized conjugate.

$$q^{-1} = \frac{q^*}{||q||} \tag{5.5}$$

5.2.2 Unit quaternions

There is a well defined subset of quaternions, called *unit* quaternions. This subset is denoted as $\mathbb{H}_1$. Its elements are quaternions which have the following property:

$$\forall_{q \in \mathbb{H}_1} ||q|| = 1$$

It is possible to introduce an additional representation for unit quaternions, which makes particular operations, presented later, easier to perform. Equation 5.6 presents unit quaternion trigonometric form.

$$\begin{aligned} q = \left[\vec{v} sin(\theta), cos(\theta)\right], \\ \vec{v} \in \mathbb{R}3, \\ \theta \in [-\pi, \pi] \end{aligned} \tag{5.6}$$

5.2.3 Quaternion functions

It is possible to define additional functions for quaternions. They allow to map quaternions from $\mathbb{S}4$ hypersphere to $\mathbb{R}3$ space and backward. They also introduce possibility of applying particular vector transformation multiple times (composition, power function), just a quotient of a transformation or interpolation between pairs of transformations.

logarithm This operation transforms quaternion from $\mathbb{S}4$ hypersphere to $\mathbb{R}3$ space (quaternion tangent space) as presented in Equation 5.7. After the transformation unit quaternion is represented by its vector part, scaled according to its angle in trigonometrical representation.

$$log\left(q\right) = log\left(\left[\vec{v}sin(\theta), cos(\theta)\right]\right) = \left[\theta\vec{v}, 0\right] \tag{5.7}$$

exponent It is an inverse operation for logarithm. It allows to transform a vector from $\mathbb{R}3$ space back to $\mathbb{S}4$ hypersphere following the Equation 5.8.

$$exp\left(q\right) = \left[\vec{v}sin(\theta), cos(\theta)\right], q = \left[\theta\vec{v}, 0\right], \theta \in \mathbb{R}, \vec{v} \in \mathbb{R}3 \tag{5.8}$$

power Based on logarithm and exponent functions it is possible to define quaternion power function. Power function for quaternions allows to combine particular vector transformation multiple times or just applying some quotient of this transformation. Equation 5.9 present the definition for power function. This operation is extensively used for quaternion interpolation.

$$q^t = exp\left(t * log\left(q\right)\right), t \in \mathbb{R} \tag{5.9}$$

Despite elaborated operations there is also a well defined differential calculus for quaternions, allowing more detailed and complex rotations analysis. We do not present it here, as it is not relevant for presented solutions, but we find it very important to mention it. Further information about differential calculus for quaternions can be found in [22].

5.2.4 Rotations

Quaternions use simpler representation for rotations than Euler angles. Instead of three independent rotations along particular axis, that Euler angles do, quaternions perform single rotation by a given angle, around particular

axis, going through the centre of the coordinate system with well defined rotation direction, according to the right hand rule. This can be explicitly connected with quaternion representation, where a 3D vector is given, representing axis direction, and the scalar part, representing rotation angle. The following actions have to be made to rotate a vector $\vec{v}$ around a given axis $\vec{r}$ by an angle α:

1. transform vector $\vec{v}$ to a quaternion form according to Equation 5.10,

$$\forall_{\vec{v}\in\mathbb{R}3} q_v = [\vec{v}, 0] \tag{5.10}$$

2. create rotation quaternion q_r for the given axis r and rotation angle α with Equation 5.11,

$$q_r = \left[\vec{v} sin\left(\frac{\alpha}{2}\right), cos\left(\frac{\alpha}{2}\right)\right] \tag{5.11}$$

3. perform rotation in quaternion space as presented in Equation 5.12

$$q_v^{`} = q_r q_v q_r^{-1} \tag{5.12}$$

4. extract resulting vector from obtained quaternion as described in Equation 5.13

$$v^{`} = Img\_part\left(q_v^{`}\right) \tag{5.13}$$

It has to be mentioned, that quaternion q and $-q$ represent the same rotation. This property is called antipodality. This dual nature of quaternions can be easily observed applying the right hand rule, switching both axis and angle direction to opposite values. Additionally, as quaternions multiplication allows to combine more vector operations at once, it is also possible to apply more than one rotation to a vector, keeping in mind that multiplication is not commutative. Equation 5.14 presents general structure for applying more than one rotation to a vector according to presented scheme.

$$\begin{aligned} q_n &= q_1 q_2 ... q_n \\ qi_n &= q_n q_{n-1} ... q_1 \\ q_v^{`} &= qi_n q_v q_n^{-1} \end{aligned} \tag{5.14}$$

Quaternion q_n represents composition of rotation quaternions in a well defined order. Quaternion qi_n represents composition of the same rotation quaternions in reverse order. Other symbols have the same meaning

as already presented. Resulting quaternion $q_v^{'}$ will represent vector rotated according to provided rotation quaternions in their specific order. This approach can be generalised also for a combination of rotation and scaling transformations.

Representing rotations and orientations with quaternion has many advantages over classical approach with Euler angles, although both representations are equivalent. Quaternions are free from gimbal-lock problem, which may occur using Euler angles, when one degree of freedom can vanish. Also, quaternion representation is uniquely representing rotation, when using Euler angles different notations and combinations are possible. Mangling various Euler angle notations and managing them can introduce serious problems when analysing motion data. Quaternions can be easily transformed to rotation matrices and Euler angles. Different rotation representation, their advantages and disadvantages with conversion from one representation to the others are presented in [14, 71, 72].

5.2.5 Interpolation

Pointing out quaternion advantages over Euler angle, quaternion interpolation techniques have to be mentioned. With Euler angles three independent data series have to be interpolated, in general loosing their correlation. As unit quaternions are compact representation for rotations it is easier to interpolate them considering their spacial properties.

$$\begin{aligned} t &\in [0;1] \\ interpolate\,(q_a, q_b, t=0) &= q_a \\ interpolate\,(q_a, q_b, t=1) &= q_b \end{aligned} \tag{5.15}$$

Recalling basic interpolation properties in Equation 5.15, the following quaternion interpolation methods are available, satisfying those rules:

lerp Although this technique treats quaternion as a 4D vector, it can be successively used for simple quaternion interpolation. After interpolating rotations additional normalization is required to maintain proper rotation representation in form of unit quaternion. Equation 5.16 presents, how to interpolate quaternions with this approach.

$$\begin{aligned} lerp\,(q_a, q_b, t) &= q_a + (q_b - q_a)\,t = \\ &= [x_a + (x_b - x_a)\,t, y_a + (y_b - y_a)\,t, z_a + (z_b - z_a)\,t, w_a + (w_b - w_a)\,t] \end{aligned} \tag{5.16}$$

slerp This interpolation technique does not require results to be normalized. *Slerp* values follow shortest path on a great arc on $\mathbb{H}1$ hypersphere between given quaternions according to Equation 5.17.

$$slerp\left(q_a, q_b, t\right) = q_a\left(q_a^* q_b\right)^t \tag{5.17}$$

squad Applying Bezier curves interpolation idea to quaternions we obtain *squad* interpolation. It requires four quaternions for interpolation, two of them are used as a interpolation range, with other two used to generate control points ensuring smooth and differentiable interpolation curve. We now index quaternions to make their role clear in the interpolation. Equation 5.18 presents complete description for this technique. Quaternions q_i and q_{i+1} are used as a key frames and quaternions s_i and s_{i+1} are control points. Equation 5.19 describes how control points are generated.

$$squad\left(q_i, q_{i+1}, s_i, s_{i+1}, t\right) = slerp\left(slerp\left(q_i, q_{i+1}, t\right), slerp\left(s_i, s_{i+1}, t\right), 2t\left(1-t\right)\right) \tag{5.18}$$

$$s_i = q_i exp\left(-\frac{log\left(q_i^{-1} q_{i+1}\right) + log\left(q_i^{-1} q_{i-1}\right)}{4}\right) \tag{5.19}$$

Detailed information about particular interpolation methods and their derivation can be found in [11]. It presents the properties and effects of interpolation methods in a well defined and visual manner.

5.3 Multi-resolution analysis

5.3.1 Introduction

The main idea of the multi-resolution analysis is to represent a signal in form of a coarse to fine hierarchy. Analysed signal is decomposed into general data description (global pattern of the signal) and a hierarchy of signal details (coefficients). Obtained multi-resolution representation can be a base for many already presented algorithms, tools and applications. This type of signal representation allows much better signal behaviour understanding by presenting its overview at different detail levels in time domain, in comparison to frequency characteristic of signal provided by the Fourier analysis [56]. Main tool for multi-resolution analysis is the wavelet transform [46].

It has very strong mathematical and analytical background, providing well defined procedure for data decomposition. Although wavelet usage offers all multi-resolution features for decomposing signal, it is based on complex mathematical theorems and therefore sometimes it is hard to ensure all its requirements. Also very often proposing proper mother wavelet function is not a trivial task and proving that so proposed mother wavelet function fulfils all wavelet transformation requirements might be hard or impossible. To dedicate those problems a new approach to the wavelet transform and multi-resolution analysis has been proposed. It is called lifting scheme, also know as a second generation wavelets [31].

5.3.2 Lifting scheme

The lifting scheme [62, 63] is a simple, yet powerful tool to construct the wavelet transform. The main advantage of this solution is the possibility of building the wavelet analysis for complex structures of data (irregular samples, curves, surfaces), additionally keeping three valuable properties [57, 13, 64]:

- speed,
- good ability of approximation,
- low memory consumption.

Wavelets proposed by the lifting scheme, in contradiction to classical wavelet transform, are not necessarily translated and dilated versions of one function (mother wavelet). In this meaning lifting scheme also considers non-linear and data-adaptive multi-resolution decompositions. It is rather engineering approach to the wavelet transform and multi-resolution analysis, as a general lifting scheme is built up with three well defined and simple operations:

Split splits input dataset into two disjoint sets of even indexed samples and odd indexed samples. The definition of lifting scheme does not impose any restrictions on how the data should be split nor on the relative size of each subset.

Predict predicts samples with odd index based on even indexed samples. Next the odd indexed input value is replaced by the difference (detail) between the odd value and its prediction (detail, coefficient).

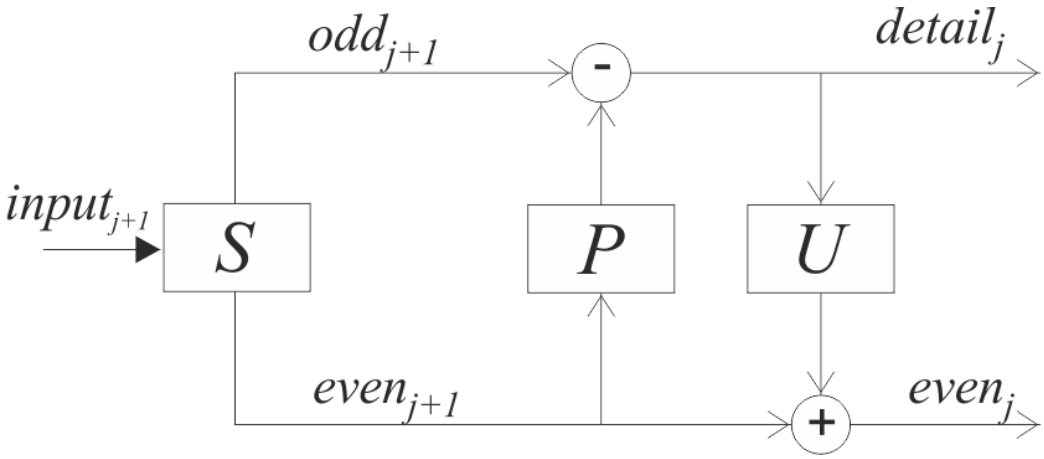


Figure 5.1: Lifting scheme forward transform

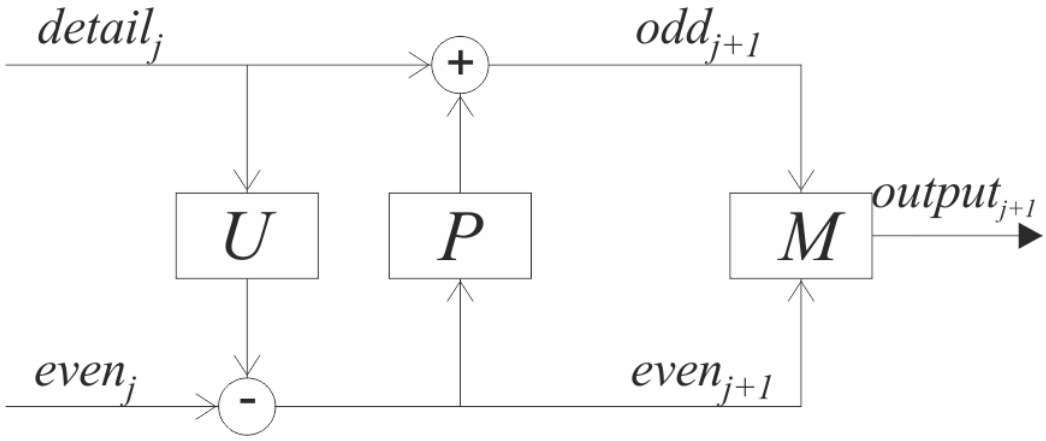


Figure 5.2: Lifting scheme inverse transform

Update updates the output, so that coarse-scale coefficients have the same average value as the input samples. This step is necessary for stable wavelet transform [31].

Overview of the lifting scheme can be seen in Figure 5.1. Decomposition algorithm for forward transform is run recursively. Updated even samples are used as an input for next algorithm run, until single, global signal average value is reached, with signal coefficients for all decomposition levels. All that calculations can be performed in-place. In all stages input samples for next stage can be overwritten by output of the current step, which means no additional buffers for temporary data are required. Inverse transform is easy to find by reversing the order of operations and flipping the building block operation signs (inverting their operations). It allows to reconstruct signal to

its original representation, when no modifications were made to decomposed details and average, or to see, how such modifications are propagated and affect the signal after the reconstruction.

Figure 5.2 presents an overview for lifting scheme inverse transform. Split operation is exchanged by a *merge* operation, which reorders samples from the current level and joins two separate sets of odd and even samples into one set, where proper input samples order is guaranteed to be maintained during the reconstruction.

5.4 Lifting schema for quaternions

We propose a new tool for motion analysis based on the lifting schema and quaternion representation for joints rotation data. We present building blocks of various lifting schemes for quaternion signals. Based on so proposed motion analysis tool, possible applications are pointed out with their description. We believe that combination of multi-resolution techniques and quaternions provide informative motion description allowing to realize more complex tasks in the field of motion analysis.

5.4.1 Rotation average value

There have been proposed many different interpretations and definitions for rotation (quaternion) averages [51, 45, 43, 20, 25]. This is a very wide topic, covering proper distance metric that must be chosen to fit particular applications. In general, there are two approaches to define rotation average value for a series of rotations:

Euclidean This approach to rotation average is defined as an optimization problem, where average rotation is obtained as a rotation matrix minimizing cumulative Frobenius norm (Euclidean distance) of differences between current average value and given rotations in matrix representation. Equation 5.20 presents Frobenius norm of matrix R with dimensions $m \times n$. Equation 5.21 describes proposed average concept, where R_F denotes resulting average rotation matrix for a series of N rotation matrices R_i.

$$||R||_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|r_{ij}^2|} \tag{5.20}$$

$$\overline{R}_F = argmin_R \sum_{i=1}^{N} ||R_i - R||_F^2 \tag{5.21}$$

Riemannian In Riemannian approach rotations mean is formulated as the angle-based optimization problem, where an arc length between given rotations and their average should be minimized. A dedicated distance measure is used to calculate arc-length between particular rotation matrices as presented in Equation 5.22. Based on this distance measure a general solution to the problem is presented in Equation 5.23. It is a matrix representation for the *slerp* interpolation, where log function is expanded as presented in [45], which seams to be generally a better approach to the rotation average than Euclidean version.

$$d_R(R_1, R_2) = \frac{1}{\sqrt{2}} ||log\left(R_1^T R_2\right)||_F \tag{5.22}$$

$$\overline{R}_R = argmin_R \sum_{i=1}^{N} ||log\left(R_1^T R_2\right)||_F \tag{5.23}$$

As already mentioned, for multi-resolution analysis with the lifting scheme average value of the input signal must be maintained by its lower resolution representations for all algorithm steps. Average value interpretation is not important in this case and averaging operator might be defined in various ways to fit particular lifting scheme applications. We decide to use straight forward averages, fitting best presented quaternion interpolation techniques. Proposing lifting schemes prediction and update blocks we have concentrated on guaranteeing constant signal average value over all resolutions and not on the average operation interpretations. To simplify analysis and proving that proposed averages fill this condition we assume that a simple recursive procedure for calculating average value of a signal according to a chosen average operation maintains constant average for all signal resolutions. Now its description is given.

5.4.1.1 Recursive average

We assume that usage of the following procedure for calculating rotation signal average according to provided averaging operation for two nearby samples, maintains signal average value at all resolutions:

Algorithm 5.1: Average recursive algorithm

```
Data: inputSignal, averageFunction
Result: signalAverage
int half = inputSignal.size() / 2;
while half != 0 do
    for i=1 to half do
        inputSignal[i] = averageFunction(inputSignal[2(i-1)],
        inputSignal[2(i-1)+1]);
    end
    half =/ 2;
end
return inputSignal[0];
```

It can be noticed, that with such average approach, obtaining average value for the current resolution requires signal lower resolution average value to be calculated first. Algorithm continues until single value is obtained, which represents signal global average value. We were following this rules designing update blocks for proposed lifting schemes. One limitation of so defined average procedure is that the averaged signal must contain number of samples equal to a power of two. This might require truncating signal samples count to the greatest power of two, smaller or equal to the initial samples number, or extending signal with some neutral and artificial values, until number of elements reach the closest power of two greater than initial number of elements.

5.4.2 Proposed lifting schemes

We present the concepts of six different lifting schemes. Provided algorithms are based on general quaternion interpolation techniques and properties. Developed solutions are divided into different categories, based on the amount of samples used for prediction step. Proposing those lifting schemes we tried to treat quaternions as 4D vectors to verify, if this approach might be used successfully with multi-resolution tools for rotations, as they explore in general local signal differences.

Before we give detailed equations for particular lifting scheme blocks we must introduce used nomenclature for clarity. Odd and even samples are described as:

- o_i^j,
- e_i^j

where i index describes particular sample index in group of odd or even samples after split operation (relative order of samples before splitting is maintained). Index j describes the level of the signal resolution. Higher resolutions have greater j values with 0 representing global signal average value. Additionally, as split operator divides samples on odd and even ones, analysed signal must contain number of samples described with Equation 5.24. It overlaps the condition for proposed recursive average value calculation procedure.

$$\begin{aligned} n &= 2^k, \\ k &\in \mathbb{C} \end{aligned} \tag{5.24}$$

5.4.2.1 Single sample prediction

Linear Haar lifting scheme The Haar approach to the lifting scheme is the simplest one to design. It uses nearby *even* samples to approximate *odd* samples values. Quaternions are treated as 4D vectors, therefore following steps can be defined:

- predict

$$o_i^j = o_i^{j+1} - e_i^{j+1}$$

- update

$$e_i^j = e_i^{j+1} + \frac{o_i^j}{2}$$

Analogically, backward transform (signal reconstruction given the coefficients and coarse signal) is given with equations:

- undo-predict

$$o_i^{j+1} = o_i^j + e_i^{j+1}$$

- undo-update

$$e_i^{j+1} = e_i^j - \frac{o_i^j}{2}$$

After each transformation all quaternions must be normalized to represent valid rotations. Simple *lerp* interpolation is used for recursive average value calculations. It can be noticed indirectly in update step, where firstly original odd value is retrieved from even sample used to predict the odd sample and stored difference, and later those values are averaged. After simplifications only even sample and calculated detail are used. For all following transforms idea is similar, therefore we would only mention, which average method is used, skipping the derivations for update steps to show this behaviour explicitly.

Quaternion Haar lifting scheme In contradiction to previous schema, now proper quaternion algebra is used to describe quaternion differences. Proposed forward transform can be written as:

- predict

$$o_i^j = o_i^{j+1} * \left(e_i^{j+1}\right)^{-1}$$

- update

$$e_i^j = \left(o_i^j\right)^{0.5} * e_i^{j+1}$$

Inverting operations with respect to quaternion algebra, the following backward transform can be obtained:

- undo-predict

$$o_i^{j+1} = o_i^j * e_i^{j+1}$$

- undo-update

$$e_i^{j+1} = \left(o_i^j\right)^{-0.5} * e_i^j$$

No quaternion normalization need be done, as quaternion algebra on unit quaternions guarantees to produce unit quaternion results. Order of multiplying quaternions should be underlined, as already mentioned - this is not a commutative operation in quaternion space in general. Introduced average value is analogical to the linear algebra version - power function is used to retrieve the quotient (0.5) of composition for rotation pairs (average).

5.4.2.2 Two samples prediction

Now lifting schema using two nearest samples for prediction block are presented.

Linear quaternion lerp lifting scheme In this approach lerp interpolation is used, representing quaternions as 4D vectors. Forward transform equations are:

- predict

$$o_i^j = o_i^{j+1} - \frac{e_i^{j+1} + e_{i+1}^{j+1}}{2}$$

- update

$$e_i^j = e_i^{j+1} + \frac{o_i^j + o_{i+1}^j}{4}$$

Backward transform equations are written as:

- undo-predict

$$o_i^{j+1} = o_i^j + \frac{e_i^{j+1} + e_{i+1}^{j+1}}{2}$$

- undo-update

$$e_i^{j+1} = e_i^j - \frac{o_i^j + o_{i+1}^j}{4}$$

Once again, after each transformation all quaternions must be normalized to represent valid rotations, as linear algebra is applied instead of proper quaternion algebra. Lerp interpolation is used for pairs of nearby samples to calculate signal average value. To retrieve original odd samples values for average calculation, more complicated computations must be done in comparison to Haar transform due to an applied prediction step, involving two even samples to estimate surrounded odd sample.

5.4.2.3 Slerp lifting scheme

Applying proper quaternion interpolation technique, considering quaternion properties and algebra, the following lifting scheme blocks are obtained:

- predict

$$o_i^j = o_i^{j+1} * slerp\left(e_i^{j+1}, e_{i+1}^{j+1}, 0.5\right)^{-1}$$

- update

$$e_i^j = slerp\left(o_i^j, o_{i+1}^j, 0.5\right) * e_i^{j+1}$$

Similarly, inverse transform is defined as follows:

- undo-predict

$$o_i^{j+1} = o_i^j * slerp\left(e_i^j + e_{i+1}^j, 0.5\right)$$

- undo-update

$$e_i^{j+1} = slerp\left(o_i^j, o_{i+1}^j, 0.5\right)^{-1} * e_i^j$$

As quaternion algebra is applied there is no need to apply additional results normalization. For average we also use *slerp* interpolation.

5.4.2.4 Four samples prediction

Squad lifting scheme Based on *squad* interpolation, the following forward transformations can be given:

- predict

$$o_i^j = o_i^{j+1} * squad\left(e_i^{j+1}, e_{i+1}^{j+1}, s_i, s_{i+1}, 0.5\right)^{-1}$$

- update - In this case update step could not be solved analytically in quaternion space, as *squad* control points calculations mix quaternion algebra and 4D vector linear algebra. It prevented finding algebraical solution to retrieve original odd sample value from calculated detail and surrounding even samples used for prediction. We have dedicated a new approach to to this problem, presented in the next lifting scheme approach. Based on obtained analytical solution for presented quaternion

tangent space interpolation, we propose to use the following equation for update step, through an analogy to the tangent space solution:

$$e_{i+1}^{j} = \left(o_i^j\right)^{0.5} e_{i+1}$$

To perform signal reconstruction the following equations should be used:

- undo-predict

$$o_i^{j+1} = o_i^j * squad\left(e_i^{j+1}, e_{i+1}^{j+1}, s_i, s_{i+1}, 0.5\right)$$

- undo-update

$$e_i^{j+1} = \left(o_i^j\right)^{-0.5} e_i$$

This is a new solution, not presented in our previous works. Proposed test will verify its correctness and applicability for various tasks and compare obtained results with other lifting schemes.

Tangent space lifting scheme To address the problem of designing forward transform for *squad* interpolation, we propose a new approach to use this quaternion interpolation technique as a prediction block for the lifting scheme. The idea is to transform *squad* operation to the quaternion tangent space - $\mathbb{R}3$ - and perform all required operation there. After that, the signal is transformed back to its original quaternion form. Converting values from quaternion space to $\mathbb{R}3$ and backward can be done with *log* and *exp* functions, as already presented. Question arise how to transform *slerp* and *squad* operations, so they operate on $\mathbb{R}3$ vectors (points).

Starting description of new lifting scheme we want to show, that for any pair of unit quaternions q_a and q_b the following equation is always true:

$$slerp\left(q_a, q_b, 0.5\right) = \frac{lerp\left(q_a, q_b, 0.5\right)}{||lerp\left(q_a, q_b, 0.5\right)||}$$

Figure 5.3 presents lerp interpolation along circle chord and slerp interpolation along circle curve. We can further reformulate this equation by expanding *slerp* to obtain such form:

$$slerp\left(q_a, q_b, 0.5\right) = q_a\left(q_a^* q_b\right)^{0.5} = q_a q_a^{*0.5} q_b^{0.5} = q_a q_a^{-0.5} q_b^{0.5} = q_a^{0.5} q_b^{0.5} = \left(q_a q_b\right)^{0.5}$$

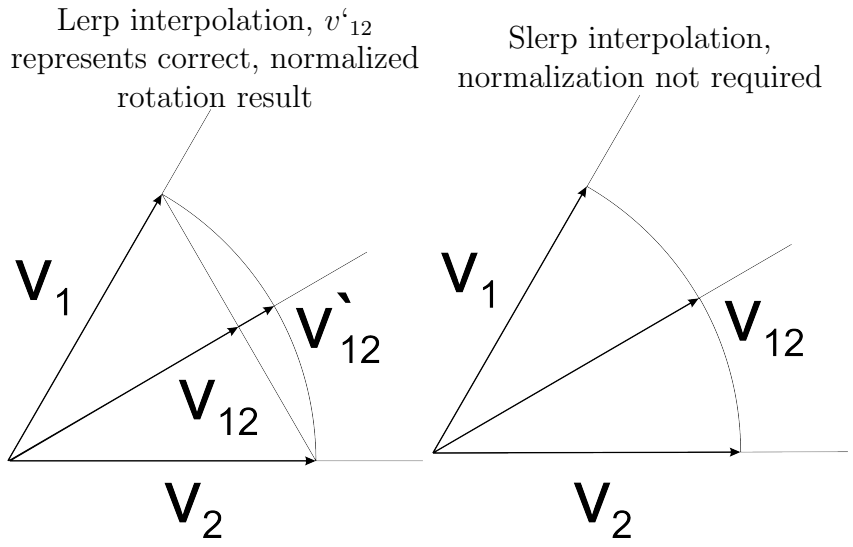


Figure 5.3: Lerp and slerp rotations interpolations comparison

Applying logarithm and exponent functions we can obtain:

$$slerp\left(q_a, q_b, 0.5\right) = exp\left(log\left(q_a q_b\right)^{0.5}\right) = exp\left(0.5 * log\left(q_a q_b\right)\right) = exp\left(\frac{log\left(q_a\right) + log\left(q_b\right)}{2}\right) \tag{5.25}$$

Equation 5.25 shows, how to do the *slerp* interpolation in the quaternion tangent space, as all operations are done according to logarithm values and obtained result is transformed with exponent function to the quaternion space. Applying it for *squad* we can perform this type of interpolation in $\mathbb{R}3$ space, based on simple vector linear algebra, and then return back to the quaternion space. Now a description of forward transform in the quaternion tangent space is given.

Firstly, we introduce an average value for a signal in the quaternion tangent space according to proposed method for averaging transformed signal values. We propose simple average method for a series of points in $\mathbb{R}3$, which is later transformed to the quaternion space, as presented in Equation 5.26.

$$avg = exp\left(\frac{1}{n}\sum_{i=1}^{n} log\left(q_i\right)\right) \tag{5.26}$$

Secondly, the *squad* interpolation is based on applying Bezier curve interpolation for $\mathbb{H}_1$ hypersphere. As signal is transformed to $\mathbb{R}3$, we propose to use classical Bezier interpolation, where for any pair of points p_1, p_2 and

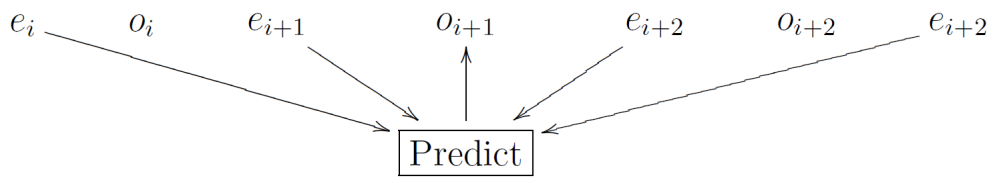


Figure 5.4: Prediction diagram for squad interpolation in tangent space

control points X, Y ($p_1, p_2, X, Y \in \mathbb{R}3$) interpolated points are described with Equation 5.27. We already show how *slerp* can be calculated in the tangent space, therefore introducing it to *squad* in the tangent space provides straight forward analogy to the following equation:

$$lerp\left(p_1, p_2, h\right) = lerp\left(lerp\left(p_1, p_2, h\right), lerp\left(X, Y, h\right), 2h\left(h-1\right)\right), \quad whereh \in [0;1] \tag{5.27}$$

We obtain average value for pair of points in $\mathbb{R}3$ with so proposed interpolation according to:

$$lerp\left(p_1, p_2, 0.5\right) = 0.5\left(0.5\left(p_1 + p_2\right) + 0.5\left(X + Y\right)\right) = \frac{p_1 + p_2 + X + Y}{4} \tag{5.28}$$

For smooth Bezier curve interpolation values X, Y are obtained according to the following equations for any pair of interpolated points p_i and p_{i+1}:

$$X = p_i + \frac{1}{4}\left(p_{i+1} - p_{i-1}\right)$$

$$Y = p_{i+1} - \frac{1}{4}\left(p_{i+2} - p_i\right)$$

Equations for control points are analogical to *squad* control points presented in Equation 5.19. Figure 5.4 presents visually proposed schema for interpolating odd sample with nearby even samples for the lifting scheme using squad interpolation in tangent space. Applying Equation 5.28 for prediction step we obtain the following formula:

$$o_i^j = o_i^{j+1} - 0.5625 * e_{i+1}^{j+1} - 0.5625 e_{i+2}^{j+1} + 0.0625 * e_i^{j+1} + 0.0625 * e_{i+3}^{j+1} \tag{5.29}$$

Calculating update, it must be ensured that average value remains constant according to Equation 5.30.

$$\frac{1}{n}\sum_{i}^{n/2}\left(e_i^{j+1}+o_i^{j+1}\right)=\frac{2}{n}\sum_{i}^{n/2}e_i^j \tag{5.30}$$

Reorganizing summation in Equation 5.30 and extending odd values according to Equation 5.29 its simpler form can be obtained:

$$e_{i+1}^j = e_{i+1} + 0.5 * o_i^j$$

This is straightforward description for the update step, that guarantees constant average value and uses only one sample for averaging, although four were used for prediction.

To obtain backward lifting scheme transform we simply change signs, as proposed for general lifting scheme construction, leading to:

- undo-predict

$$o_i^{j+1} = o_i^j + 0.5625 * e_{i+1}^j + 0.5625 e_{i+2}^j - 0.0625 * e_i^j - 0.0625 * e_{i+3}^j$$

- undo-update

$$e_i^{j+1} = e_i^j - 0.5 * o_i^j$$

So proposed lifting scheme allows to do forward and backward lifting scheme transform for quaternions according to *squad* interpolation in the quaternion tangent space with an application of Bezier curves interpolation. One drawback of this method is that proposed average value assumes that signal is periodic, as values required for calculating this average might be taken from outside of the signal range. When signal is really periodic this will guarantee correct average values, but in practice it does not have to be periodic. This might cause at the borders of the interpolated range to occur greater and unnatural differences between averaged values. To eliminate such behaviour we suggest to duplicate signal border values, what should lead to a more reliable average values at the borders. Similar behaviour is applied to lerp and slerp lifting schemes, but for them only single sample can be queried from outside valid range.

5.5 Applications

We propose two applications for the developed motion analysis tool:

- noise reduction
- compression.

In all cases extracted coefficients are used to modify motion data. In noise reduction a simple quaternion threshold method is proposed. For compression a removal of high-resolution details is used.

5.5.1 Noise reduction

To eliminate noise in motion data represented by quaternions we propose a simple threshold method based on the signal coefficients obtained with multi-resolution tools. To eliminate high-frequency noise, which would be mostly visible in high-resolution signal details, we remove details with significantly small rotation angles by substituting them with the zero rotation quaternion $q = [(0, 0, 0), 1]$. Lower resolutions should not be modified in general, as small details at this levels are propagated to higher resolutions, with great probability affecting resulting signal structure significantly. We propose to do the soft thresholding for the decomposed signal details (in quaternion form) according to a given rotation cut-off angle α, where α is given in radians in range $0 < \alpha < 0.5$ according to formula:

$$c = \begin{cases} c & \text{if } \alpha < v < 1 - \alpha \\ [(0, 0, 0), 1] & \text{otherwise} \end{cases}$$

where $v = \frac{arccos(c.w)}{\pi}$. Such approach eliminates small rotations (close to 0° rotation angle, not changing body orientation significantly) and great rotations (close to 360° rotation, which brings the body very close to its initial orientation). As rotation quaternions are always represented as cosine of half desired rotation angle, therefore α values were limited to the given range.

After modifying particular coefficients levels, signal should be reconstructed with proper backward lifting scheme transform. Resulting signal is expected to be less noisy than before proposed modifications.

5.5.2 Data compression

We propose a simple lossy data compression technique. Compressed signal is stored in form of decomposed lifting scheme details and global signal average value, allowing to reconstruct signal structure, where some of the detail

resolutions are removed completely. Those resolutions would be filled with zero rotation quaternion during signal reconstruction. Question arise, which coefficients levels should be deleted. We propose to remove several highest resolutions, picked up with a rule of thumb, but we believe, that there exist particular moves characteristics, allowing to estimate precisely sets of best resolutions to be removed (be the least informative for this move). It is highly probable that such characteristic might be given for particular joints in human musculo-skeletal model for a given motions. As highest coefficient levels contain most of the decomposed signal data, removing only the highest resolution provides compression ratio of 50%. To limit number of values required to store quaternions in comparison to Euler angles, logarithm function might be used additionally, to represent quaternions only with three, instead of four values. During decompression those values would be extracted back to quaternion space. Removed coefficient levels should not disturb significantly input signal structure.

5.6 Tests

To verify that proposed tools work correctly, we planed several tests. First of all, we want to see if proposed lifting schemes transformations allow to reconstruct the decomposed signal to its original form, when no modifications were made to the coefficients. Secondly, we want to verify that developed lifting schemes can be used successfully for proposed applications. In this step only those lifting schemes are tested, that allowed correct signal reconstruction based on the results of the first test. This chapter presents detailed description of used motion data, experiments configurations, obtained results and their comments.

5.6.1 Test data

For all proposed tests motion data recorded at HML is used. Left knee joint has been extracted from delivered skeleton motion (healthy, 26 years old male). Recorded data set samples count was equal to 812, but for experiments first 512 samples are used, as this is the closes power of two. Data were recorder with 100Hz resolution what gives recording duration approximately equal to 5 seconds - long enough to capture several steps. Figure 5.5 presents knee rotations in time as Euler angles for clarity. Angle values are truncated to a range of $\langle -180°; 180° \rangle$. All results are also presented in this form. Additionally, showing lifting schemes results also unmodified samples, indexed with values equal or greater to 512 are presented for comparison. Presenting

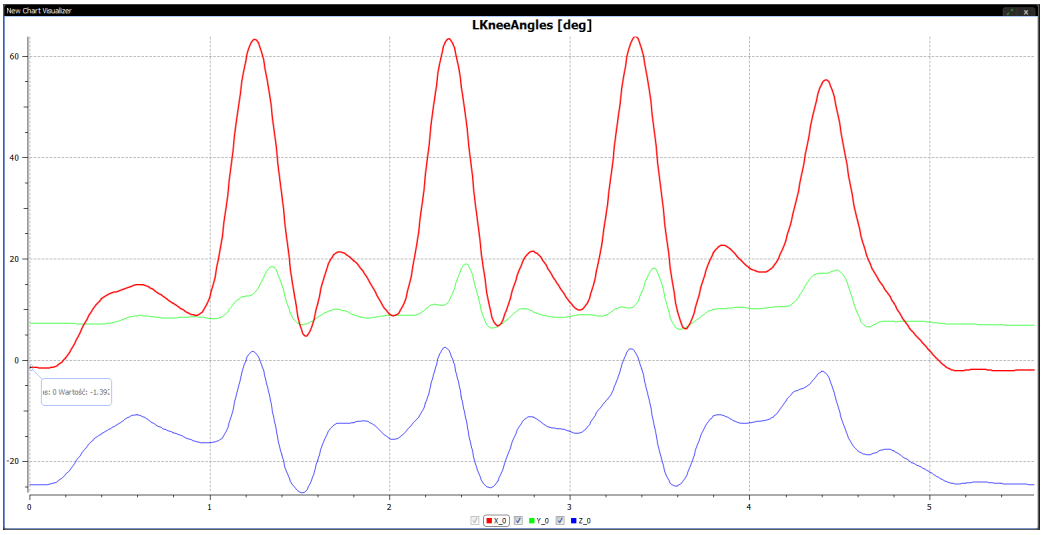


Figure 5.5: Test data - left knee of 26 years old and healthy male

lifting schemes decomposition details of all resolutions are shown on a single chart, where lower resolutions can be observed by recursive subdividing chart plot on half, moving from right to left. On the left side the lowest resolutions details are presented up to the highest resolution coefficients on the right half of the plot. Moreover, we give the following mapping between colour and rotation axis:

- red → x-axis,
- green → y-axis,
- blue → z-axis.

Time axis presents values always in seconds.

5.6.2 Comparing results

To compare obtained results we introduce a simple distance measure for quaternion signals. It is similar to Riemannian approach for rotation distance as a length of the shortest great-arc between rotations. Lets define:

$$d_q\left(q_a, q_b\right) = arccos\left(Re\left(q_a q_b^{-1}\right)\right)$$

It can be noticed, that when both rotations are close, this distance would be small, in particular, it is equal to zero, when two identical rotations are

compared. With so defined distance measure difference between two quaternion signals Q_A, Q_B, with number of elements equal to N is defined as:

$$D_q = \sum_{i=1}^{N} d_q \left(Q_{Ai}, Q_{Bi} \right)$$

The greater the difference between particular signals elements the greater the distance value is.

5.6.3 Signal reconstruction

5.6.3.1 LinHaar

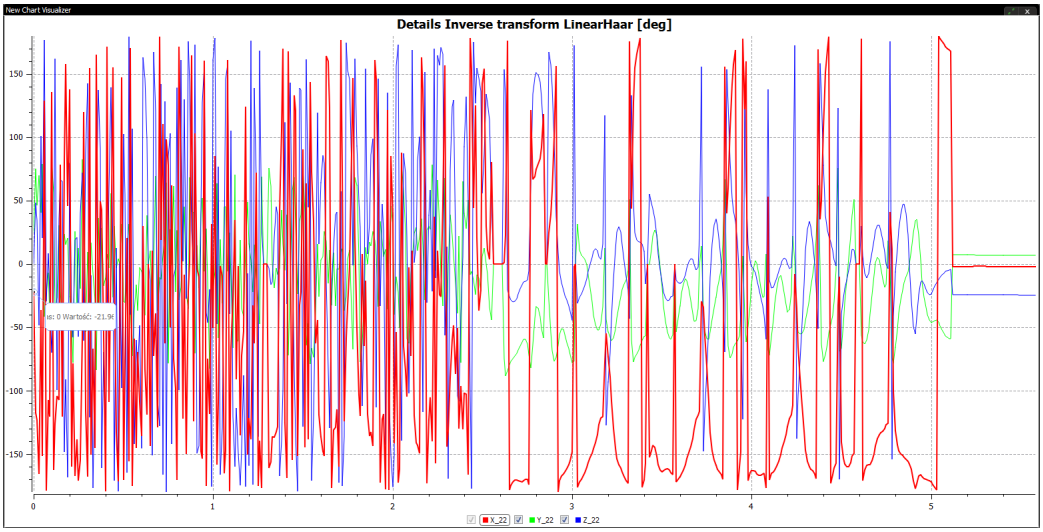


Figure 5.6: LinHaar details after forward transform

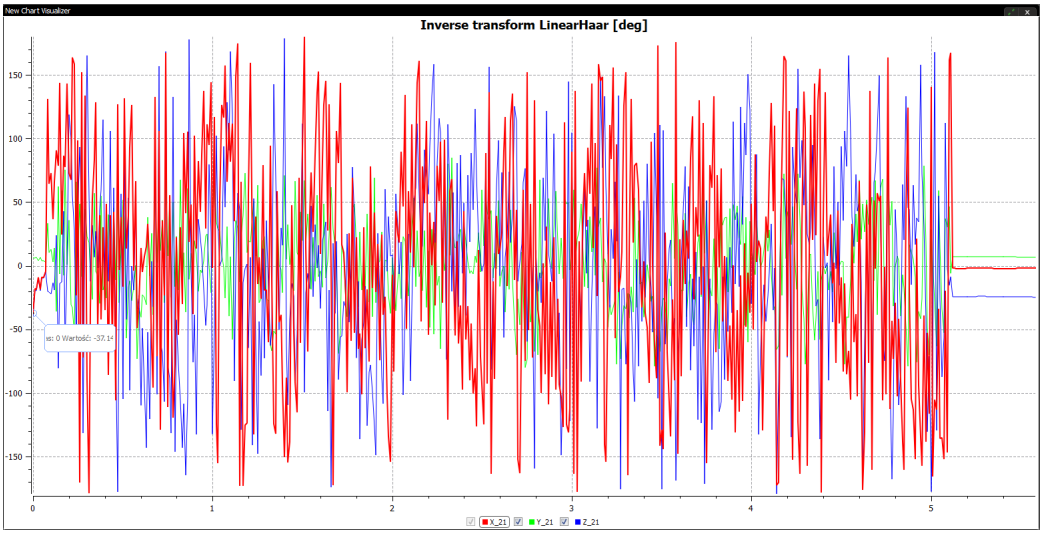


Figure 5.7: LinHaar signal reconstruction

5.6.3.2 QuatHaar

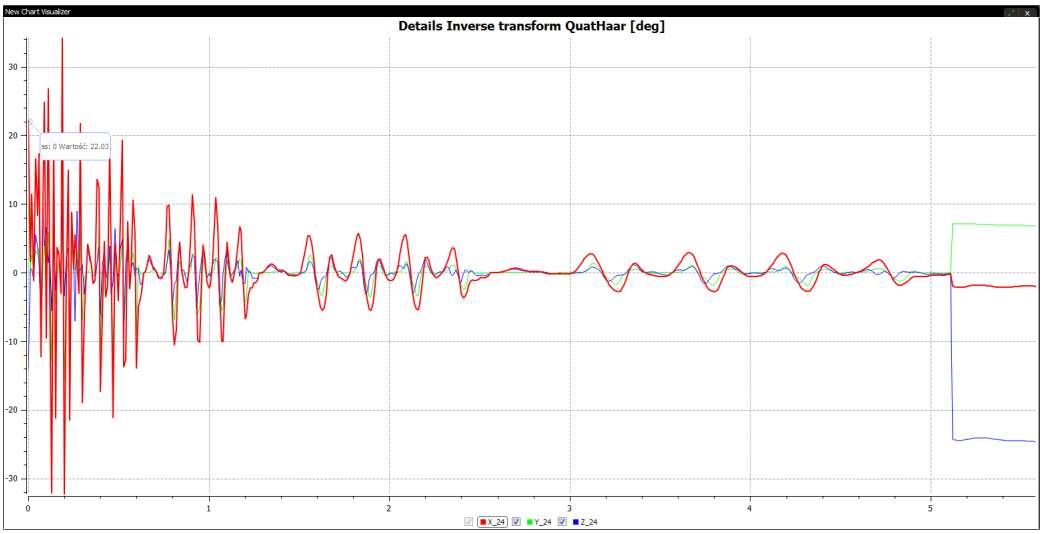


Figure 5.8: QuatHaar details after forward transform

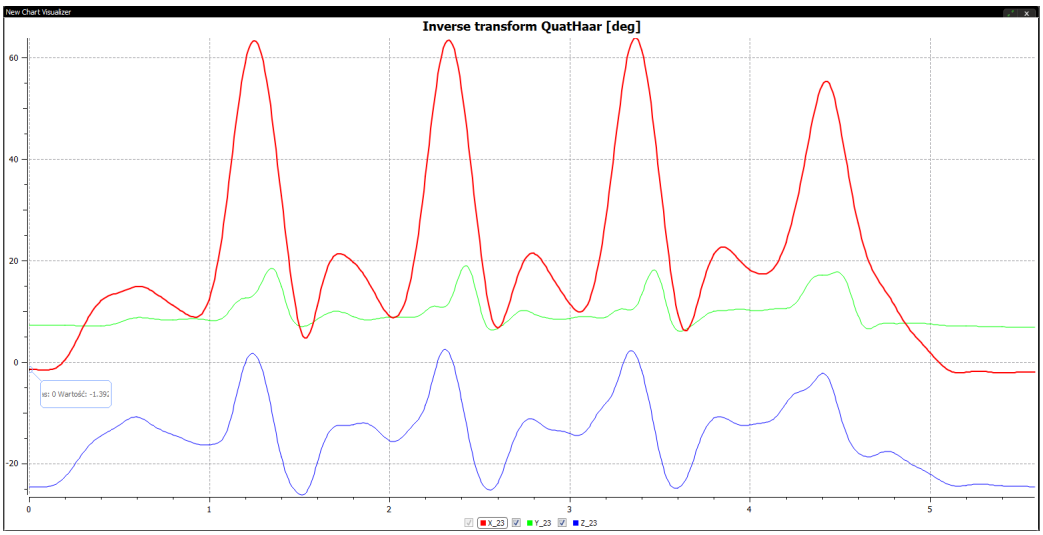


Figure 5.9: QuatHaar signal reconstruction

5.6.3.3 lerp

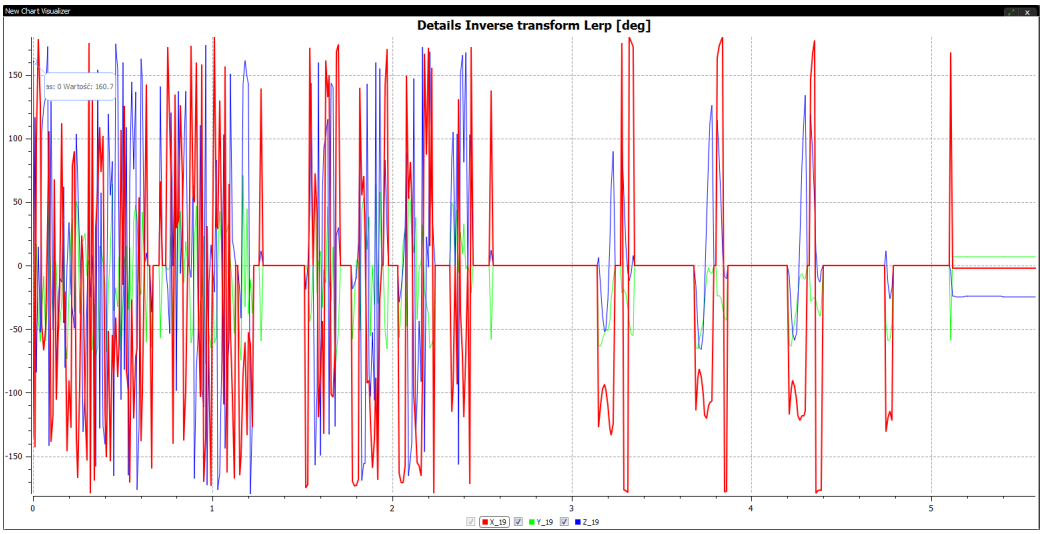


Figure 5.10: Lerp details after forward transform

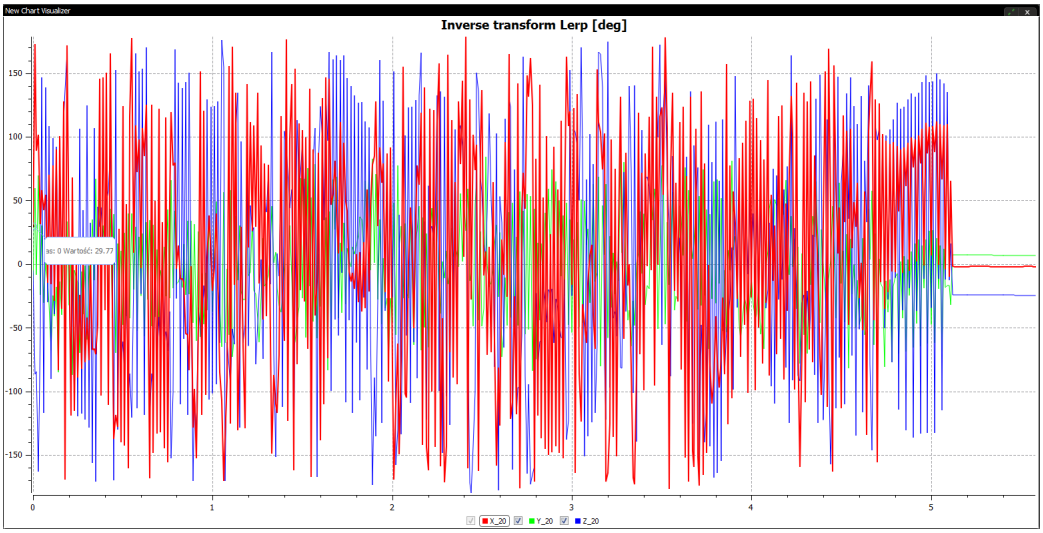


Figure 5.11: Lerp signal reconstruction

5.6.3.4 slerp

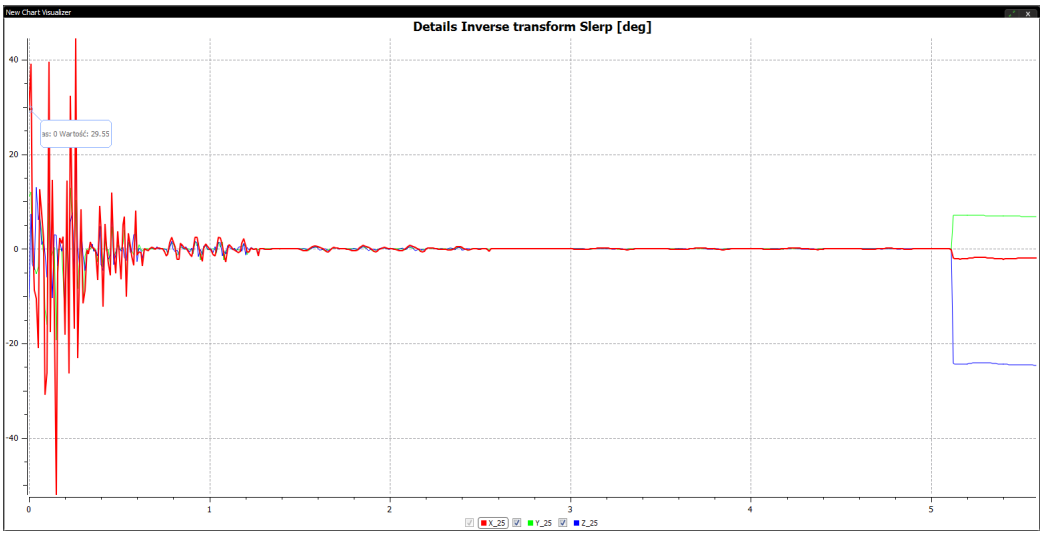


Figure 5.12: Slerp details after forward transform

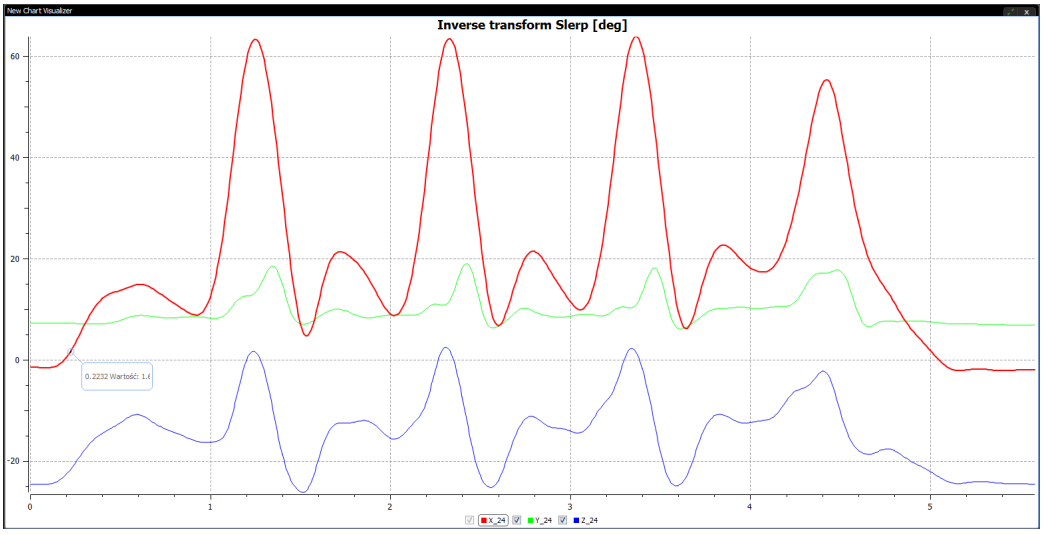


Figure 5.13: Slerp signal reconstruction

5.6.3.5 squad

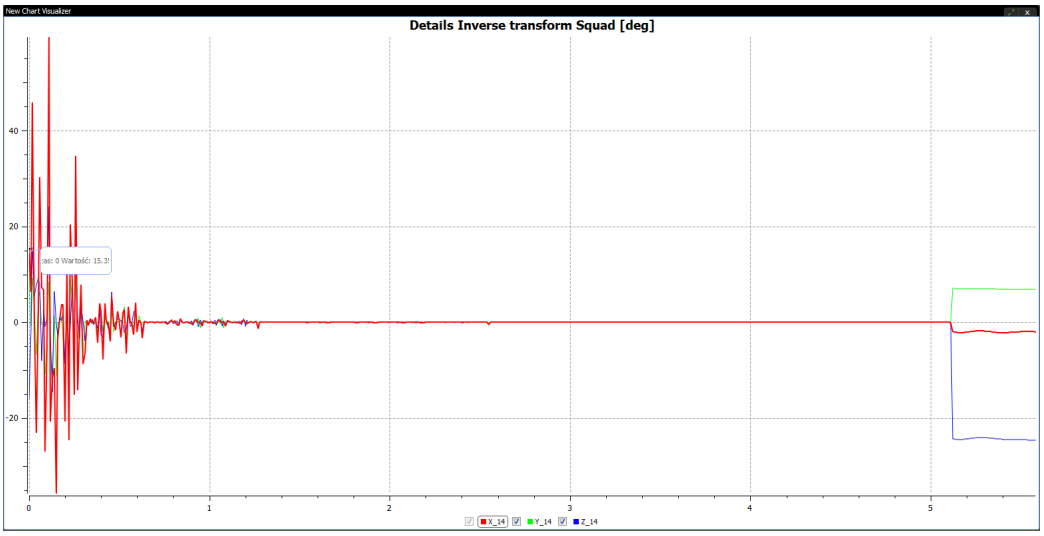


Figure 5.14: Squad details after forward transform

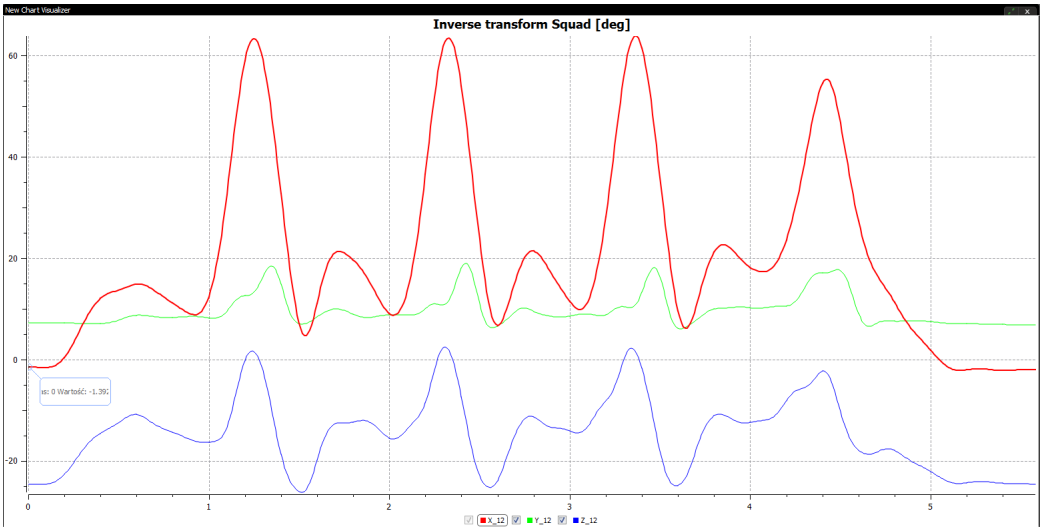


Figure 5.15: Squad signal reconstruction

5.6.3.6 tangentSpace

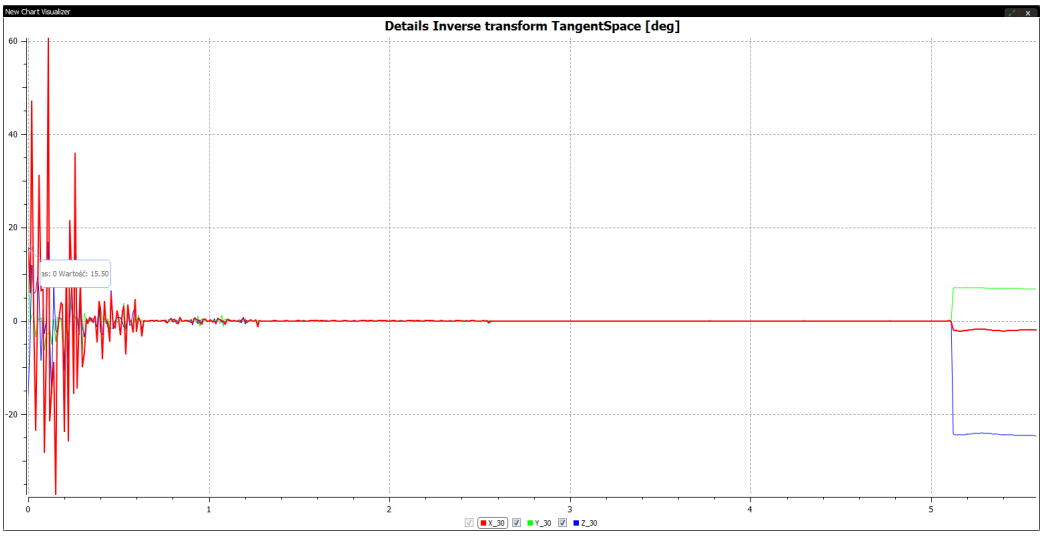


Figure 5.16: TangentSpace details after forward transform

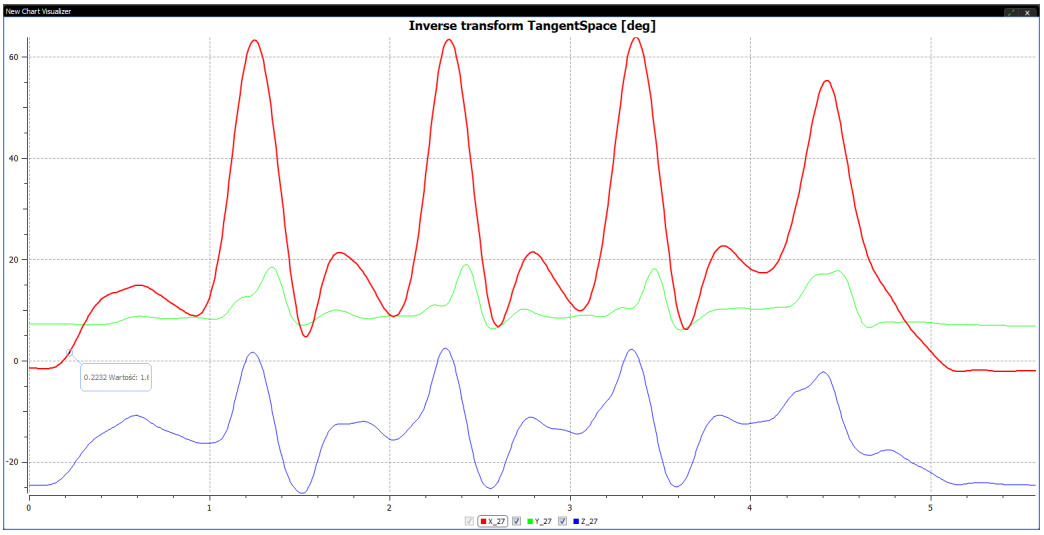


Figure 5.17: TangentSpace signal reconstruction

5.6.3.7 Summary

Presented signal reconstruction results show that linear approach to quaternions does not allow to analyse signals with multi-resolution tools, as they do not allow to reconstruct decomposed signals to their original form. Although

Table 5.1: Noisy signals distances to original signal

Noise ($\sigma[^\circ]$)	**Distance (radians)**
0.5	2.330736673
2	9.323036076
5	23.30663269

so proposed approach tries to capture signal details in $\mathbb{R}4$ space, the extracted rotation details converge to value $[[0,0,0),0]$ for identical rotations, which does not represent valid rotation (unit quaternion), instead of $[[0,0,0),1]$, representing neutral quaternion, not modifying vector in any way. Normalization for such results additionally introduces errors, as rotation angles are interpolated with linear function, not handling at all rotation periodic behaviour.

Test have proven correct squad lifting scheme construction, as its decomposed details are almost identical with dedicated tangent space transformation. Additionally, squad approach allowed to reconstruct previously decomposed signal to its original form.

Signal reconstruction test indicates, that only quaternion algebra should be applied for multi-resolution motion analysis.

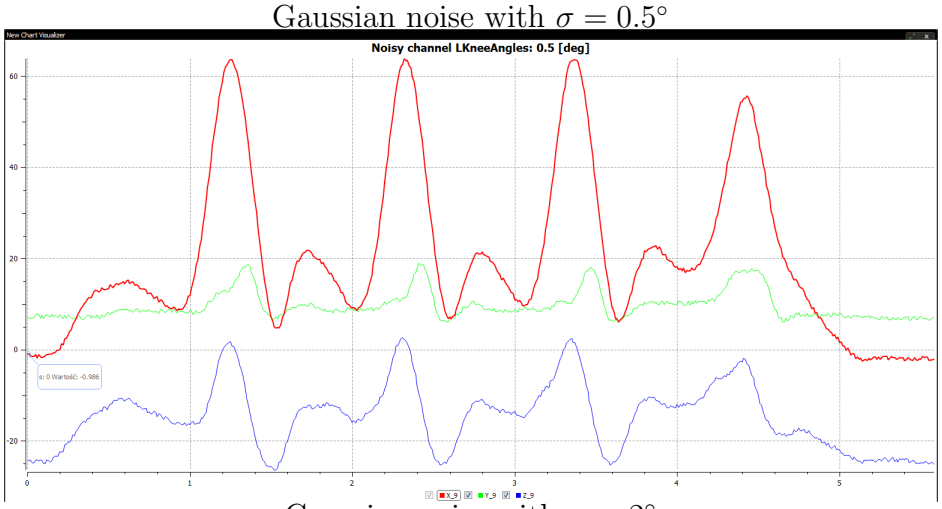
5.6.4 Noise reduction

To test application of lifting schemes for noise reduction we had to add artificial noise to the test data. This is due to a data post-processing done in the laboratory, where data are collected. In this stage most of the noise for recorded data is removed and signal is smoothed. We decided to add three levels of white Gaussian noise for rotations in Euler angles representation, independently for each angle, with σ equal to:

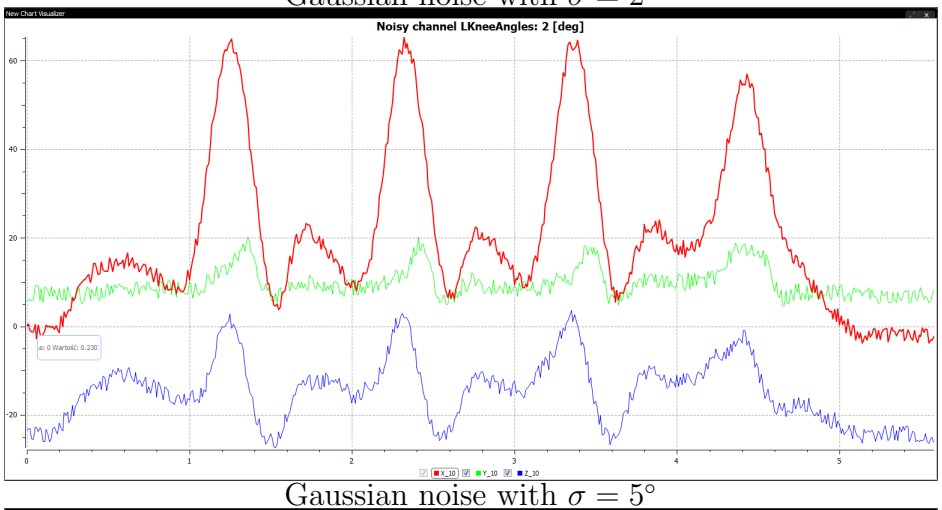
- 0.5°,
- 2°,
- 5°.

Figure 5.18 presents data after noise addition. Distances of generated noisy data to its original form are collected in Table 5.1 according to introduced quaternion signals distance measure.

Gaussian noise with $\sigma = 0.5°$



Gaussian noise with $\sigma = 2°$



Gaussian noise with $\sigma = 5°$

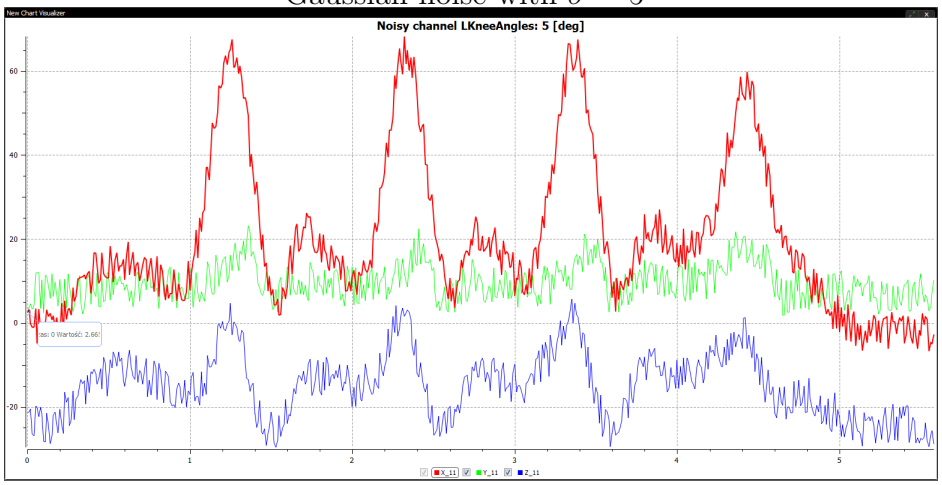


Figure 5.18: Noisy test data

Each lifting scheme was used for noise reduction according to three sets of filtered details resolutions:

- 8,
- 7 and 8,
- 6, 7 and 8.

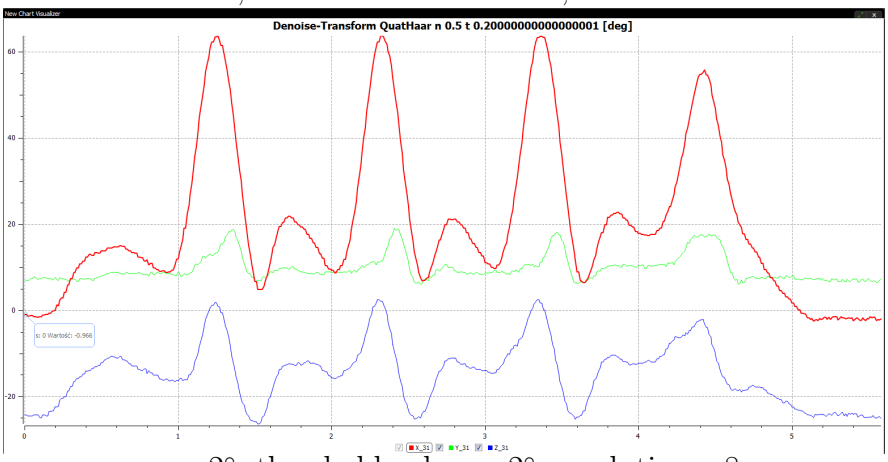
with three values of rotation threshold value for each set:

- 0.5°,
- 2°,
- 5°.

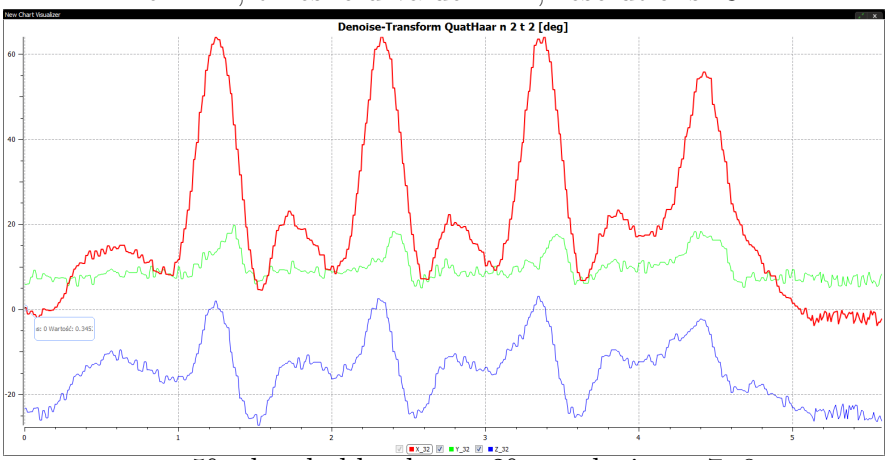
Only the best results for particular signal noises for all lifting schemes are presented. In Section 5.6.4.5 Table 5.2 presents the best results for this experiment with distances of filtered signals to the original data to verify their effectiveness.

5.6.4.1 QuatHaar

$\sigma = 0.5°$, threshold value $= 0.2°$, resolutions: 8



$\sigma = 2°$, threshold value $= 2°$, resolutions: 8



$\sigma = 5°$, threshold value $= 2°$, resolutions: 7, 8

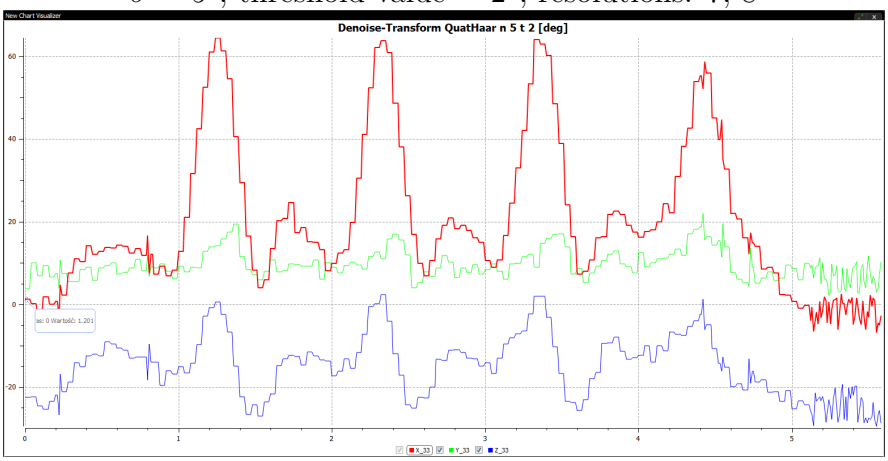
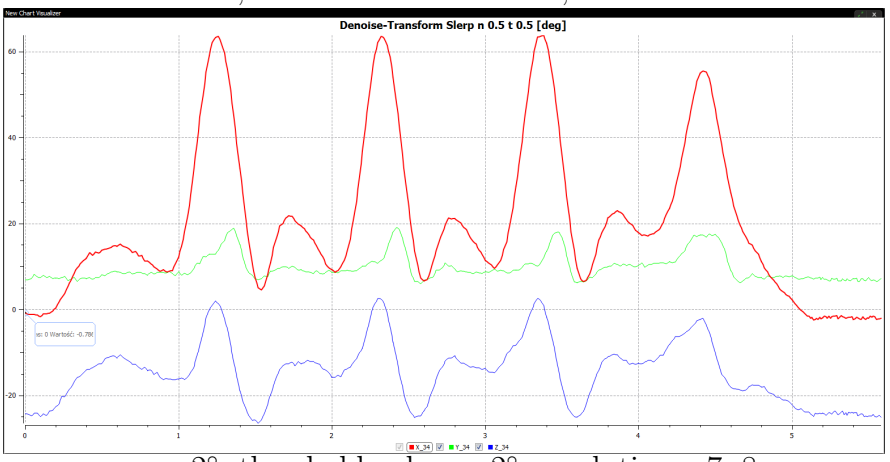


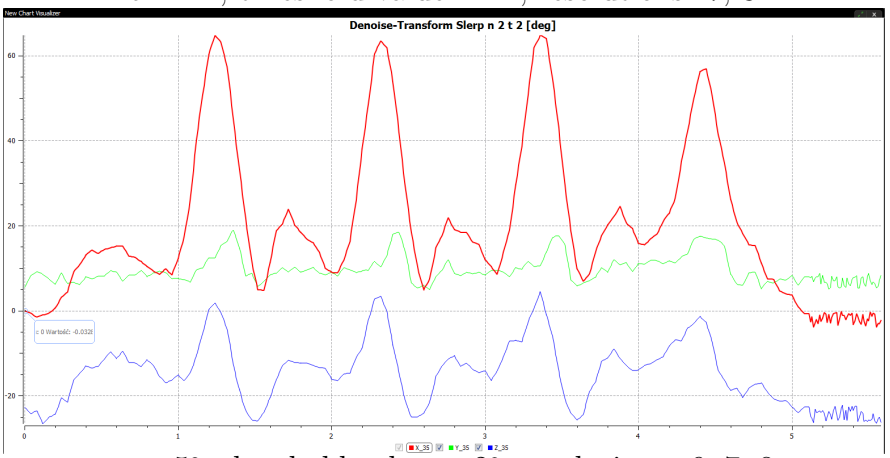
Figure 5.19: QuatHaar lifting scheme noise reduction best results

5.6.4.2 slerp

$\sigma = 0.5°$, threshold value $= 0.5°$, resolutions: 8



$\sigma = 2°$, threshold value $= 2°$, resolutions: 7, 8



$\sigma = 5°$, threshold value $= 2°$, resolutions: 6, 7, 8

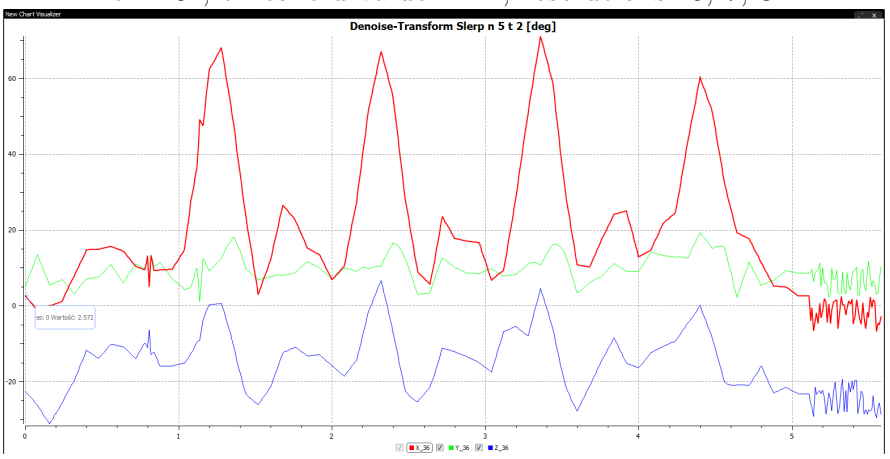
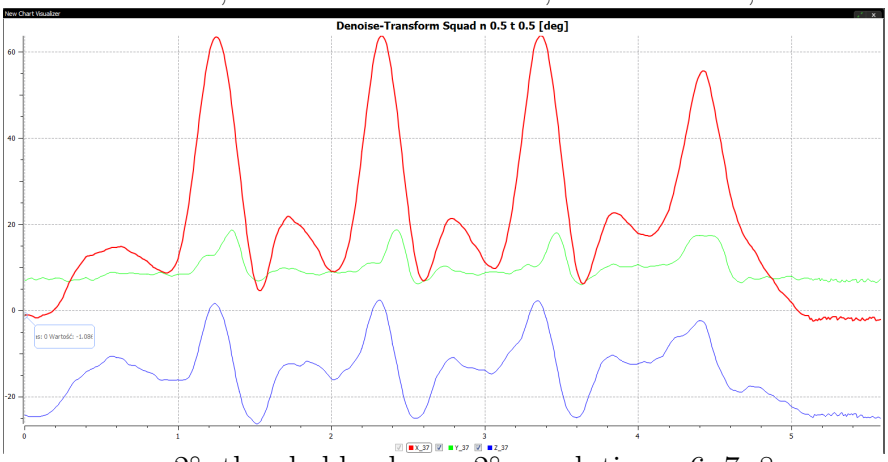


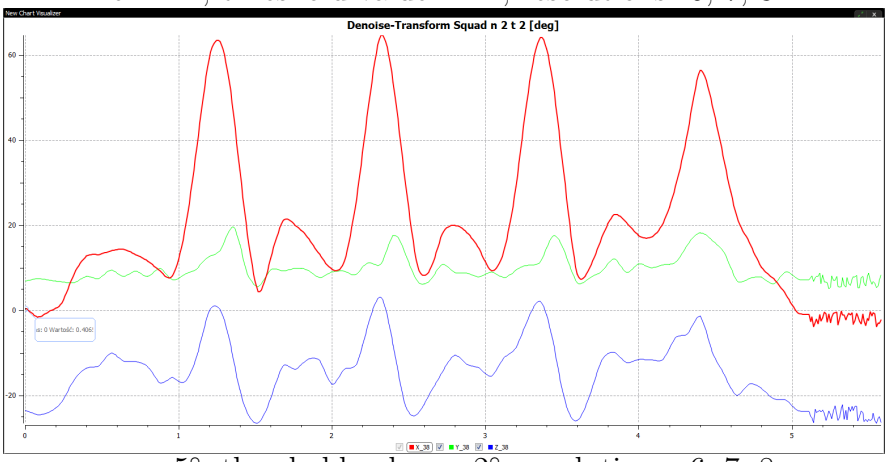
Figure 5.20: Slerp lifting scheme noise reduction best results

5.6.4.3 squad

$\sigma = 0.5°$, threshold value $= 0.5°$, resolutions: 7, 8



$\sigma = 2°$, threshold value $= 2°$, resolutions: 6, 7, 8



$\sigma = 5°$, threshold value $= 2°$, resolutions: 6, 7, 8

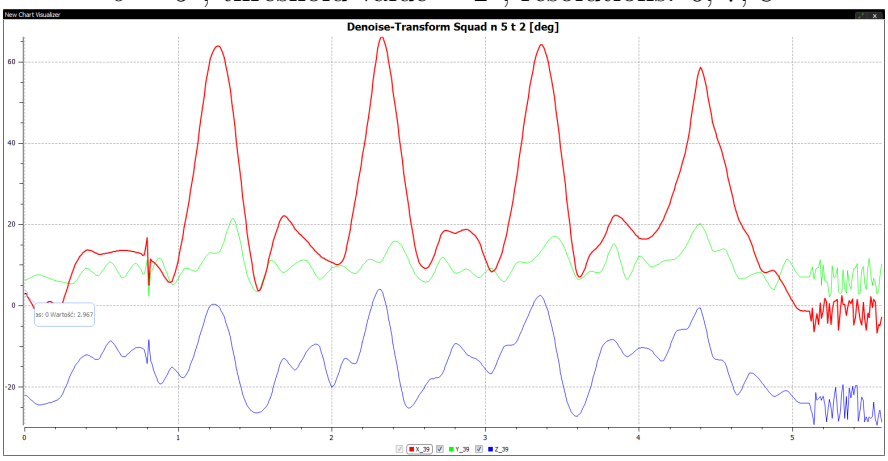
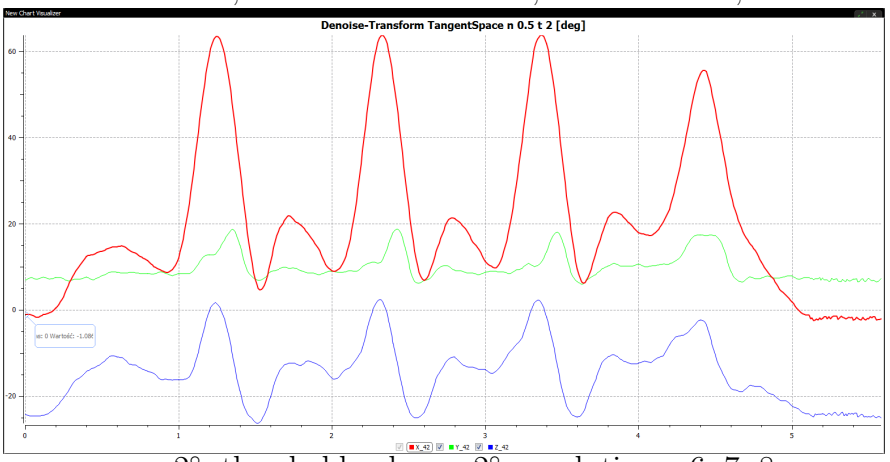


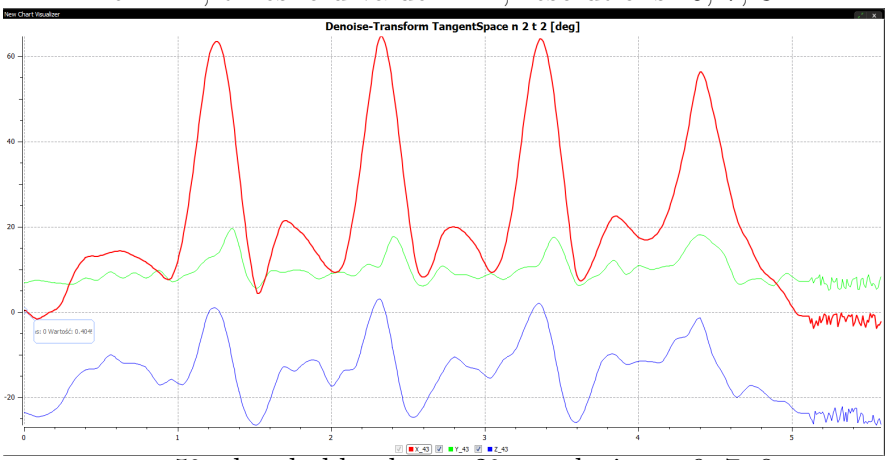
Figure 5.21: Squad lifting scheme noise reduction best results

5.6.4.4 tangentSplines

$\sigma = 0.5°$, threshold value $= 2°$, resolutions: 7, 8



$\sigma = 2°$, threshold value $= 2°$, resolutions: 6, 7, 8



$\sigma = 5°$, threshold value $= 2°$, resolutions: 6, 7, 8

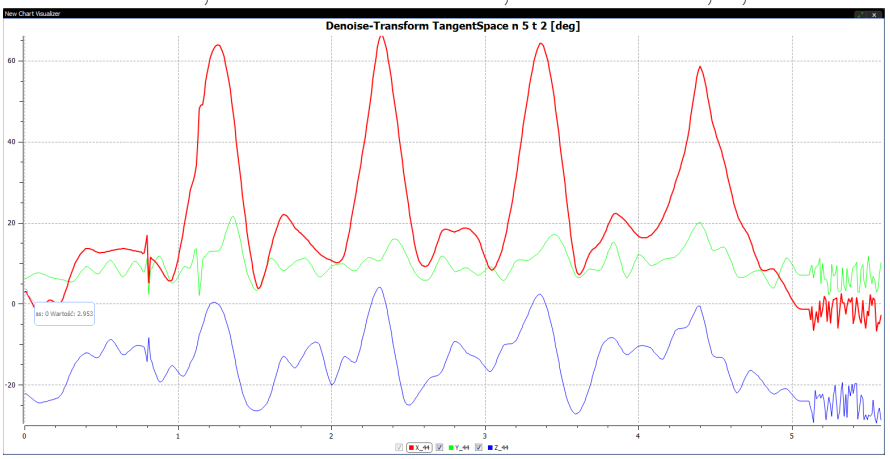


Figure 5.22: TangentSpace lifting scheme noise reduction best results

Table 5.2: Noise reduction results

Lifting scheme	**Noise** ($\sigma[^\circ]$)	**Threshold angle** ($^\circ$)	**De-noise details resolutions**	**Distance (radians)**
QuatHaar	0.5	0.2	8	2.2
QuatHaar	2	2	8	7.4
QuatHaar	5	2	7, 8	14.5
Slerp	0.5	0.5	8	1.8
Slerp	2	2	7, 8	6.8
Slerp	5	2	6, 7, 8	16.0
Squad	0.5	0.5	7, 8	1.6
Squad	2	2	6, 7, 8	5.6
Squad	5	2	6, 7, 8	13.3
TangentSpace	0.5	2	7, 8	1.6
TangentSpace	2	2	6, 7, 8	5.6
TangentSpace	5	2	6, 7, 8	13.5

Table 5.3: Compression ratios

Removed resolutions	**Compression ratio (%)**
8	50
7, 8	75
6, 7, 8	87.5

5.6.4.5 Summary

Comparing presented results in Table 5.2 with noisy data distance to original signal in Table 5.1 it can be seen, that all lifting schemes have improved signal quality by reducing the noise level. The more complex interpolation method (using more data samples) in prediction step, the better noise reduction results for all noise levels are achieved. Once again results for squad and tangent space lifting schemes are almost identical, proving proper structure for squad lifting scheme.

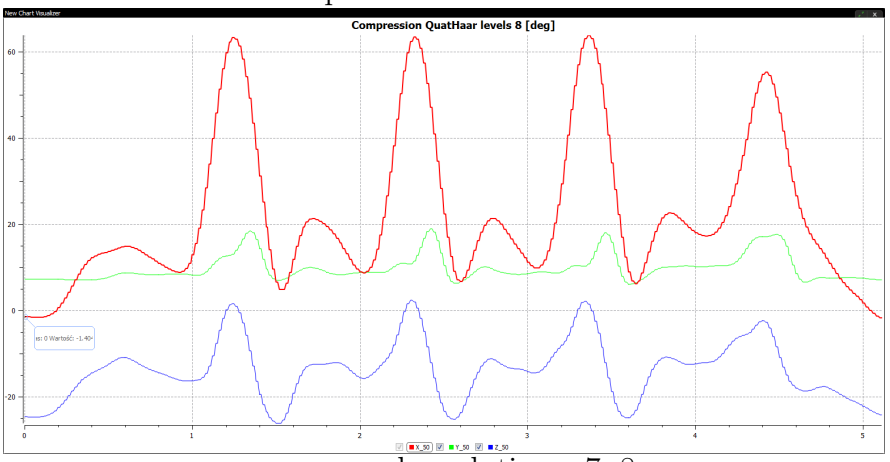
5.6.5 Compression

Testing compression abilities according to proposed compression method, three levels of compression were proposed for each lifting scheme describing removed details resolutions from decomposed signal.

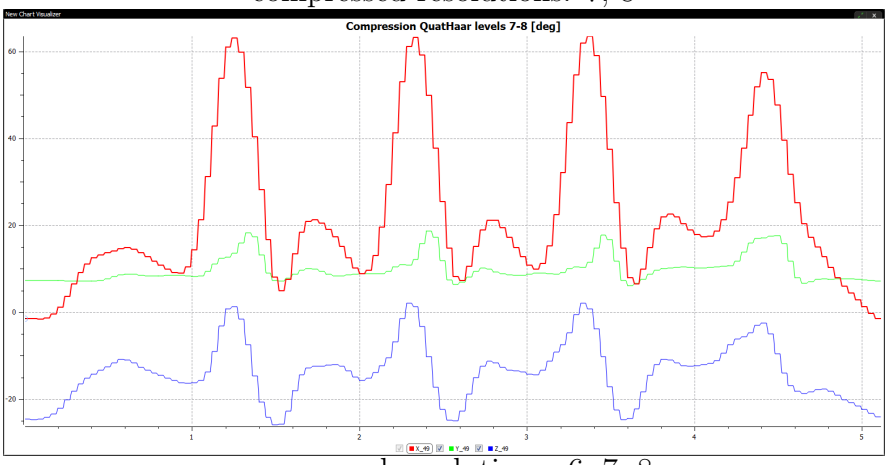
Table 5.3 presents removed details resolutions configurations and corresponding compression ratios. To compare compression quality for proposed lifting schemes, the distance of decompressed and reconstructed signal to original data is measured.

5.6.5.1 QuatHaar

compressed resolutions: 8



compressed resolutions: 7, 8



compressed resolutions: 6, 7, 8

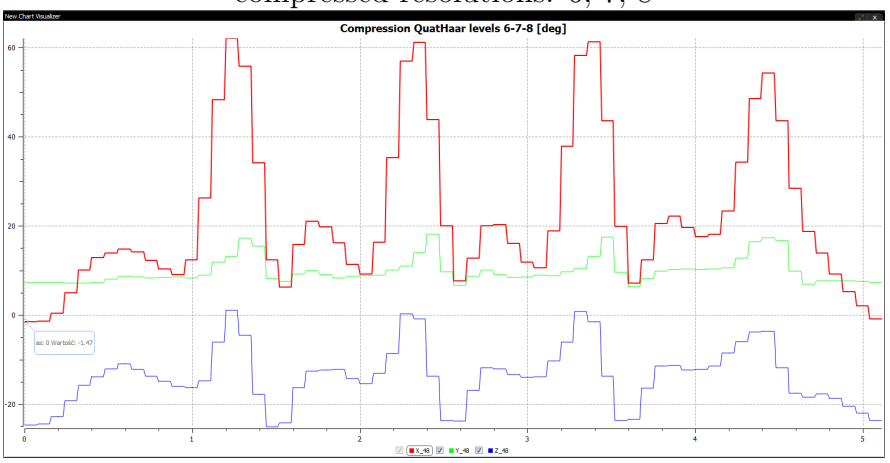
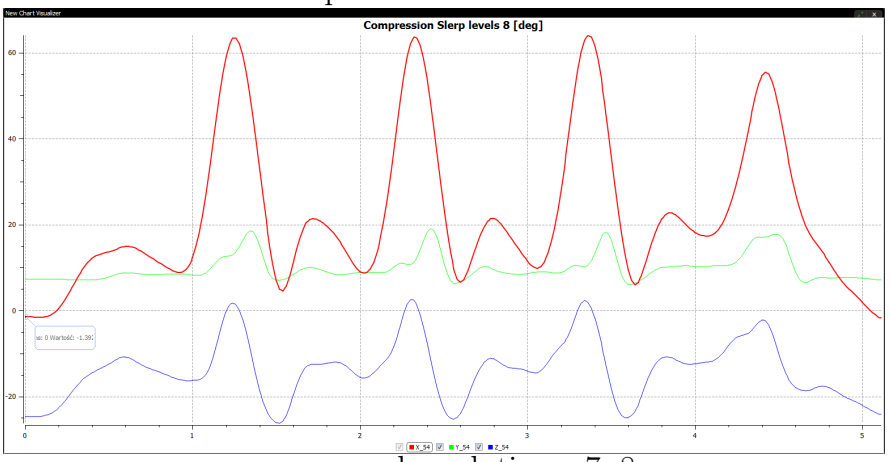


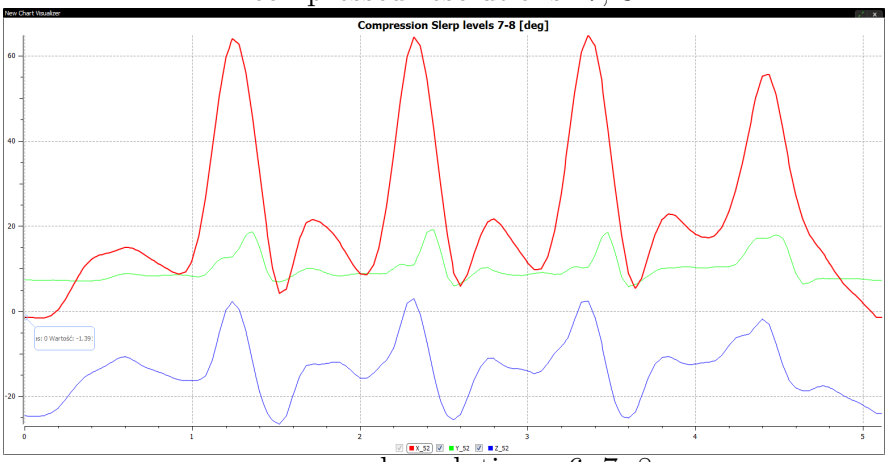
Figure 5.23: QuatHaar compression quality loss

5.6.5.2 slerp

compressed resolutions: 8



compressed resolutions: 7, 8



compressed resolutions: 6, 7, 8

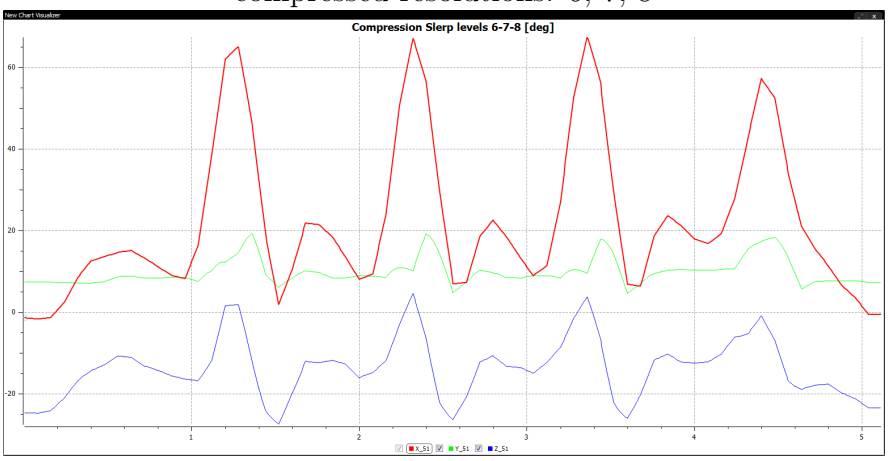
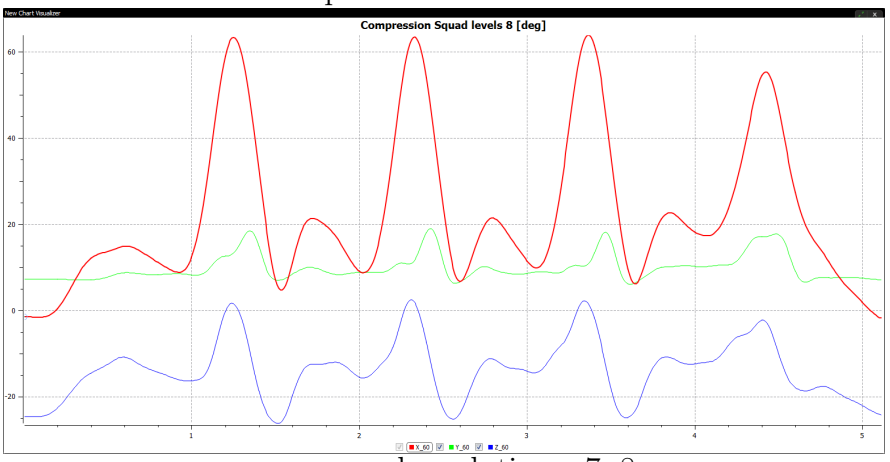


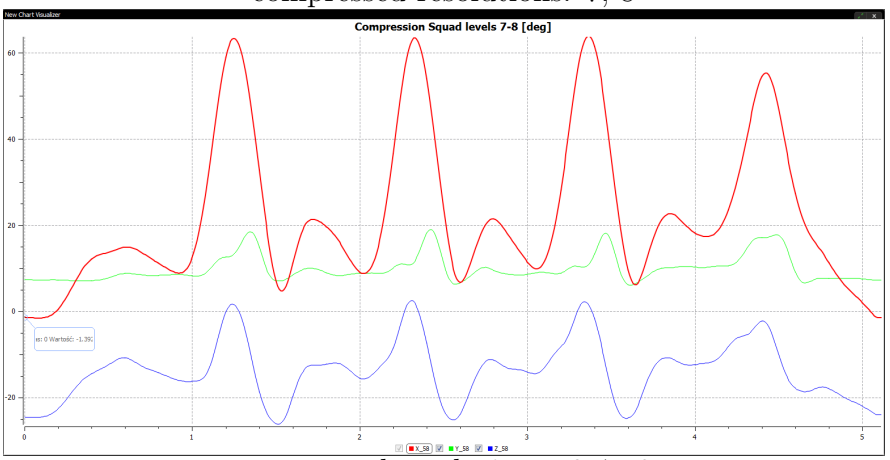
Figure 5.24: Slerp compression quality loss

5.6.5.3 squad

compressed resolutions: 8



compressed resolutions: 7, 8



compressed resolutions: 6, 7, 8

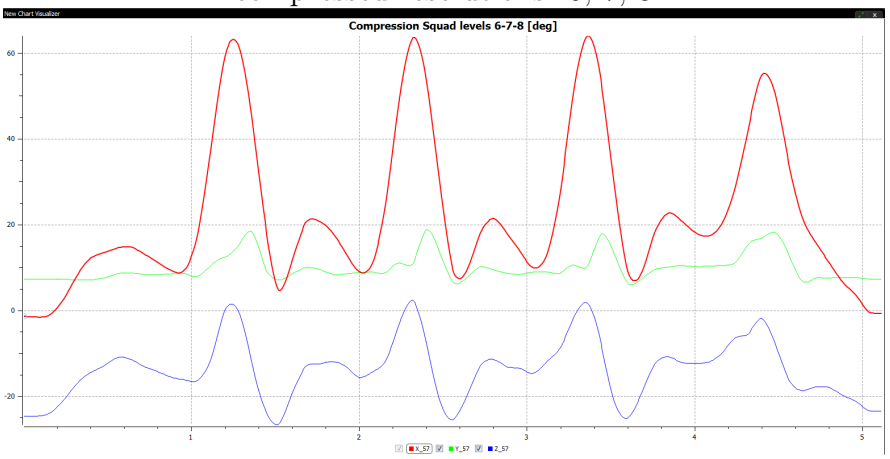
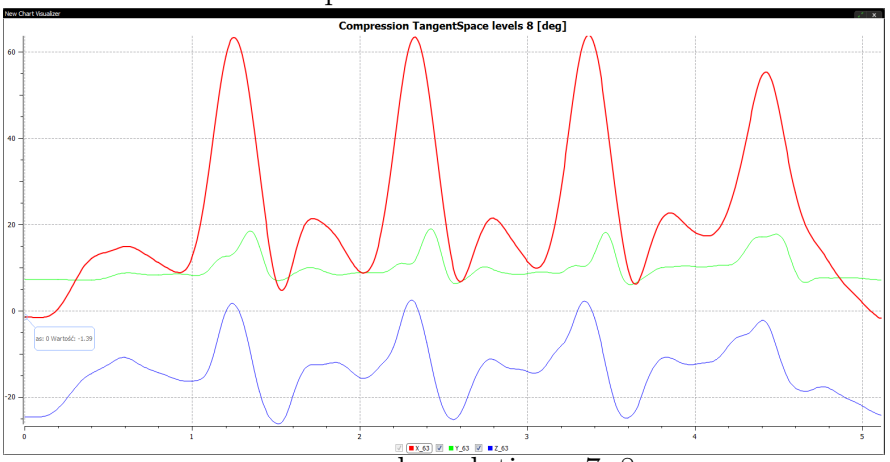


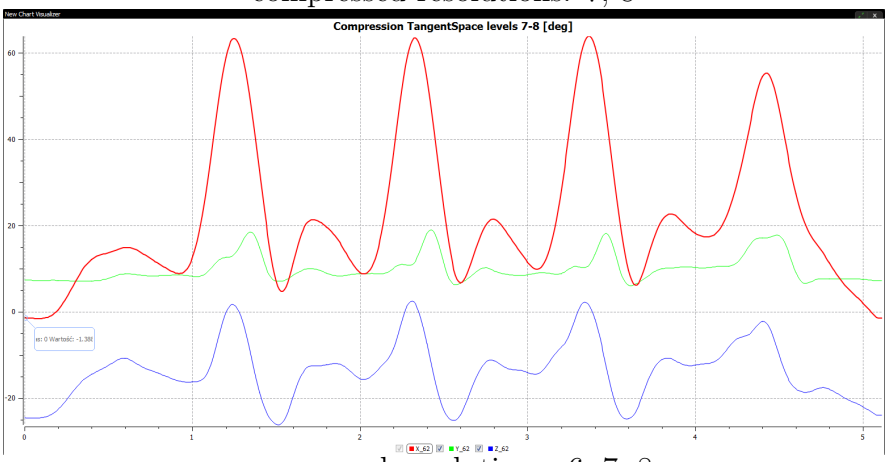
Figure 5.25: Squad compression quality loss

5.6.5.4 tangentSpace

compressed resolutions: 8



compressed resolutions: 7, 8



compressed resolutions: 6, 7, 8

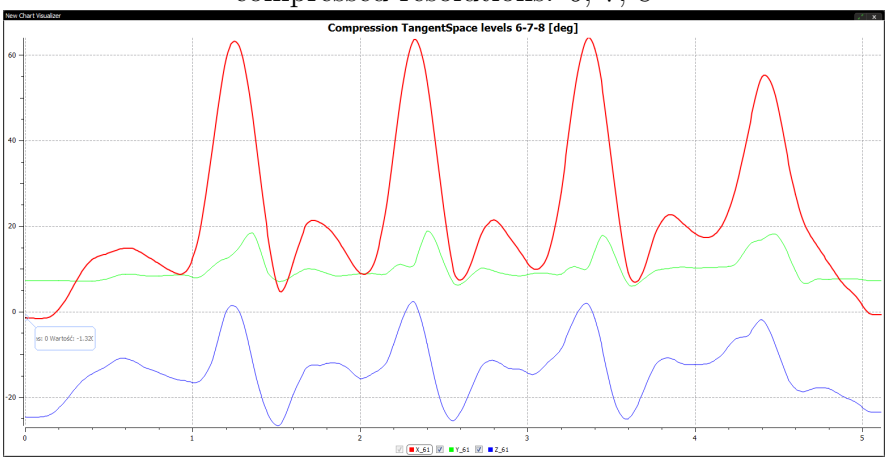


Figure 5.26: TangentSpace compression quality loss

Table 5.4: Compression results

Lifting scheme	Removed resolutions	Distance (radians)
QuatHaar	8	2.5
QuatHaar	7, 8	5.0
QuatHaar	6, 7, 8	10.0
Slerp	8	0.2
Slerp	7, 8	0.6
Slerp	6, 7, 8	2.8
Squad	8	0.006
Squad	7, 8	0.08
Squad	6, 7, 8	1.3
TangentSpace	8	0.009
TangentSpace	7, 8	0.09
TangentSpace	6, 7, 8	1.3

5.6.5.5 Summary

Table 5.4 presents comparison of various lifting schemes used for compression with different details resolutions levels removed. It can be seen that applying slerp, squad or tangent pace lifting schemes allows to limit data up to only 12.5% of its original size with almost unnoticeable lose of quality. This allows to limit significantly recorded motion data in form of rotation signals, still providing very detailed signal description. It is noticeable especially for squad and tangent space lifting schemes, based on their decomposed signal details, where first few details levels do not carry any relevant signal information (all Euler angles very close to zero).

5.7 Implementation overview for MDE

I have implemented all presented lifting schemes with their applications for noise reduction and compression as elements of MDE data flow processing framework. All tests were done with help of visual data flow environment. Presented graphical results were produced by MDE built-in chart visualizer. Only data flow specific implementations are presented, skipping general purpose code for lifting scheme and quaternion operations, available in separate, general purpose libraries: *QuatUtils* and *GeneralAlgorithms*. Both libraries, with source code, are available on attached CD.

Data format converters As built-in data source for data flow delivers joint rotation data in Euler angles format, denoted in degrees (°), four processors were implemented to convert data to quaternion format and backward to Euler angles:

- EulerDegreesToRadiansConverter,
- EulerRadiansToDegreesConverter,
- QuaternionToEulerConverter,
- EulerToQuaternionConverter.

First two converters changes angles values representation from degrees (°) to radians and backward. This is required for further rotation representation change to quaternions, for which next two converters are dedicated. They change rotation representation from Euler angles denoted in radians to quaternions and backward, so that data can be processed with proposed lifting schemes and visualized with chart visualizer.

Noise generator To generate noisy data based on Euler angle representation, a dedicated processor *EulerNoiseAdderProcessor* was developed. It allows to set up custom noise level with a simple configuration widget. Configured noise is added to each sample in the time series. Noise is added independently for each Euler angle.

Lifting schemes To implement various lifting schemes and their coefficients modifications there was proposed an universal, generic data processor - *QuaternionProcessorT*. It accepts as a template parameter types offering specific static method - *process(JointAnglesInputPin * in, JointAnglesOutputPin * out)*, taking input and output pin with data in quaternion form, that are going to be processed and propagated. Provided helper template class *TransformProcessorT* implements such behaviour for all lifting schemes, taking two template parameters:

```
template<bool forward, class LS>
class TransformProcessorT;
```

First of the parameters defines lifting scheme transformation direction, as all lifting schemes implement common interface. Second parameter defines lifting scheme that is going to be used to decompose and reconstruct the processed data.

Compression and noise reduction tools To test applications of developed motion analysis tools for noise reduction, a processors realizing signal de-noising was proposed - *QuaternionDenoiseProcessor*. It has configurable details resolutions, for which noise removal would be performed according to chosen threshold value. For compressing and decompressing motion data two classes were proposed:

- *QuaternionCompressorProcessor*,
- *QuaternionDecompressorProcessor*.

QuaternionCompressorProcessor is a configurable processor, allowing to define compression ratio as a number of highest resolutions to remove. *QuaternionDecompressorProcessor* automatically reconstructs signal to its original form (resolution) based on data format used to store compressed signals. Both *QuaternionDenoiseProcessor* and *QuaternionCompressorProcessor* work on decomposed signals details provided by various lifting schemes. Additionally, to measure the distance between signals in quaternion form a processor *QuaternionDistanceProcessor* was developed, returning scalar value representing signals distance.

Data types In header file *Types.h* various types are defined, representing input and output pins for data flow, wrapping particular data types. Also several new types are wrapped with OW to be supported by the MDE:

- kinematic::JointAngleChannel - motion data in quaternion format,
- QuatUtils::QuatLiftingCompressor::CompressedSignal - compressed motion data format exchanged by *QuaternionCompressorProcessor* and *QuaternionDecompressorProcessor*,
- core::ConstObjectsList - aggregate of data wrapped with OW mechanism to deliver an aggregate of test results.

In the end all lifting schemes type definitions are given, to simplify their usage with *TransformProcessorT* template, when creating forward and backward transforms processors.

Classes diagram Presented processors with dedicated helper classes are shown in Figure 5.27. As it can be noticed, all processors implement minimum two interfaces:

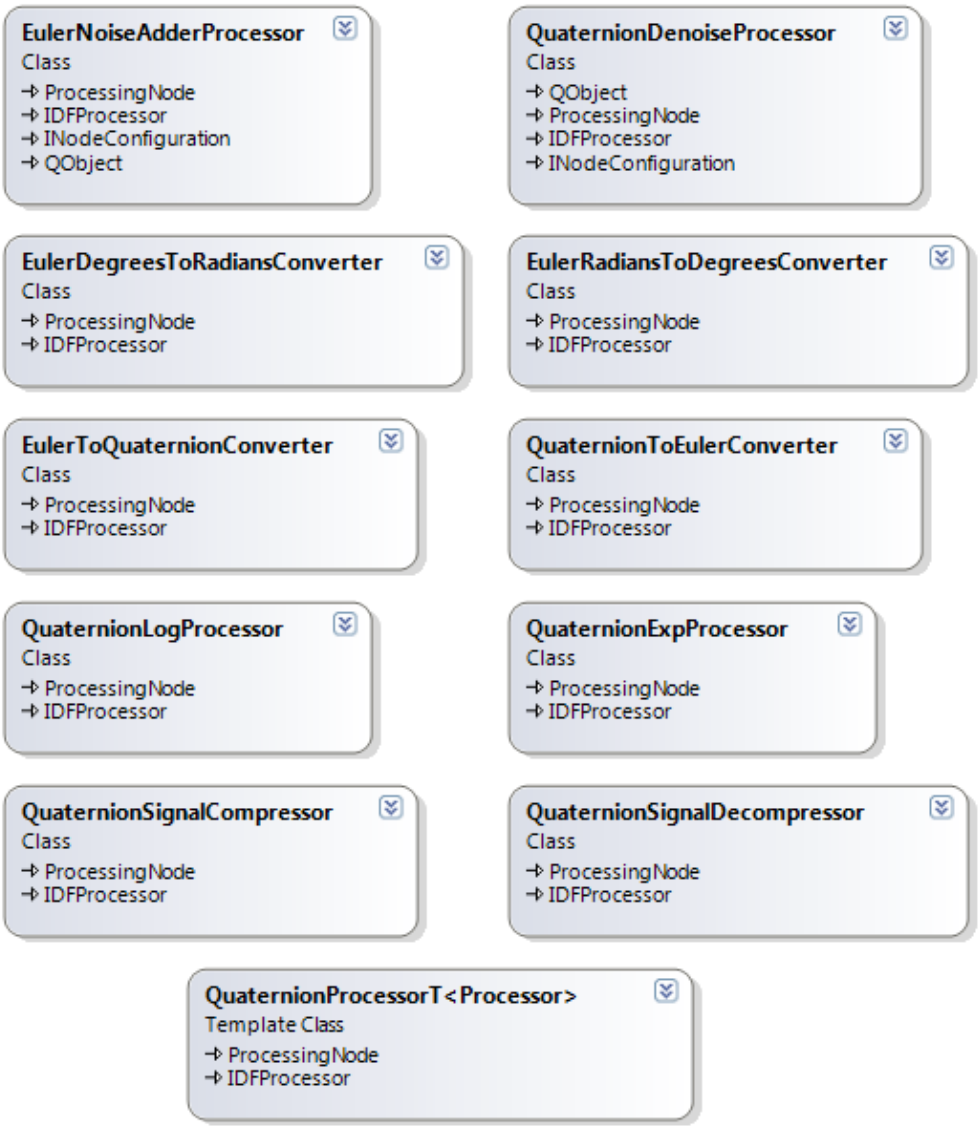


Figure 5.27: Quaternion based motion analysis data processors

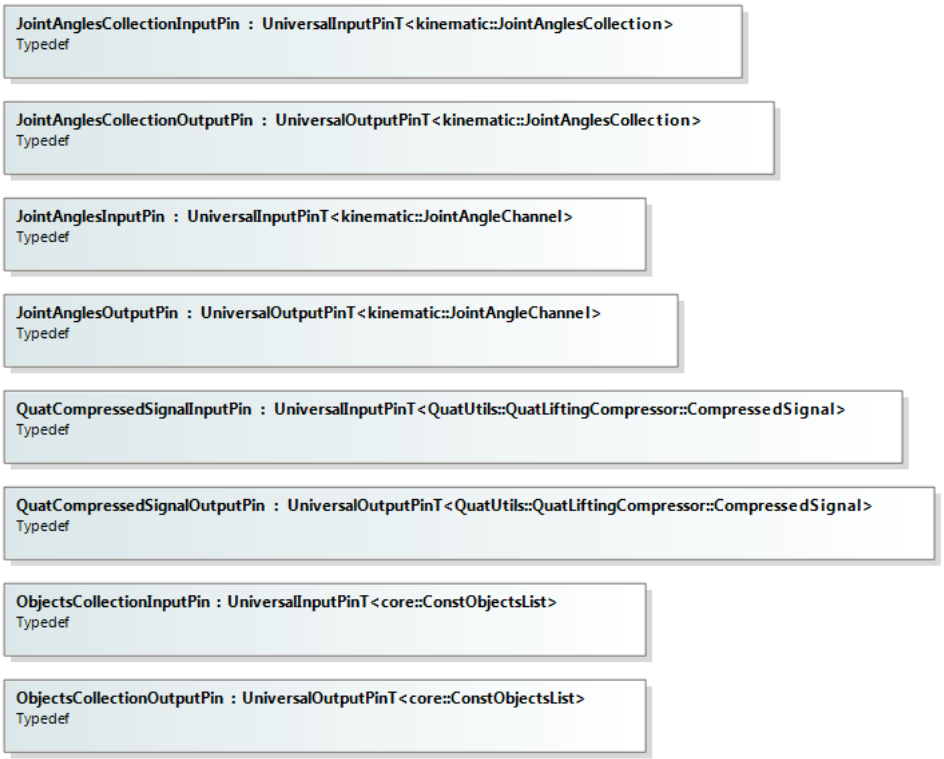


Figure 5.28: Pins data wrappers

- one for data flow connectivity logic and model definition (*df::ProcessingNode*)
- second for data flow processing logic to control processing execution (*df::IDFProcessor*).

Helper classes delivered with plug-in *dfElements* provide wrappers for pins with various data types, implementing complete data exchange based on OW mechanism. Proper pin type definitions can be seen in Figure 5.28.

Example quaternion processor Now a simple quaternion processor example is presented for logarithm function. Only processor constructor, creating node pins configuration for data flow model, and processing method for data flow processing logic are given. Listing 5.1 provides node declaration, where nodes pins configuration can be seen. Listing 5.2 presents processor constructor and Listing 5.3 shows data unpacking from input pins, creation

Listing 5.1: Procesing node declaration

```
#include "Types.h"           //defined pins
#include <dflib/Node.h>      //data flow model node (pins configuration)
#include <dflib/IDFNode.h>   //data flow processing logic

class QuaternionLogProcessor: public df::ProcessingNode, public ↩
    df::IDFProcessor
{
public:
    //! Default constructor
    QuaternionLogProcessor();
    //! Virtual destructor
    virtual ~QuaternionLogProcessor();
    //! Implementation for processing logic
    virtual void process();
    //! Implementation for processing logic
    virtual void reset();

private:
    //! Output pin with quaternion data
    JointAnglesOutputPin* outPinA;
    //! Input pin with quaternion data
    JointAnglesInputPin* inPinA;
};
```

of output data, data processing and propagation of output data with output pins. More examples with full source code for presented solutions and tests can be found on attached CD. To simplify experiments a general class performing all test was proposed - *CompleteTestProcessor*.

Listing 5.2: Procesing node constructor

```
QuaternionLogProcessor::QuaternionLogProcessor()
{
    //Pins are taking reference so that they know which node to inform,
    //when data is ready to consume (input), or data is consumed (output)
    inPinA = new JointAnglesInputPin(this);
    outPinA = new JointAnglesOutputPin(this);
    //Pins are added to node - create its configuration for model
    addInputPin(inPinA);
    addOutputPin(outPinA);
}
```

Listing 5.3: Processing node data modification

```
void QuaternionLogProcessor::process()
{
    //Extract data from input pin
    auto inQuatData = inPinA->getValue();

    //output data creation - copy of input data properties
    kinematic::JointAngleChannelPtr outQuatData(new ↩
        kinematic::JointAngleChannel(inQuatData->getSamplesPerSecond()));
    outQuatData->setName("Quaternion log channel");
    outQuatData->setTimeBaseUnit(inQuatData->getTimeBaseUnit());
    outQuatData->setValueBaseUnit(inQuatData->getValueBaseUnit());

    //data edition with external library for quaternion functionality
    for(kinematic::JointAngleChannel::size_type i = 0; i < ↩
        inQuatData->size(); ++i){
        outQuatData->addPoint( osg::QuatUtils::log(inQuatData->value(i)) );
    }

    //output data is set-up in output pin for propagation
    outPinA->setValue(outQuatData);
}
```

5.8 Summary and future work

This chapter presented new tools for motion analysis based on multi-resolution approach in form of lifting scheme and quaternion representation for rotation data. New lifting scheme based on *squad* interpolation was given, which correct construction was verified with various tests. Additionally, it has been shown that custom solutions for motion analysis can be easily implemented in MDE data processing framework, what standardizes already developed tolls and provide out-of-the-box optimal computational resources utilization, data visualization and data loading.

As presented, test have proven very good features of proposed lifting schemes in areas of motion data compression and noise reduction. Those techniques can be extended with dual quaternions [34] approach, where despite rotations, also translation can be investigated. This would provide additional motion description, characteristic for particular actor. Developed lifting schemes can be also used for other dual quaternion applications like ones presented in [12, 47, 35]. Proposed technique for interpolating quaternions in tangent space can be used to explore wider range of quaternion interpolation techniques, based on B-splines over greater amount of samples. This should improve even more compression ratios without loose of data quality and noise reduction ability of presented tools. We believe that proposed

motion data decomposition can be effectively used to improve work on natural animations presented in [32]. Our tools explore in more details motion characteristics, providing grater level of accuracy for motion blending, where particular details resolutions could be morphed independently. Additionally, presented motion data decomposition can be used to send this data effectively over the network, where firstly signal coarse description is transmitted and then details, allowing to reconstruct signal with requested quality. This might be compared to interlaced technique for images coding. More detailed description for this idea, applied for surface meshes, can be found in [65].

Chapter 6

Summary and final conclusions

Motion Data Editor (MDE) In this thesis a new software - Motion Data Editor (MDE) - dedicated to general data processing, developed at Polsko-Japońska Wyższa Szkoła Technik Komputerowych (PJWSTK), was presented. Detailed application architecture and logic description have shown, that this software is very mature and reliable to use for various research work and computations. Elaborated logic elements and dedicated plug-in system show, how users can easily extend application with custom solutions and functionalities. Moreover, it has been presented that MDE provides built-in, cross-platform functionalities, supporting development of new features. Elaborated logic for threads management and efficient computations based on jobs manager concept provide reliable and standardized solution for various data processing tasks, optimally utilizing available processing power. Users can concentrate on implementing essential algorithms, instead of developing threads management and synchronization. Described procedure for loading data with data sources, parsers and dedicated data managers shows how easily new data can be delivered to application. A novel solution for uniform data handling, independently from data types has been described for strongly typed C++ language. It was presented how such mechanism, despite wrapping any data types, can also provide comprehensive type information about encapsulated data. A possibility of data lazy initialization was described for raw data and data loaded with dedicated file and stream managers, what significantly reduces application memory consumption, as data is loaded to memory only when required. A mechanism for simple reporting was elaborated, allowing to summarize work results by storing visualizer scenes with proper comments in form of rich text documents with template and styles support. It has been shown how presented logic was implemented along various libraries and how to to use described libraries to introduce custom GUI logic based on presented data processing framework.

Despite architecture and logic, designed for efficient and easy data processing, a dedicated plug-in for flexible composing of various processing blocks in more complex processing schemes, based on data flow representation, was presented. It provides application users simple mechanism of designing custom processing pipelines, with well defined data sources, processing element and data sinks. Such models can be created within dedicated graphical environment, following the concept of visual programming. This approach introduces also higher abstraction level for data processing model design with more general operations like nodes grouping to create more complex algorithms, reusable in the future, based on simpler data operations. It has been shown, how this mechanism is implemented for MDE, fitting proposed data processing logic with threads and jobs concept.

To prove MDE quality, stability and reliability, a whole procedure of software development was described. All most important tools supporting development process, with tools managing projects configuration for various platforms were presented. Proposed CI process was described in details, showing the right direction for automating commonly performed operations, minimizing risk of potential errors because of various human mistakes. It has been shown how to develop such procedure and extend it with additional functionalities like testing, generating installers and validating code quality. Various well tested and commonly used external libraries and frameworks were presented as a background for a low level modules implementation. Different programming techniques, used for application development, have been pointed out, which are considered as good practices for modern software development, minimizing application maintenance costs and improving its flexibility for new functionalities.

Further development directions for different application elements were proposed along with CI process. They cover new functionalities for visualizers, providing access to their scenes, giving opportunity to introduce new functionalities and elements to visualized data scene. Also new scheduling mechanism for job manager was presented, which should improve overall data processing performance by limiting delays on single jobs queue operations synchronization. New services for users data sharing have been proposed, to improve team work cooperation and knowledge exchange, changing application to general purpose research platform.

Quaternion lifting schemes This work has presented new approach for motion data analysis with multi-resolution tools for joints rotation data in quaternion form. Provided test results for data noise reduction and compression has proved very good properties for proposed data decomposition and representation for those applications, allowing to reduce signal noise significantly, providing high compression ratios with very limited signal distortions levels. Given new lifting scheme based on squad quaternion interpolation was tested for correct signal decomposition. Presented results have shown, that this lifting scheme works properly, allowing to reconstruct previously decomposed signal to its original form. Compression results and noise reduction properties are almost identical to another lifting scheme, with equivalent interpolation technique in quaternion tangent space. Application of second generation wavelets on quaternion signals opens new directions for motion analysis tools. It was proposed to extend lifting schemes application for dual quaternions to analyse additionally actor specific data according to his skeletal features. We have proposed also application of developed tools for more efficient and realistic motion blending. Proposed motion data decomposition can be used to create more reliable and descriptive motion features, used further for motion classification, recognition and prediction.

To perform presented tests a dedicated library was developed, based on MDE functionality and its dedicated data flow processing framework. Previously implemented solutions were wrapped with delivered helper classes to fit data flow model and logic. With this approach time required for obtaining results has been limited significantly, due to a well defined procedure of loading new data for processing and automated utilization of all available computation resources. Custom libraries have been automatically standardized, offering great flexibility in their usage with other application components and features. Moreover, moving to new data analysis tool made research easier, as all required tools are available in one application. Previously various standalone software (gnuplot, spread sheets, custom testing framework for quaternion based lifting schemes) for different tasks were used, what required many data formats conversions and manual operations. Now whole procedure is maximally simplified in MDE, providing great support for general purpose data processing with dedicated solutions, making motion analysis much simpler, efficient and reliable.

Summary It has been proven with a detailed description of MDE architecture and logic, along with presented extensions allowing simple and flexible creation of complex data processing models that MDE is a mature, efficient

and easily expandable data analysis tool. This statement is additionally proven with presented software development methodology and work organization. Proposed novel approach to motion data analysis with multi-resolution tools for quaternions provides great results in the field of data compression and noise reduction. Presented technique opens new directions in motion analysis, opening new directions for development of new motion analysis techniques. Presented research results were obtained with help of MDE software, proving that it can be easily adopted for custom solutions, rising their quality and standardizing them for cooperation with other built-in functionalities, making data analysis simpler and faster. Having proved those statements all thesis goals have been achieved with success. Additionally further research in motion analysis is now supported with MDE application, standardizing current tools and algorithms.

Appendix

The structure of attached CD is as follows:

- *thesis* - contains electronic version of this thesis in pdf format,
- *application* - contains a demo version of MDE application (installer),
- *libraries* - contains all libraries required to develop custom solutions for MDE, libraries are packed with *7zip* format,
- *tools* - contains tools required to configure provided projects,
- *src* - contains source code for presented implementation
  - *plug-in* - contains source code for MDE plug-in with wrapped tools for motion analysis,
  - *utilities* - contains general purpose algorithms (lifting scheme) and quaternions operations (interpolations, functions).
- *results* - experiments results in Excel format with presented charts,
- *instructions* - contains instructions how to configure plug-in projects, build them and test with application.

Application is delivered with built-in, demo user account, with limited access to motion data. Test data, used for experiments are already provided. Application installer provides automatically presented plug-in with motion analysis tools described in the thesis.

Streszczenie

1 Wprowadzenie

Nowe technologie pozwalają rejestrować coraz dokładniej dane o otaczającym nas świecie. To bezpośrednio przekłada się na zwiększone wymagania dla mocy obliczeniowych do przetwarzania dużych zbiorów danych oraz dla przestrzeni dyskowych niezbędnych do ich przechowywania. Proponuje się specjalizowane rozwiązania dla zbiorów danych o różnych wielkościach. Dla dużych zbiorów - tak zwanych *big data* [69] - stosuje się hurtownie danych i dedykowane, rozproszone bazy danych. Dla średnich i małych zbiorów proponowane są systemy algebry komputerowej (Computer Algebra Systems (CAS) [73]), arkusze kalkulacyjne oraz specjalizowane, naukowe języki programowania [21].

Obecnie analiza danych wykonywana jest nie tylko przez wykwalifikowaną kadrę w dziedzinie analizy matematycznej z doświadczeniem z zakresu programowania, ale również przez mniej doświadczonych użytkowników w tych dziedzinach, którzy bazują na gotowych rozwiązaniach i opracowaniach interesujących ich zagadnień statystyki.

Na rynku jest niewiele aplikacji dedykowanych ogólnemu przetwarzaniu danych, które wspierałyby użytkowników w często powtarzanych operacjach na danych: ładowanie, konwersja, proces przetwarzania czy raportowanie rezultatów analiz. Taki stan rzeczy spowalnia i utrudnia postęp w prowadzonych badaniach, ograniczając przy tym przepływ wiedzy pomiędzy członkami grup badawczych. Pociąga to za sobą potrzebę stworzenia nowych narzędzi, łatwych w użyciu i umożliwiających automatyczne, optymalne wykorzystanie dostępnych mocy obliczeniowych. Ich celem jest pomóc użytkownikom skupić się na celach, które chcą osiągnąć, a nie na technologiach i środkach, jakimi można te cele osiągnąć.

Przeglądając dostępne aplikacje wspierające analizę ruchu można zauwa-

żyć znaczące braki w ich funkcjonalności w zakresie przetwarzania danych. Pozwalają one jedynie na przeglądanie zgromadzonych danych i proste raportowanie. Uderza to w sporą grupę potencjalnych użytkowników, obejmującą lekarzy, trenerów sportowych i naukowców. Brak kompletnych rozwiązań dla analizy i przetwarzania ruchu ogranicza rozwój następujących dziedzin:

- medycyna (ortopedzi, neurochirurdzy i neurolodzy),
- sport (nowe miary postępu w treningach, spersonalizowane treningi, porównywanie techniki zawodników),
- bezpieczeństwo (rozpoznawanie osób po ich ruchach, wykrywanie potencjalnie niebezpiecznych sytuacji),
- rozrywka (realistyczne animacje syntezowanego ruchu, rozszerzona rzeczywistość).

Aby udostępnić użytkownikom kompletne narzędzia dla analizy danych w PJWSTK stworzono aplikację MDE. Jest ona zorientowana na wsparcie ogólnego przetwarzaniu danych, niezależne od ich formatu, typu i charakteru. MDE wprowadza dobrze zdefiniowane standardy dla efektywnego przetwarzania i analizy danych.

Bazując na wielorozdzielczych narzędziach zaproponowano nowe podejście dla analizy i przetwarzania danych ruchu w reprezentacji kwaternionowej [52] aby wesprzeć badania nad ruchem ludzkiego ciała. Metoda ta jest rozwinięciem obecnych rozwiązań [26, 27, 37, 38, 39, 10, 18, 7, 6, 40, 41, 5], operujących głównie na klasycznej analizie falkowej rotacji zapisanych jako niezależne kąty Eulera.

Celem pracy jest zaprezentowanie architektury i logiki aplikacji MDE - wyjaśnienie w jaki sposób standaryzują one procedurę analizy i przetwarzania danych przy pomocy wbudowanych funkcjonalności. Aby zweryfikować zalety MDE oraz możliwość łatwego użycia tego rozwiązania dla własnych potrzeb, spróbowano zaimplementować i przetestować zbiór nowych narzędzi dla analizy ruchu, opartych na wielorozdzielczej analizie danych ruchu w zapisie kwaternionowym.

2 Tezy pracy

W pracy sformułowano dwie tezy:

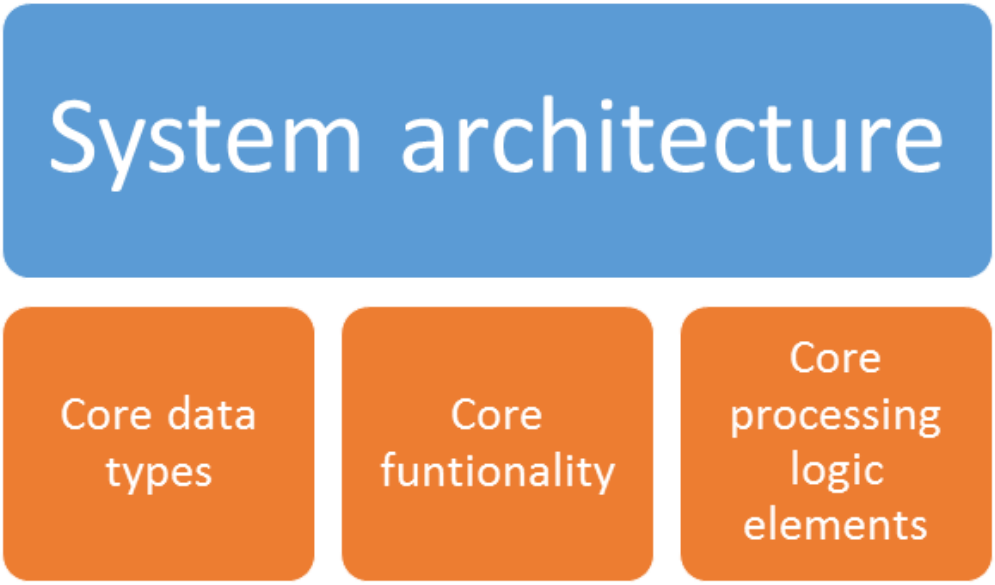
Teza 1 Aplikacja MDE jest dojrzałym, stabilnym narzędziem, opartym na ogólnym szkielecie (framework) dla przetwarzania danych. MDE może być elastycznie rozszerzany i konfigurowany na potrzeby specyficznych wymagań użytkowników poprzez dedykowany mechanizm wtyczek (plugins). Narzędzie to wspiera pełną, dobrze zdefiniowaną procedurę ładowania danych. Gwarantuje wydajne zarządzanie danymi w zunifikowany sposób, niezależnie od ich typu, oferując mechanizmy automatycznego i optymalnego wykorzystania dostępnych zasobów obliczeniowych. Przeglądanie danych jest znormalizowane dla wszystkich obsługiwanych typów danych, przedstawiając je z różnych perspektyw, specyficznych dla danego typu.

Teza 2 Zaproponowane narzędzia dla analizy ruchu, oparte o wielorozdzielczą analizę danych ruchu w reprezentacji kwaternionowej, oferują bardzo dobre właściwości dekompozycji danych dla zastosowań odfiltrowywania szumów oraz kompresji. Algorytmy te, zrealizowane w formie bibliotek ogólnego przeznaczenia, można w bardzo prosty sposób zaimplementować w ramach aplikacji MDE. Niezbędne dane do analizy oraz otrzymane wyniki eksperymentów są automatycznie obsługiwane przez specjalizowane moduły aplikacji, umożliwiając łatwe definiowanie i realizację badań wraz z analizą wyników. Wykorzystanie wcześniej przygotowanych bibliotek wymaga niewielkiego nakładu pracy programistycznej, dostosowując istniejące rozwiązania do akceptowalnych przez MDE interfejsów. Taka implementacja automatycznie gwarantuje optymalne wykorzystanie wszystkich zasobów obliczeniowych bez samodzielnego zarządzania wątkami.

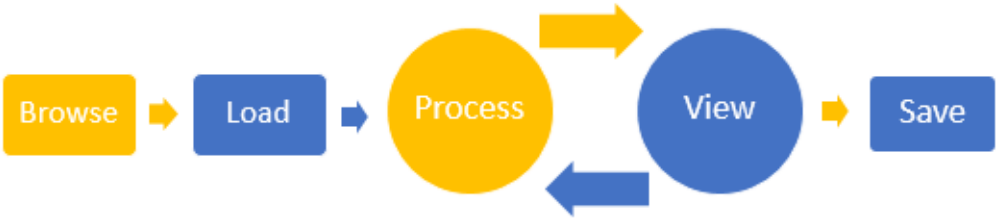
3 Charakterystyka aplikacji Motion Data Editor

Architektura MDE oparta jest na trzech podstawowych komponentach (Rysunek 6.1). Podstawowe typy danych obejmują:

- zunifikowany mechanizm przechowywania i zarządzania danymi w pamięci, niezależnie od ich typów dla języka programowania C++,
- generyczny typ dla danych o charakterze czasowym,
- typy danych narzędziowych ogólnego przeznaczenia.

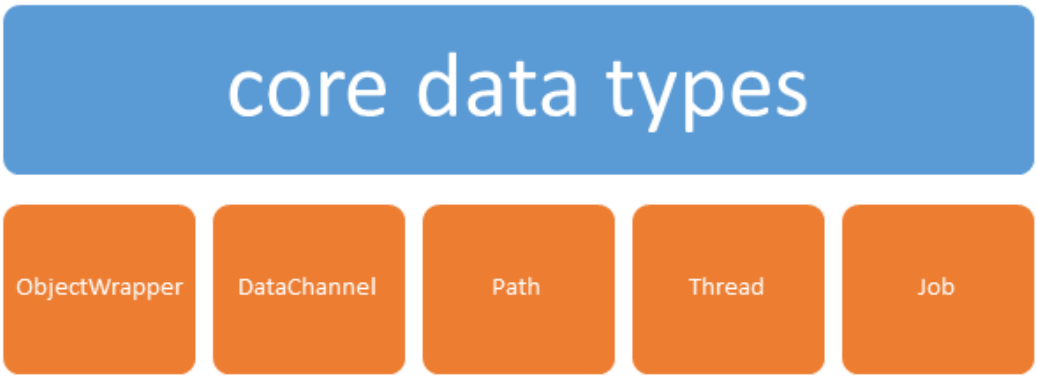


Rysunek 6.1: Ogólna architektura aplikacji



Rysunek 6.2: Ogólna procedura przetwarzania i analizy danych

MDE dostarcza dedykowany moduł pozwalający optymalnie wykorzystać wszystkie dostępne zasoby obliczeniowe na potrzeby realizacji różnych obliczeń. Dodatkowo, możliwe jest zunifikowane logowanie wiadomości na temat aktualnego stanu aplikacji. Dostarczone rozwiązania wspierają tworzenie i projektowanie specjalizowanych widoków dla realizowanej logiki przetwarzania. Aplikacja oferuje mechanizm wtyczek, pozwalający rozszerzać jej funkcjonalność o nowe możliwości. Wbudowane mechanizmy zarządzające elementami logiki przetwarzania danych wspierają proces analizy danych (Rysunek 6.2). Podstawowe elementy logiki przetwarzania danych obejmują obiekty dostarczające i ładujące dane do aplikacji (źródła i parsery), wizualizujące dane (wizualizatory) oraz szeroko pojęte nowe funkcjonalności aplikacji w formie serwisów.



Rysunek 6.3: Podstawowe typy danych

3.1 Podstawowe typy danych

Dwoma najważniejszymi typami danych w MDE są OW oraz *DataChannel*. Przedstawiona architektura aplikacji jest w całości oparta na tych obiektach oraz ich właściwościach. Wspierają one uniwersalny mechanizm zarządzania dowolnymi typami danych oraz standaryzują obsługę danych o charakterze czasowym. Rysunek 6.3 przedstawia wszystkie bazowe typy danych MDE.

3.1.1 ObjectWrapper

OW wprowadza nowe podejście dla ogólnego zarządzania danymi dowolnych typów dla języka programowania C++. Koncepcja oparta jest na wspólnym typie danych dla wszystkich obiektów, znanym z takich języków programowania jak *C#* oraz *Java*, gdzie bazą dla wszystkich danych jest ogólny typ *Object*. OW oparty jest na programowaniu generycznym. Wprowadza jednolity interfejs dla zapytań o informację o typach dla opakowanych danych, wypakowywanie danych oraz ich inicjalizację. Dodatkowo OW wspiera mechanizm leniwej inicjalizacji danych oraz system meta-danych w formie pary literałów $[klucz, wartosc]$. Mechanizm OW jest konfigurowalny w ramach dwóch polityk:

- wskaźnika - typ wskaźnika używanego do przechowywania danego typu danych,
- klonowanie - sposób tworzenia kopii danych.

Opcje te pozwalają dostosować OW do obsługi dowolnych typów danych, od wbudowanych podstawowych typów dla języka C++, po własne typy danych, zdefiniowane na potrzeby konkretnych zastosowań. OW wspiera informację o hierarchii opakowanych typów. Ta funkcjonalność działa tylko

dla typów pochodnych, dla których opis hierarchii typów bazowych jest dostępny. Obecna implementacja tej funkcjonalności pozwala obsługiwać jedynie liniowe hierarchie typów, więc dla dziedziczenia wielobazowego tylko jeden typ bazowy może być użyty. Mechanizm ten wykorzystywany jest do filtrowania danych według typu oraz przedstawiania danych z różnych perspektyw, wynikających z ich hierarchii dziedziczenia.

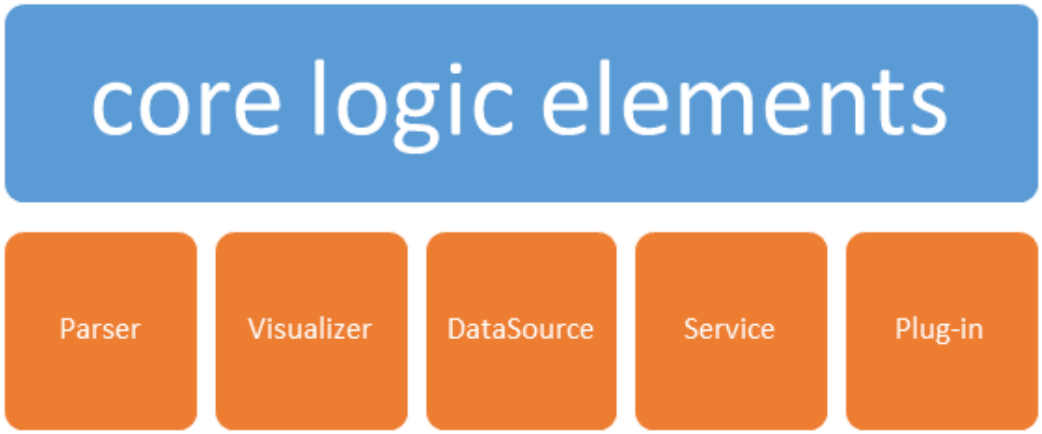
3.1.2 Dane o charakterze czasowym

Generyczny typ *DataChannel* wprowadzono aby zunifikować obsługę danych o charakterze czasowym. Pozwala on na przechowywanie próbek w formie $[indeks(czas) \rightarrow wartosc]$, gdzie indeksy są unikalne i mają ściśle zdefiniowaną relację mniejszości. Indeksy są ponadto niemodyfikowalne, ponieważ porządkują dane w domenie czasu. Ładowanie danych do obiektów typu *DataChannel* musi odbywać się w porządku rosnącym dla czasu, ponieważ wewnętrznie dane są dodatkowo opatrzone indeksami reprezentowanymi przez liczby całkowite, odpowiadające numerom próbek w kolejności w jakiej zostały załadowane. *DataChannel* oparty jest na łączeniu interfejsów oraz dedykowanych im implementacji (mix-ins). Pozwala to zmieniać charakter danych bez potrzeby ich kopiowania, poprzez przykrywanie instancji danych dedykowanym interfejsem, zmieniającym typ przechowywanych w *DataChannel* wartości. Obiekty typu *DataChannel* można podzielić na dwie kategorie ze względu na wartości znaczników czasu:

- równo-odległe indeksy czasu - stała różnica pomiędzy czasami kolejnych próbek danych,
- nieregularne indeksy czasu - zmienna różnica pomiędzy czasami kolejnych próbek danych.

Ta własność używana jest do optymalnego dostępu do wartości dla zapytań poprzez indeks czasowy danych.

Jako rozszerzenie dla obiektów *DataChannel* zaprojektowano kilka pomocniczych typów, pozwalających traktować dane dyskretne w sposób ciągły. Oferują one gotowe metody interpolacji oraz możliwość dostarczania własnych interpolatorów. Klasy te są używane kiedy *DataChannel* przeglądany jest w większej rozdzielczości niż wynika to z surowych danych. Struktura wielu algorytmów wymaga zapytań o indeksy próbek spoza danego zakresu. Aby wesprzeć tego rodzaju operacje zaprojektowano kilka schematów ekstrapolacji dla *DataChannel*:



Rysunek 6.4: Bazowe elementy logiki przetwarzania danych

- wyjątki - przy zapytaniu o dane spoza dostępnego zakresu rzucane są wyjątki,
- wartości brzegowe - odpowiednio najmniejsza lub największa próbka są powielane dla wszystkich zapytań spoza zakresu,
- symulacja periodyczności danych - zapytanie o dane spoza zakresu jest zaokrąglane do właściwego przedziału danych na podstawie odległości pomiędzy brzegowymi próbkami.

Obiekty z danymi czasowymi często są rozszerzane o stan opisujący ich aktualny czasu. Dla *DataChannel* wprowadzono typ *Timer*. Łączy on stan czasu z *DataChannel* pozwalając pobierać dane dla aktualnie ustawionego w *Timer* czasu. Dodatkowo umożliwia on prowadzenie niezależnego stanu czasu dla tych samych danych bez potrzeby ich kopiowania.

3.2 Bazowe elementy logiki przetwarzania danych

Rysunek 6.4 przedstawia podstawowe elementy logiki przetwarzania danych. Odpowiedzialne są one za ładowanie nowych danych do aplikacji, przeglądanie danych oraz rozszerzanie MDE o nowe funkcjonalności poprzez dedykowany mechanizm wtyczek.

3.2.1 Parsery

Koncepcja parserów powstała aby znormalizować proces wypakowywania danych z różnych źródeł (najczęściej plików bądź strumieni). Każdy parser dostarcza informacji o obsługiwanych źródłach oraz typach danych, które

potencjalnie może z nich dostarczyć. Ponadto, każdy parser musi charakteryzować się minimum jedną z dwóch funkcjonalności obsługujących różne sposoby dostępu do danych:

- indywidualne operacje wejścia wyjścia - parser sam wykonuje niskopoziomowe operacje dostępu do danych,
- obsługa strumieni danych - parser dostarcza danych z ściśle zdefiniowanych strumieni danych.

Taka realizacja parserów pozwala na optymalizację ładowania danych dla plików. Plik może być wczytany raz do pamięci i dostarczony w formie strumienia do parsera. Możliwe jest również obsłużenie pliku przez kilka parserów, wzajemnie się uzupełniających pod kątem wypakowywanych danych.

Mechanizm parserów współpracuje z mechanizmem leniwej inicjalizacji OW, gdzie faktyczne parsowanie przeprowadzane jest w momencie odpytywania OW o dane. Pozwala to ograniczyć potrzebną dla danych pamięć, gdyż nie wszystkie dane muszą być używane podczas analizy i przetwarzania.

3.2.2 Źródła danych

Idea źródeł danych (*DataSource*) została zaproponowana, aby ujednolicić sposób przeglądania dostępnych danych na potrzeby analizy i przetwarzania. Obiekty tego typu odpowiedzialne są za wskazywanie i dostarczanie danych w ich kontenerach (ścieżka do lokalnego pliku, ściągnięcie archiwum z serwera FTP, odpytanie bazy danych, otwarcie połączenia z danym urządzeniem) oraz ładowanie danych z kontenerów do aplikacji (najczęściej z pomocą parserów i dedykowanych menadżerów). Źródła danych pozwalają MDE obsługiwać specyficzne sposoby dostarczania danych, których przykładem może być HMDB [19] - scentralizowana usługa dostarczająca danych ruchu nagranych w HML.

3.2.3 Wizualizatory

Koncepcja wizualizatorów (*Visualizer*) wprowadza warstwę abstrakcji dla przeglądania danych. Obiekty tego typu obsługują mechanizm serii danych, pozwalających przeglądać rożne typy danych. Zaproponowane rozszerzenie dla serii danych umożliwia prezentację danych o charakterze czasowym, dla których wprowadzono dodatkowe operacje: skalowanie w czasie (scale), przesunięcie w czasie (offset) oraz modyfikację aktualnego czasu dla danych w serii. Wizualizatory mogą wspierać różne ilości serii danych, zależnie od

charakteru prezentowanych danych i właściwości wizualizatora. Wśród serii danych można wyróżnić aktywną serię danych, aktualnie zarządzaną przez użytkownika, dla której wizualizator może dostarczać dodatkowych funkcjonalności, specyficznych dla danego typu. Każdy wizualizator jest opisany typami danych, które może obsłużyć, co ułatwia użytkownikom przeglądanie danych za pomocą różnych wizualizatorów, prezentujących inne perspektywy danych.

3.2.4 Serwisy

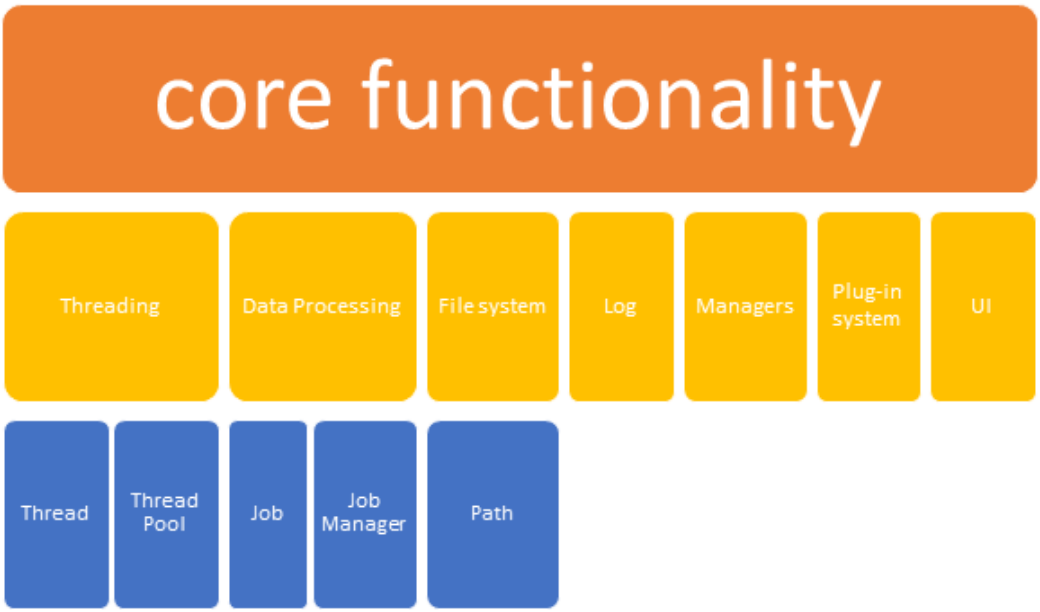
Serwisy wprowadzono jako uogólnienie nowych, szeroko pojętych funkcjonalności aplikacji. Trudno jest zaproponować zbiór wspólnych operacji dla serwisów, ponieważ mogą one prezentować całkowicie odmienne rozszerzenia aplikacji. Z tego też powodu serwisy są najbardziej uprzywilejowanymi elementami logiki MDE, mającymi dostęp do wszystkich modułów logiki aplikacji, aby umożliwić tworzenie różnorodnych, nowych funkcjonalności aplikacji.

3.2.5 Wtyczki

Każda analiza danych operuje na innych typach danych oraz narzędziach w ramach ściśle zdefiniowanej, powtarzalnej procedury. Aby ułatwić użytkownikom dostosowywanie uniwersalnego mechanizmu wspierającego przetwarzanie danych, opartego na logice zaimplementowanej w MDE, wprowadzono system wtyczek, pozwalających rozszerzać możliwości aplikacji. Wtyczki pozwalają rejestrować w aplikacji nowe typy danych, parsery, wizualizatory, źródła danych i serwisy. Ponieważ wtyczki reprezentowane są jako niezależne, dynamiczne biblioteki, ładowane podczas startu aplikacji, należało wprowadzić dedykowane rozwiązania, weryfikujące ich kompatybilność z MDE. Brane są tutaj pod uwagę wersje bibliotek zależnych, używanych w aplikacji i rozwiązaniach dostarczanych z wtyczkami, wersja publicznego interfejsy aplikacji oraz wersja interfejsu użyta do budowy wtyczki. Dodatkowo sprawdzany jest typ kompilacji wtyczki i aplikacji (debug lub release).

Timeline Jedną z najistotniejszych wtyczek dla MDE jest *Timeline*. Serwis ten pozwala na synchronizację danych o charakterze czasowym w ramach ściśle zdefiniowanej hierarchii. *Timeline* umożliwia edycję właściwości czasu dla danych, bez konieczności ich kopiowania. Wszystkie operacje na czasie są automatycznie propagowane na całą hierarchię. Nowymi operacjami *Timeline* są:

- podziel - tworzone są dwa niezależne zestawy danych dla zadanego punktu podziału,



Rysunek 6.5: Wbudowane funkcjonalności

- scal - dwa niezależne zestawy danych są połączone w zadanej kolejności.

Operacje te pozwalają ograniczyć zakres analizowanych danych w domenie czasu do zadanego okna czasu. *Timeline* dostarcza również mechanizmu do odtwarzania danych o charakterze czasowym w wizualizatorach.

3.3 Wbudowane funkcjonalności

Rysunek 6.5 przedstawia wbudowane funkcjonalności MDE, tworzące warstwę abstrakcji dla operacji specyficznych dla systemów operacyjnych, systemów plików oraz tworzenia GUI.

3.3.1 Obsługa systemu plików

Zaproponowano zunifikowany typ do obsługi ścieżek dla systemu plików - *Path*, ponieważ większość danych dostarczana jest w formie plików. Jest on bazą dla zbioru podstawowych operacji na plikach i folderach. MDE dostarcza w ten sposób informacji o specyficznych zasobach aplikacji.

3.3.2 Log

Aby umożliwić prezentację istotnych komunikatów użytkownikowi, zaprojektowano mechanizm hierarchicznych logów. Oparty jest on na kilkupozio-

mowym statusie informacji, gdzie użytkownik może zdefiniować minimalny poziom, o którym ma być powiadamiany. Pozostałe informacje są automatycznie filtrowane i usuwane. Log ma konfigurowalne ujścia dla informacji, od prostej konsoli, przez sformatowane pliki tekstowe, po graficzne okno. Każda wtyczka inicjowana jest dedykowanym poziomem loga, pozwalającym zidentyfikować źródło i kontekst wiadomości.

3.3.3 Wielowątkowość

Optymalne wykorzystanie zasobów obliczeniowych komputera wymaga poprawnego użycia wątków do równoległej realizacji zadań. Ponieważ zarządzanie wątkami jest operacją specyficzną dla systemów operacyjnych, w MDE wprowadzono typ *Thread*, tworzący warstwę abstrakcji niezależną od użytkowanej platformy. Aby kontrolować ilość tworzonych wątków dla celów diagnostycznych oraz zapewnienia stabilności i wydajności aplikacji, wprowadzono typ *ThreadPool*. Obiekt ten pozwala na tworzenie ściśle zdefiniowanej ilości wątków. Kiedy wartość ta zostanie osiągnięta, nie można utworzyć nowych wątków do momentu ukończenia obliczeń przez już stworzone wątki i zwolnienia ich zasobów. Aby zminimalizować narzut związany z tworzeniem nowych wątków, *ThreadPool* utrzymuje zdefiniowaną, minimalną ilość wolnych wątków jako bufor dla najbliższych zapytań o nowe wątki, starając się ponownie wykorzystać wątki, które zakończyły swoje zadania.

3.3.4 Przetwarzanie danych

Ciągłe tworzenie nowych wątków oraz inicjowanie ich działania może drastycznie pogorszyć wydajność aplikacji. Aby temu zapobiec zaprojektowano mechanizm zarządzający i kolejkujący zadania do wykonania. Ściśle zdefiniowana grupa wątków odpowiada za przetwarzanie zleconych zadań. *Job* i *JobManager* pozwalają efektywnie wykorzystać dostępne zasoby obliczeniowe dzięki utrzymywaniu optymalnej ilości wątków przetwarzających, która dla współczesnych procesorów wynosi $rdzenie\_procesora * 2 - 1$. Jeden z wątków pozostaje wolny na potrzeby obsługiwania graficznego interfejsu użytkownika. Zadania pobierane są ze wspólnej kolejki przez wyznaczone wątki, w kolejności w jakiej zostały dodane.

3.3.5 Menadżery

W centrum logiki aplikacji leżą dedykowane menadżery dla podstawowych elementów logiki przetwarzania i analizy danych. Część z nich została już omówiona (wątki i zadania). Funkcjonalność menadżerów została zdekomponowana na operacje niemodyfikujące i modyfikujące dany obiekt. Takie

podejście pozwala udostępniać globalnie niemodyfikującą część operacji oraz dostarczać lokalnie funkcjonalności pozwalające zmieniać stan menadżerów do ściśle zdefiniowanych elementów architektury, gdzie takie zachowanie zostało przewidziane. Taka dekompozycja wprowadza porządek w logice, określając dokładnie obszary odpowiedzialności poszczególnych modułów oraz ich potencjalne możliwości, eliminując niepotrzebne udostępnianie wszystkich funkcjonalności.

Operacje na menadżerach są synchronizowane. Aby zwiększyć wydajność grupy operacji na danym menadżerze oraz wprowadzić izolację dla tych operacji zaprojektowano mechanizm transakcji. Szeregują on inne operacje na danym obiekcie do momentu zakończenia aktualnej transakcji. Efekty transakcji mogą być zatwierdzone - stan obiektu zostaje trwale zmodyfikowany, lub mogą zostać wycofane, gdzie wszystkie zmiany na danym obiekcie są anulowane, a jego stan sprzed transakcji jest przywrócony.

3.4 Proces Ciągłej Integracji (CI)

MDE oparte jest na wielu zewnętrznych bibliotekach, co pozwoliło zaoszczędzić czas na implementowanie dobrze znanych i przetestowanych rozwiązań. Niestety, ilość zależności stopniowo utrudniała rozwój aplikacji, gdyż kompilacja potrzebnych bibliotek zajmowała coraz więcej czasu i była miejscem wielu błędów ludzkich. Procedurę CI wprowadzono, aby w pierwszej kolejności zautomatyzować budowę zewnętrznych bibliotek. Potem rozszerzono ją o budowę własnych projektów, testy i kontrolę jakości kodu. Dla poprawnie zweryfikowanych artefaktów CI generuje instalatory gotowych produktów, udostępniając je użytkownikom do instalacji i aktualizacji. Dodatkowo tworzona jest też dokumentacja techniczna wewnętrznych bibliotek.

3.5 Rozwój aplikacji

Aby zapewnić wysoką jakość kodu dla MDE wprowadzono zbiór dobrze zdefiniowanych i zalecanych praktyk programistycznych:

- programowanie generyczne,
- wzorce projektowe,
- dekompozycja i minimalna zależność między modułami,
- minimalne komentarze nagłówków,

- samo-komentujący się kod implementacji wraz z ściśle zdefiniowanym stylem kodowania,
- dostarczanie prostego, minimalnego, ale kompletnego API dla użytkowników aplikacji.

Zastosowanie tych metod pozwoliło utrzymać elastyczność MDE na nowe rozwiązania, ograniczyć możliwości potencjalnych błędów oraz skrócić czas na poprawę błędów.

4 Potokowe przetwarzanie danych

Celem zapewnienia użytkownikom konfigurowalnego, łatwego w użyciu narzędzia do projektowania i realizacji dowolnych schematów przetwarzania danych opracowano dedykowaną wtyczkę. Wprowadza ona do MDE serwis pozwalający tworzyć dobrze zdefiniowane potoki przepływu danych. Ich modele oparte są na strukturze grafu. W ramach modelu można wyróżnić trzy typy węzłów ze względu na ich funkcję:

- źródła - dostarczają nowych danych do potoku,
- procesory - odpowiadają za przetwarzanie danych,
- terminatory - utrwalają wyniki przetwarzania.

W przeciwieństwie do grafów, węzły w potoku nie łączą cię bezpośrednio ze sobą. Wprowadzono nowy element - tak zwany *pin* - poprzez który węzły mogą być ze sobą łączone. Każdy węzeł opisany jest stałą konfiguracją pinów, którą można porównać do sygnatury funkcji w językach programowania. W ten sposób rozumiane piny opisują listę parametrów wejściowych funkcji oraz jej zwracane wartości. Aby dokładniej odzwierciedlić definiowane węzłów jako funkcje wprowadzono dodatkowe rozszerzenia do opisu modelu na poziomie pinów, gdzie można scharakteryzować piny dodatkowymi właściwościami:

- wymagany - pin wejściowy musi być połączony z innym pinem, aby zapewnić minimalną funkcjonalność węzła (argument wymagany),
- zależny - na danym pinie można spodziewać się rezultatów wyłącznie jeśli jego piny zależne są podłączone.

Dostosowanie opracowanych już algorytmów przetwarzania danych na potrzeby potoków polega na opakowaniu ich interfejsami węzłów (najczęściej

procesorów). Logika przetwarzania gwarantuje, że przetwarzanie w potoku jest automatycznie zrównoleglone, poprzez delegowanie obliczeń w formie zadań do *JobManagera*. Dzięki temu użytkownik otrzymuje narzędzie standaryzujące proces przetwarzania i automatycznie wykorzystujące całą dostępną moc obliczeniową na potrzeby własnych obliczeń, bez potrzeby indywidualnego tworzenia i zarządzania wątkami oraz ich synchronizacji.

Aby uprościć wykorzystanie potokowego przetwarzania danych użytkownikom mniej biegłym w programowaniu, zaprojektowano graficzne środowisko programowania. Moduł ten umożliwia wizualną realizację modeli przetwarzania poprzez łączenie prostych bloków funkcyjnych, realizujących poszczególne zadania. Użytkownik informowany jest na bieżąco o możliwościach łączenia pinów, wynikających z kompatybilności typów danych które reprezentują i reguł łączenia modelu. Wizualne środowisko pozwala również na grupowanie połączonych i skonfigurowanych już węzłów, tworząc bardziej rozbudowane funkcjonalności, które można użyć w późniejszym czasie bez potrzeby ponownego ich tworzenia. Takie złożone węzły mogą być wymieniane pomiędzy użytkownikami.

5 Wielorozdzielcza analiza danych ruchu w zapisie kwaternionowym

Zaproponowano nowe podejście do analizy danych ruchu na bazie falek drugiej generacji - schemat liftingu [62] i kwaternionowego zapisu rotacji dla stawów. Schemat liftingu pozwala na dekompozycje danych do wielopoziomowej reprezentacji, przedstawionej jako sumę reprezentacji szczegółowych i zgrubnych. Operacja ta oparta jest na trzech krokach, powtarzanych rekursywnie:

blok podziału dzieli dane na dwa podzbiory dla próbek indeksowanych wartościami parzystymi i nieparzystymi,

blok predykcji estymuje próbki indeksowane wartościami nieparzystymi przy pomocy próbek indeksowanych wartościami parzystymi. Estymowana próbka zastępowana jest różnicą pomiędzy jej wartością a proponowaną estymatą.

blok uaktualnienia aktualizuje wartości próbek parzystych, aby ich średnia odpowiadała wejściowej średniej sygnału.

W porównaniu do kątów Eulera, kwaterniony opisują rotacje w sposób kompaktowy, oferując przy tym lepsze metody interpolacji [11] z punktu widzenia realizacji rotacji:

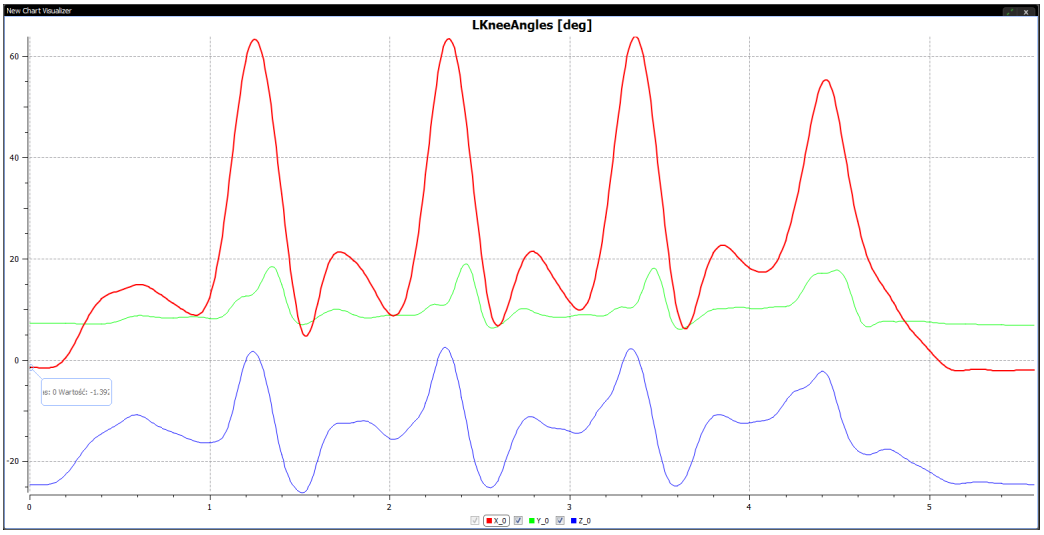
- slerp,
- squad.

Na bazie tych metod i własności kwaternionów zaproponowano szereg schematów liftingu. Głównym problemem przy tworzeniu tych algorytmów wydaje się być miara odległości oraz interpretacja wartości średniej dla kwaternionów, które nie są jednoznacznie określone [45]. Schemat liftingu wymaga jednak tylko zachowania wartości średniej dla danych na każdym poziomie rozdzielczości, dlatego można przyjmować dowolne interpretacje dla średniej, aby najlepiej pasowały do poszczególnych zastosowań. Zaproponowane schematy operują również na kwaternionach rozumianych jako wektory przestrzeni $\mathbb{R}4$, aby zweryfikować ich poprawność i możliwość zastosowania dla analizy ruchu. Ponadto przedstawiona metoda interpolacji kwaternionów w przestrzeni stycznej ($\mathbb{R}3$) wskazuje kierunek dla nowych metod interpolacji dla kwaternionów na bazie większej ilości próbek dla osiągnięcia gładkiej reprezentacji ruchu.

Aby zweryfikować zaproponowane narzędzia wielorozdzielczej analizy ruchu wprowadzono pojęcie odległości pomiędzy dwoma seriami danych w reprezentacji kwaternionowej. Odległość ta oparta jest na skumulowanej wartości kątów obrotu dla różnicy (iloraz) pomiędzy kolejnymi parami kwaternionów w obu seriach danych. Wartość ta przedstawiona jest w radianach i jest tym mniejsza im kwaterniony przedstawiają bardziej zbliżone obroty, zmierzając do 0 dla identycznych danych.

5.0.1 Testy

Wykonano szereg testów sprawdzających poprawność wszystkich transformat - jakość detali oraz rekonstrukcja sygnału po dekompozycji. Rysunek 6.6 przedstawia dane testowe. Dodatkowo, przetestowano zastosowanie rezultatów dekompozycji dla redukcji szumów oraz kompresji danych ruchu. Rysunek 6.7 przedstawia detale danych dla schematu liftingu opartego na interpolacji metodą *squad*. Po zastosowaniu proponowanej stratnej metody kompresji na przedstawionych detalach osiągnięto poziom kompresji równy 87.5% przy średnim zniekształceniu sygnału po dekompresji na poziomie 0.002 radiana na kąt obrotu kwaternionu (Rysunek 6.10). Równie dobrze wypadają testy odfiltrowywania szumów. Dla zniekształconych danych po



Rysunek 6.6: Dane testowe - lewe kolano 26-letniego, zdrowego mężczyzny

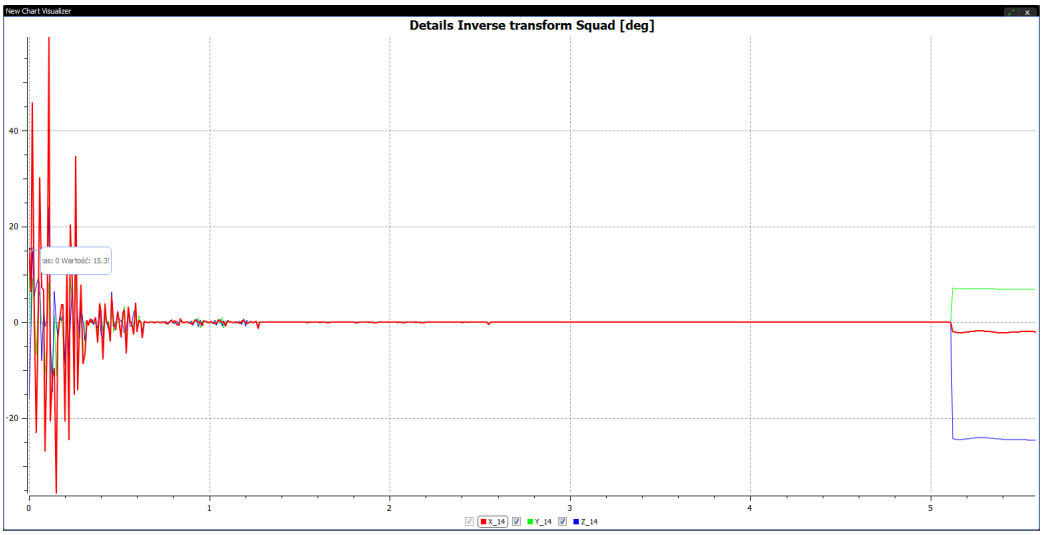
dodaniu białego szumu opisanego funkcją Gaussa dla $\sigma = 5°$ (Rysunek 6.8) udało się zrekonstruować dane ze zredukowanymi zakłóceniami o blisko 50% (Rysunek 6.9). Przedstawione wyniki obejmują cały zakres sygnału, chociaż schemat liftingu operował tylko na części początkowych próbek, będącej największą możliwą potęgą liczby 2, wynikającą z binarnego podziału w bloku split. Dla porównania przedstawiano również niezmodyfikowaną część danych.

5.0.2 Implementacja

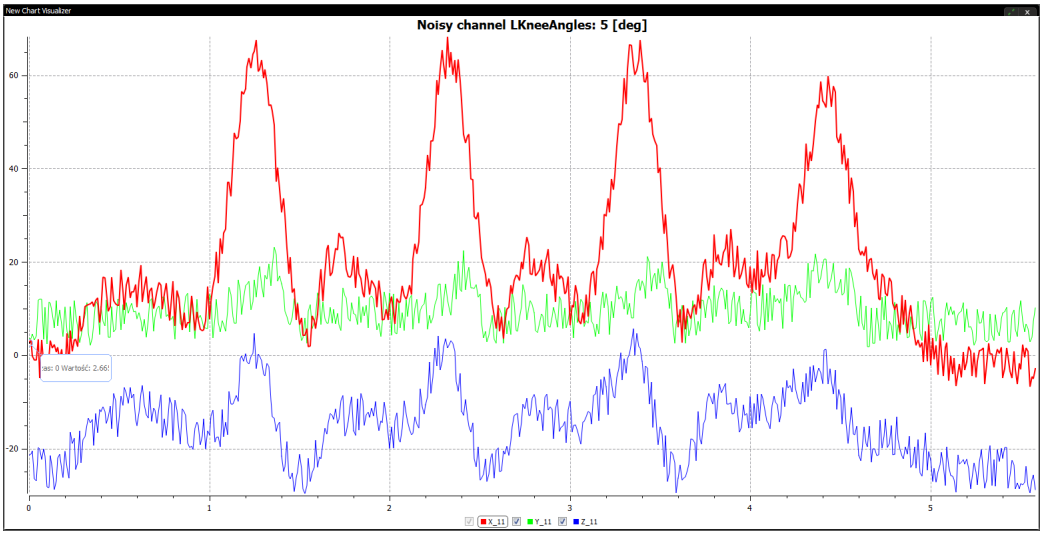
Wszystkie zaproponowane schematy liftingu zostały zaimplementowane w formie algorytmów ogólnego użytku. Zostały one opakowane interfejsami dla węzłów przetwarzających potokowego przetwarzania danych w MDE. Dane do analizy pobrano z dostępnej bazy danych ruchu (HMDB) poprzez wbudowane źródło danych. Eksperymenty i wyniki powstały w ramach możliwości oferowanych przez MDE. Dedykowany wizualizator pozwolił przedstawić dane w formie kątów Eulera, które są łatwiejsze do analizy wizualnej od kwaternionów.

6 Oryginalne wyniki

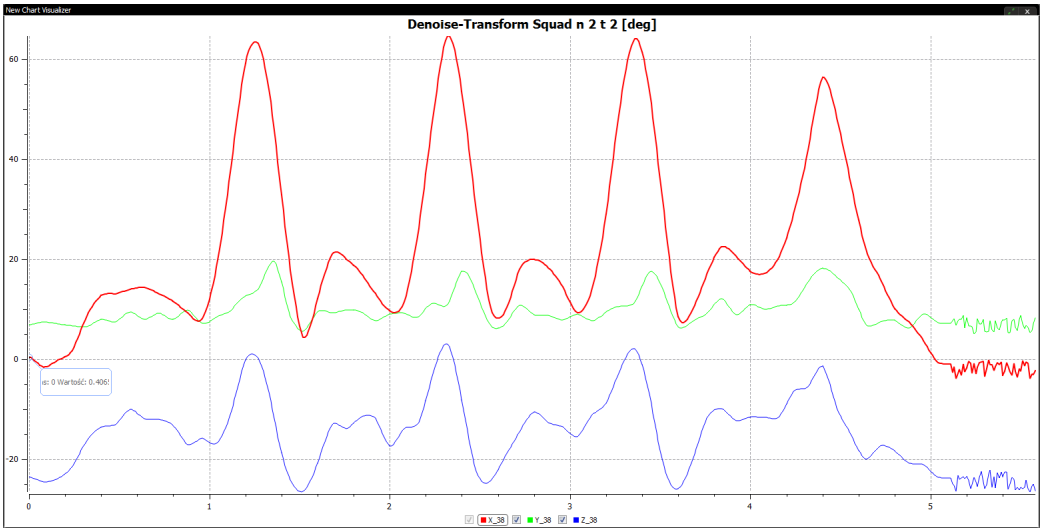
Jako główne, autorskie elementy przedstawione w pracy należy wymienić następujące zagadnienia:



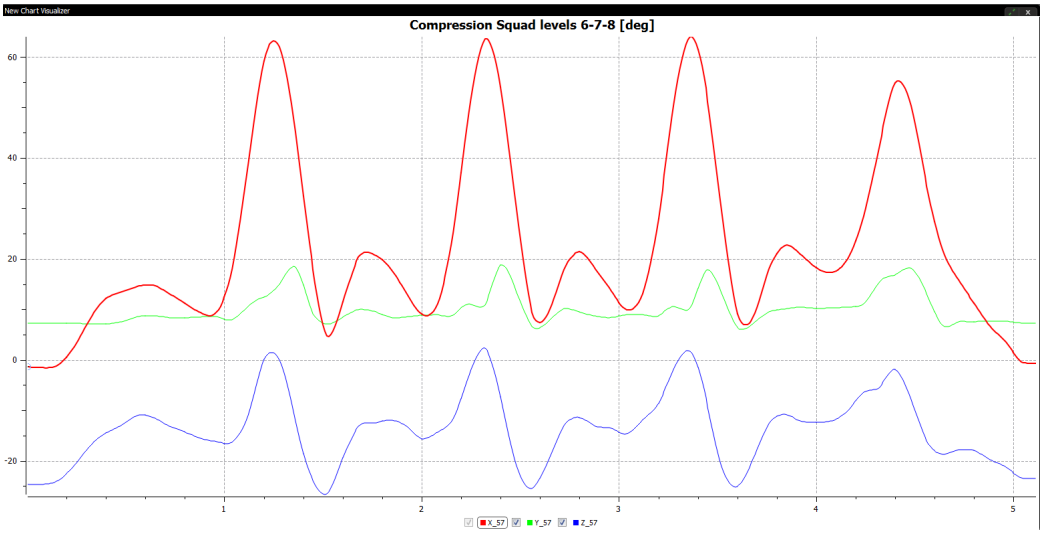
Rysunek 6.7: Detale wszystkich rozdzielczości dla interpolacji metodą *squad*



Rysunek 6.8: Zniekształcone dane przez dodanie białego szumu Gaussa z $\sigma = 5°$



Rysunek 6.9: Odfiltrowany szum schematem liftingu opartym na *squad* dla wartości progowej kątów obrotu = 2° i zmodyfikowanych rozdzielczościach: 6, 7, 8



Rysunek 6.10: Wyniki dekompresji dla schematu liftingu opartego na *squad* po usunięciu wszystkich detali dla rozdzielczości: 6, 7, 8 (87,5% wszystkich danych)

- dla rozwoju aplikacji MDE:
  - zaprojektowanie architektury i logiki dla MDE, które unifikują proces analizy i przetwarzania danych,
  - wprowadzenie koncepcji *DataChannel* z pomocniczymi funkcjonalnościami, które standaryzują obsługę danych o charakterze czasowym,
  - zaproponowanie procedury leniwej inicjalizacji poprzez parsery oraz ich dualnej natury dla obsługi źródeł danych, które pozwalają przyspieszyć ładowanie danych do aplikacji,
  - opracowanie obiektu typu *ThreadPool* w ramach MDE, pozwalającego kontrolować ilość wątków aplikacji,
  - wprowadzenie koncepcji przetwarzania danych opartej o *Job* i *JobManager* umożliwiającej automatycznie wykorzystywać wszystkie dostępne zasoby obliczeniowe,
  - zaprojektowanie *Timeline* jako warstwy abstrakcji dla operacji na czasie,
  - opracowanie procedury weryfikacji kompatybilności ładowanych wtyczek z aplikacją oraz ich inicjalizacji kontekstem aplikacji,
  - wprowadzenie koncepcji hierarchicznego modelu logowania wiadomości,
  - zaprojektowanie elastycznego i prostego w użyciu modułu potokowego przetwarzania danych wraz z graficznym środowiskiem programowania,
  - wprowadzenie procesu Ciągłej Integracji, wspierającego rozwijanie własnych projektów poprzez automatyzacje wielu zadań,
  - opracowanie dedykowanych skryptów na potrzeby konfiguracji nowych projektów, wspierających proces wyszukiwania i używania bibliotek zewnętrznych oraz zarządzania zależnościami pomiędzy projektami
- dla badań nad analizą ruchu:
  - opracowanie nowego podejścia do analizy danych ruchu na bazie schematu liftingu oraz interpolacji kwaternionów metodą *squad*,
  - opracowanie testów i ich wyników dla porównania różnych schematów liftingu na kwaternionach orz ich zastosowania dla kompresji i filtrowania szumu,

– przykładowa implementacja własnych narzędzi w ramach potokowego przetwarzania danych dla MDE.

7 Podsumowanie

W pracy przedstawiono w jaki sposób aplikacja MDE stworzona w PJWSTK wspiera ogólne przetwarzanie i analizę danych. Opisano standardy dla analizy ruchu, jakie wprowadza MDE poprzez zaproponowaną architekturę i logikę. Omówiono mechanizmy wspierające wydajne przetwarzanie danych z automatycznym wsparciem dla wielowątkowego przetwarzania oraz nowym mechanizmem jednolitego przechowywania i zarządzania danymi dowolnego typu dla C++. Rozwiązania te dowodzą doskonałych właściwości MDE jako platformy dla różnego typu projektów badawczych, gdzie wydajne i proste przetwarzanie danych stanowią kluczowym element badań. Ponadto pokazano jak prosto można rozszerzać możliwości aplikacji o własne rozwiązania poprzez dedykowany mechanizm wtyczek. W tym celu zaprezentowano możliwość prostego dostosowania zewnętrznych bibliotek na potrzeby potokowego przetwarzania danych w ramach MDE (wcześniej stworzone biblioteki ogólnego użytku z przedstawionymi schematami liftingu zostały opakowane niezbędnymi interfejsami dla potokowego przetwarzania danych). Wszystkie przedstawione eksperymenty i ich rezultaty zostały przeprowadzone w graficznym środowisku programowania, ułatwiającym tworzenie schematów przetwarzania danych. Implementując gotowe rozwiązania udało się dodatkowo zdekomponować je na prostsze operacje, kompatybilne z innymi modułami przetwarzającymi. Ponadto uzyskano optymalne wykorzystanie zasobów obliczeniowych bez dodatkowych nakładów pracy, co pozwoliło znacząco skrócić czas potrzebny na dostosowanie się do nowego mechanizmu przetwarzania danych, jak i czas potrzebny na przeprowadzenie niezbędnych testów.

Postawione w pracy tezy zostały udowodnione poprzez szczegółowy opis architektury oraz pozytywne wyniki przeprowadzonych testów dla proponowanych narzędzi analizy ruchu. Udało się również osiągnąć wszystkie postawione w pracy cele implementując istniejące wcześniej rozwiązania w ramach MDE i przeprowadzając testy za pomocą potokowego przetwarzania danych. Przedstawione rezultaty pracy mogą być użyte jako baza dla wielu istniejących algorytmów przetwarzania danych ruchu. Pokazano szereg zalet i zastosowań dla danych w reprezentacji wielorozdzielczej z użyciem proponowanych schematów liftingu. Ponadto, przedstawiono możliwości zastosowań MDE na potrzeby prac badawczych, jako narzędzia wielofunkcyjnego, wspierającego ładowanie, przetwarzanie i analizę dowolnego rodzaju danych.

Wskazano również nowe kierunki dla rozwoju MDE w niedalekiej przyszłości, które jeszcze bardziej uproszczą proces analizy i przetwarzania danych oraz umożliwią swobodną wymianę wiedzy pomiędzy użytkownikami aplikacji.

List of Figures

2.1 Human musculo-skeletal system 8
2.2 Vicon Polygon 17
2.3 Mokka 18

3.1 Common data processing pipeline 21
3.2 System architecture overview 24
3.3 Core data types 25
3.4 OW class hierarchy 27
3.5 OW lazy initialization on data query 28
3.6 *DataChannel* types 31
3.7 *DataChannel* concept 32
3.8 Core processing logic elements 35
3.9 Core functionality 42
3.10 Various logger outputs 45
3.11 JobManager overview 49
3.12 Core managers 50
3.13 Registering new data type 52
3.14 Extracting data from file 54
3.15 File data lazy initialization 56
3.16 Loading plug-ins 59
3.17 Unpacking and registering plug-in content 60
3.18 Text editor and reporting 62
3.19 Communication data source 68
3.20 Kinematic visualizer 69
3.21 Chart visualizer 71
3.22 Timeline playback controller 72
3.23 Subject hierarchy and data grouping 73
3.24 Video visualizer 74
3.25 Python environment 75
3.26 CI process overview 80
3.27 External libraries hierarchy with custom projects dependencies 81

4.1 Example data flow structure for a given graph 85
4.2 Input pin wrapper state chart 97
4.3 Output pin wrapper state chart 98
4.4 Source node wrapper state chart 99
4.5 Processing node state chart 100
4.6 Sink node wrapper state chart 101
4.7 Examples of visual programming environments 105
4.8 Visual data flow environemnt exmaple for MDE 106

5.1 Lifting scheme forward transform 119
5.2 Lifting scheme inverse transform 119
5.3 Lerp and slerp rotations interpolations comparison 128
5.4 Prediction diagram for squad interpolation in tangent space . 129
5.5 Test data - left knee of 26 years old and healthy male 133
5.6 LinHaar details after forward transform 135
5.7 LinHaar signal reconstruction 135
5.8 QuatHaar details after forward transform 136
5.9 QuatHaar signal reconstruction 136
5.10 Lerp details after forward transform 137
5.11 Lerp signal reconstruction 137
5.12 Slerp details after forward transform 138
5.13 Slerp signal reconstruction 138
5.14 Squad details after forward transform 139
5.15 Squad signal reconstruction 139
5.16 TangentSpace details after forward transform 140
5.17 TangentSpace signal reconstruction 140
5.18 Noisy test data 142
5.19 QuatHaar lifting scheme noise reduction best results 144
5.20 Slerp lifting scheme noise reduction best results 145
5.21 Squad lifting scheme noise reduction best results 146
5.22 TangentSpace lifting scheme noise reduction best results . . . 147
5.23 QuatHaar compression quality loss 150
5.24 Slerp compression quality loss 151
5.25 Squad compression quality loss 152
5.26 TangentSpace compression quality loss 153
5.27 Quaternion based motion analysis data processors 157
5.28 Pins data wrappers 158

List of Tables

2.1 Motion data types 8
2.2 Basic motion recording equipment 9
2.3 Complex motion recording systems 11
2.4 Different file formats for storing motion data 13
2.5 Motion analysis tools comparison 15

3.1 MDE features 20
3.2 OW policies 29
3.3 *DataChannel* properties 33
3.4 Application paths 43
3.5 corelib exported classes 64
3.6 coreui exported classes 65

5.1 Noisy signals distances to original signal 141
5.2 Noise reduction results 148
5.3 Compression ratios 148
5.4 Compression results 154

List of Algorithms

3.1 Requesting ThreadPool for threads 47
3.2 Releasing unused thread 47

5.1 Average recursive algorithm 122

List of Acronyms

MDE	Motion Data Editor
PJWSTK	Polsko-Japońska Wyższa Szkoła Technik Komputerowych
HML	Human Motion Laboratory
HMDB	Human Motion Database
IMU	inertial measurement unit
EMG	electromyography
GRF	ground reaction forces
MoCap	motion capture
FTP	File Transfer Protocol
STL	Standard Template Library
SVN	Source Version Control
HD	High Definition
B-tk	Biomechanical ToolKit
RTTI	Real-Time Type Info
POD	Plain Old Data
RTR	Return Type Resolver
CPU	Central processing unit
GPU	Graphics processing unit
CI	Continuous Integration

API	application public interface
GUI	graphical user interface
SDK	software development kit
I/O	input/output
OW	ObjectWrapper
ABI	application binary interface
UI	user interface
IDE	Integrated Development Environment
PIMPL	Private Implementation
RAII	Resources Acquisition Is Initialization
URL	Uniform Resource Locator
CAS	Computer Algebra Systems

Bibliography

[1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. *Theory of Computing Systems*, 35(3):321–347, 2002.

[2] A. Ahmed, A. Hilton, and F. Mokhtarian. Adaptive compression of human animation data. In *Proceedings of EuroGraphics Conference, Saarbrucken, Germany*, 2002.

[3] Joseph Albahari and Ben Albahari. *C# 5.0 in a Nutshell: The Definitive Reference*. O'Reilly Media, 5 edition, 6 2012.

[4] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional, 1 edition, 2 2001.

[5] Okan Arikan. Compression of motion capture databases. In *ACM Transactions on Graphics (TOG)*, volume 25, pages 890–897. ACM, 2006.

[6] Philippe Beaudoin, Pierre Poulin, and Michiel van de Panne. Adapting wavelet compression to human motion capture clips. In *Proceedings of Graphics Interface 2007*, pages 313–318. ACM, 2007.

[7] Thomas Beth, Ingo Boesnach, Martin Haimerl, Jörg Moldenhauer, Klaus Bös, and Veit Wank. Characteristics in human motion–from acquisition to analysis. In *IEEE International Conference on Humanoid Robots*, pages 56–75, 2003.

[8] Jasmin Blanchette and Mark Summerfield. *C++ GUI Programming with Qt 4 (2nd Edition) (Prentice Hall Open Source Software Development Series)*. Prentice Hall, 2 edition, 2 2008.

[9] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[10] Armin Bruderlin and Lance Williams. Motion signal processing. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 97–104. ACM, 1995.

[11] Erik B. Dam, Martin Koch, and Martin Lillholm. Quaternions, interpolation and animation. 1998.

[12] Konstantinos Daniilidis. Hand-eye calibration using dual quaternions. *The International Journal of Robotics Research*, 18(3):286–298, 1999.

[13] Ingrid Daubechies, Igor Guskov, Peter Schröder, and Wim Sweldens. Wavelets on irregular point sets. *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 357(1760):2397–2413, 1999.

[14] James Diebel. Representing attitude: Euler angles, unit quaternions, and rotation vectors. 2006.

[15] James Dinan, Brian Larkins, Ponnuswamy Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 53. ACM, 2009.

[16] Paul M. Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, 1 edition, 7 2007.

[17] John W. Eaton, David Bateman, and Soren Hauberg. *Gnu Octave Version 3.0.1 Manual: A High-Level Interactive Language For Numerical Computations*. CreateSpace Independent Publishing Platform, 1 edition, 3 2009.

[18] YC Fangt, CC Hsieh, MJ Kim, JJ Chang, and TC Woo. Real time motion fairing with unit quaternions. *Computer-Aided Design*, 30(3):191–198, 1998.

[19] Wiktor Filipowicz, Piotr Habela, Krzysztof Kaczmarski, and Marek Kulbacki. A generic approach to design and querying of multi-purpose human motion database. In *ICCVG (1)*, pages 105–113, 2010.

[20] Johan Fredriksson and Carl Olsson. Simultaneous multiple rotation averaging using lagrangian duality. In *Computer Vision–ACCV 2012*, pages 245–258. Springer, 2013.

[21] Amos Gilat. *MATLAB: An Introduction with Applications*. Wiley, 4 edition, 12 2010.

[22] Klaus Gürlebeck, Wolfgang Sprössig, and Klaus Guerlebeck. *Quaternionic and Clifford Calculus for Physicists and Engineers*. Wiley, 2 edition, 6 1998.

[23] Yi Guo, Jisheng Zhao, Vincent Cave, and Vivek Sarkar. Slaw: A scalable locality-aware adaptive work-stealing scheduler. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[24] Andrew J. Hanson. *Visualizing Quaternions (The Morgan Kaufmann Series in Interactive 3D Technology)*. Morgan Kaufmann, 1 edition, 1 2006.

[25] Richard Hartley, Khurrum Aftab, and Jochen Trumpf. L1 rotation averaging using the weiszfeld algorithm. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pages 3041–3048. IEEE, 2011.

[26] Chung-Chi Hsieh. B-spline wavelet-based motion smoothing. *Computers & industrial engineering*, 41(1):59–76, 2001.

[27] Chung-Chi Hsieh. Motion smoothing using wavelets. *Journal of Intelligent and Robotic Systems*, 35(2):157–169, 2002.

[28] L. Huang. *A Concise Introduction to Mechanics of Rigid Bodies: Multidisciplinary Engineering*. Springer, 2012 edition, 11 2011.

[29] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Addison-Wesley Signature Series (Fowler))*. Addison-Wesley Professional, 1 edition, 8 2010.

[30] O'Reilly Media Inc. Big data now: 2012 edition, 10 2012.

[31] Maarten H. Jansen, Patrick Oonincx, and Patrick J. Oonincx. *Second generation wavelets and applications*. Springer, 2005.

[32] Michael Patrick Johnson. *Exploiting quaternions to support expressive interactive character motion*. PhD thesis, Massachusetts Institute of Technology, 2002.

[33] Nicolai M. Josuttis. *The C++ Standard Library: A Tutorial and Reference (2nd Edition)*. Addison-Wesley Professional, 2 edition, 4 2012.

[34] Bert Jüttler. Visualization of moving objects using dual quaternion curves. *Computers & Graphics*, 18(3):315–326, 1994.

[35] Ladislav Kavan, Steven Collins, Jiří Žára, and Carol O'Sullivan. Geometric skinning with approximate dual quaternion blending. *ACM Transactions on Graphics (TOG)*, 27(4):105, 2008.

[36] Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*. Wiley, 3 edition, 7 2013.

[37] JeHee Lee and Sung Yong Shin. Motion fairing. In *Computer Animation'96. Proceedings*, pages 136–143. IEEE, 1996.

[38] Jehee Lee and Sung Yong Shin. A coordinate-invariant approach to multiresolution motion analysis. *Graphical Models*, 63(2):87–105, 2001.

[39] Jehee Lee and Sung Yong Shin. General construction of time-domain filters for orientation data. *Visualization and Computer Graphics, IEEE Transactions on*, 8(2):119–128, 2002.

[40] Shiyu Li, Masahiro Okuda, and Shin ichi Takahashi. Compression of human motion animation using the reduction of interjoint correlation. *Journal on Image and Video Processing*, 2008:2, 2008.

[41] Yi Lin and Michael D. McCool. Nonuniform segment-based compression of motion capture data. In *Advances in Visual Computing*, pages 56–65. Springer, 2007.

[42] Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo. *C++ Primer (5th Edition)*. Addison-Wesley Professional, 5 edition, 8 2012.

[43] Landis F. Markley, Yang Cheng, John Lucas Crassidis, and Yaakov Oshman. Averaging quaternions. *Journal of Guidance, Control, and Dynamics*, 30(4):1193–1197, 2007.

[44] Douglas B. Meade, S.J. Michael May, C-K. Cheung, and G.E. Keough. *Getting Started with Maple*. Wiley, 3rd edition, 3 2009.

[45] Maher Moakher. Means and averaging in the group of rotations. *SIAM journal on matrix analysis and applications*, 24(1):1–16, 2002.

[46] Donald B. Percival and Andrew T. Walden. *Wavelet Methods for Time Series Analysis (Cambridge Series in Statistical and Probabilistic Mathematics)*. Cambridge University Press, 1 edition, 2 2006.

[47] Alba Perez and Michael McCarthy. Dual quaternion synthesis of constrained robotic systems. *Journal of Mechanical Design*, 126:425, 2004.

[48] David C. Preston and Barbara E. Shapiro. *Electromyography and Neuromuscular Disorders: Clinical-Electrophysiologic Correlations (Expert Consult - Online and Print), 3e*. Saunders, 3 edition, 12 2012.

[49] Martin Reddy. *API Design for C++*. Morgan Kaufmann, 1 edition, 2 2011.

[50] Herbert Schildt. *Java, A Beginner's Guide, 5th Edition*. McGraw-Hill Osborne Media, 5 edition, 8 2011.

[51] Inna Sharf, Alon Wolf, and MB Rubin. Arithmetic and geometric solutions for average rigid-body rotation. *Mechanism and Machine Theory*, 45(9):1239–1251, 2010.

[52] Ken Shoemake. Animating rotation with quaternion curves. *ACM SIGGRAPH computer graphics*, 19(3):245–254, 1985.

[53] Peter Siao, Didier Cros, and Steve Vucic. *Practical Approach to Clinical Electromyography*. Demos Medical Pub, 1 edition, 1 2011.

[54] Jason McC. Smith. *Elemental Design Patterns*. Addison-Wesley Professional, 1 edition, 4 2012.

[55] Magdalena Stawarz, Andrzej Polański, Stanisław Kwiek, Magdalena Boczarska-Jedynak, Łukasz Janik, Andrzej Przybyszewski, and Konrad Wojciechowski. A system for analysis of tremor in patients with parkinson's disease based on motion capture technique. In Leonard Bolc, Ryszard Tadeusiewicz, Leszek J. Chmielewski, and Konrad W. Wojciechowski, editors, *ICCVG*, volume 7594 of *Lecture Notes in Computer Science*, pages 618–625. Springer, 2012.

[56] Elias M. Stein and Rami Shakarchi. *Fourier Analysis: An Introduction (Princeton Lectures in Analysis)*. Princeton University Press, 3 2003.

[57] Eric J. Stollnitz and Tony D. De Rose. *Wavelets for computer graphics: theory and applications*. Morgan Kaufmann, 1996.

[58] A. Sudbery. Quaternionic analysis. In *Math. Proc. Camb. Phil. Soc*, volume 85, pages 199–225. Cambridge Univ Press, 1979.

[59] Mark Summerfield. *Advanced Qt Programming: Creating Great Software with C++ and Qt 4 (Prentice Hall Open Source Software Development)*. Prentice Hall, 1 edition, 7 2010.

[60] Herb Sutter. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley Professional, 11 1999.

[61] Herb Sutter and Andrei Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Addison-Wesley Professional, 1 edition, 11 2004.

[62] Wim Sweldens. Lifting scheme: a new philosophy in biorthogonal wavelet constructions. In *SPIE's 1995 International Symposium on Optical Science, Engineering, and Instrumentation*, pages 68–79. International Society for Optics and Photonics, 1995.

[63] Wim Sweldens. The lifting scheme: A construction of second generation wavelets. *SIAM Journal on Mathematical Analysis*, 29(2):511–546, 1998.

[64] Agnieszka Szczęsna. The multiresolution analysis of triangle surface meshes with lifting scheme. In *Computer Vision/Computer Graphics Collaboration Techniques*, pages 274–282. Springer, 2007.

[65] Agnieszka Szczęsna. *Multiresolution processing of irregular surface meshes using second generation wavelets*. PhD thesis, Silesian University of Technology, 2007.

[66] Agnieszka Szczęsna, Janusz Słupik, and Mateusz Janiak. Motion data denoising based on the quaternion lifting scheme multiresolution transform. *Machine graphics & vision*, 20(3):238–249, 2011.

[67] Agnieszka Szczęsna, Janusz Słupik, and Mateusz Janiak. Quaternion lifting scheme for multi-resolution wavelet-based motion analysis. In *ICONS 2012, The Seventh International Conference on Systems*, pages 223–228, 2012.

[68] Agnieszka Szczęsna, Janusz Słupik, and Mateusz Janiak. The smooth quaternion lifting scheme transform for multi-resolution motion analysis. In *Proceedings of the 2012 international conference on Computer Vision and Graphics*, ICCVG'12, pages 657–668, Berlin, Heidelberg, 2012. Springer-Verlag.

[69] O'Reilly Radar Team. Big data now: Current perspectives from o'reilly radar, 8 2011.

[70] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley Professional, 1 edition, 11 2002.

[71] John Vince. *Quaternions for Computer Graphics*. Springer, 2011 edition, 6 2011.

[72] John Vince. *Rotation Transforms for Computer Graphics*. Springer, 2011 edition, 1 2011.

[73] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, 3 edition, 6 2013.

[74] Paul Wellin. *Programming with Mathematica®: An Introduction*. Cambridge University Press, 4th revised edition edition, 2 2013.

[75] Anthony Williams. *C++ Concurrency in Action: Practical Multithreading*. Manning Publications, 1 edition, 2 2012.

[76] Adam Świtoński, Andrzej Polański, and Konrad Wojciechowski. Human identification based on gait paths. In *Advances Concepts for Intelligent Vision Systems*, pages 531–542. Springer, 2011.