

Integration of Web-Based Information Visualizations into a Scientific Visualization Environment

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Medizinische Informatik

eingereicht von

Johannes Bauer

Matrikelnummer 0427512

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Mitwirkung: Dipl.-Math. Dr.techn. Katja Bühler, VRVis

Wien, 3.12.2014

(Unterschrift Verfasserin)

(Unterschrift Betreuung)

Integration of Web-Based Information Visualizations into a Scientific Visualization Environment

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Medical Informatics

by

Johannes Bauer

Registration Number 0427512

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Assistance: Dipl.-Math. Dr.techn. Katja Bühler, VRVis

Vienna, 3.12.2014

(Signature of Author)

(Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Johannes Bauer
Zeleborgasse 14-16/6, 1120 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)

(Unterschrift Verfasserin)

Abstract

Today's neuro-biological research is often based on brains of the *Drosophila Melanogaster*, the commonly known fruit fly. To study the function of neuronal circuits scientists often have to compare the neuronal structures of a set of different brains. Their aim is to find out how complex behavior is generated. Therefore the scientists at the Institute of Molecular Pathology (IMP) in Vienna are using a confocal microscope to produce volumetric images of *Drosophila* brains. Today they have acquired more than 40.000 images and a large amount of additional data.

In many cases 3D renderings of these volumetric images are not sufficient to solve certain scientific problems especially when multiple brains have to be considered. Therefore the researchers rely on additional data which is stored in databases. The problem here is that the scientists have two different data sources without a connection between them. On the one hand there are the volumetric images and on the other hand there is the additional data which has certain relations to the brains.

This thesis proposes a software design concept to establish a connection between 3D renderings of volumetric images and additional data by using information visualizations. Highlighting techniques can be introduced to link volume visualizations of the brains to related data visualized by 2D information visualizations. Therefore the implementation of this design concept gets integrated into an existing scientific visualization environment. To evaluate this concept common neuro-biological use cases are introduced and it is described how the implementation of this design concept supports the work flow of the researchers.

Kurzfassung

Die heutige neurobiologische Forschung basiert oft auf Gehirnen der *Drosophila Melanogaster* Fruchtfliege. Um die Funktion von neuronalen Netzwerken der Gehirne zu erforschen müssen die Wissenschaftler oftmals neurale Strukturen von mehreren Gehirnen miteinander vergleichen. Ihr Ziel ist es heraus zu finden wie komplexes Verhalten entsteht. Zu diesem Zweck verwenden die Wissenschaftler des Forschungsinstituts für molekulare Pathologie (IMP) in Wien ein konfokales Mikroskop um volumetrische Bilder der Gehirne zu erzeugen. Heute haben sie bereits mehr als 40.000 dieser Bilder gemeinsam mit einer großen Menge an zusätzlichen Daten erzeugt.

In vielen Fällen sind drei dimensionale Darstellungen nicht ausreichend um gewisse wissenschaftliche Probleme zu lösen. Gerade dann, wenn mehrere Gehirne in Betracht gezogen werden müssen. Deswegen benötigen die Forscher zusätzliche Daten welche in Datenbanken abgespeichert sind. Hier ergibt sich das Problem dass die Forscher zwei unterschiedliche Datenquellen ohne direkte Verbindung dazwischen in Betracht ziehen müssen. Auf der einen Seite gibt es die drei dimensional Bilder und auf der anderen Seite zusätzliche Daten die aber gewisse Beziehungen zu den Bildern haben.

Diese Arbeit stellt ein Software Design Konzept vor um eine Verbindung zwischen den drei dimensional Darstellungen der volumetrischen Bilder her zu stellen. Highlighting Techniken können verwendet werden um Verbindungen zwischen Volumen Visualisierungen der Gehirne und den dazu in Beziehung stehenden Daten in den zwei dimensional Informationsvisualisierungen her zu stellen. Dafür wird die Implementierung dieses Software Design Konzepts in eine existierende wissenschaftliche Visualisierungsumgebung integriert. Um dieses Konzept zu evaluieren werden häufige neurobiologische Anwendungsfälle herangezogen und beschrieben, wie die Implementierung dieses Design Konzepts den Arbeitsverlauf der Forscher unterstützen kann.

Contents

1	Introduction	1
1.1	Problem Statement	2
1.2	Approach	2
1.3	Requirements	3
1.4	Thesis Overview	3
2	Background	5
2.1	Visualization Pipeline	5
2.2	Types of Data Visualizations	9
2.3	Multiple Coordinated Views	10
2.4	Model View Controller Pattern	12
3	Related Work	15
3.1	Data Visualization	15
3.2	Applications of the Visualization Pipeline	16
3.3	Conclusion	20
4	Software Design	23
4.1	Overview	23
4.2	Existing Client-Server Environment	23
4.3	Existing User Interface	25
4.4	Architectural Design Overview	28
4.5	Server features	29
4.6	Client Features	32
5	Implementation	37
5.1	Existing Environment	37
5.2	Server Features	37
5.3	Client Features	42
6	Evaluation	55
6.1	Objective Evaluation	55
6.2	Case studies	58

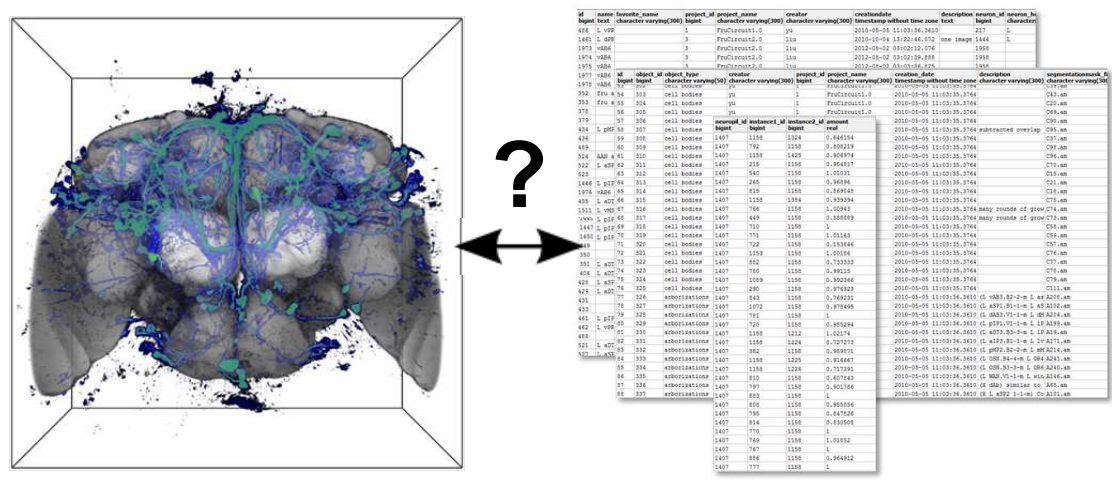
7 Conclusion and Future Work	69
7.1 Conclusion	69
7.2 Future work	69
Bibliography	75

Introduction

Data generated by scientific experiments can be divided into two categories. In the last decade two scientific communities originated out of this division each focusing on the visualization of data of one of these categories [28].

On the one side there is the community about scientific visualization (SciVis). Here data sets are typically generated from measurements, simulations or scans of a three-dimensional object. It deals with data that represents objects from the physical world - data, which has a spatial context.

On the other hand there is the information visualization community (InfoVis). Its different research area covers visual representations of abstract data that primarily has no spatial dimensions but might have relations to a physical object. Representations aim at illustrating multiple dimensions to enable humans to identify patterns, clusters, trends or outliers in the data [31].



Using InfoVis methods to display properties of physical objects might be useful to establish a link between these two research areas. Thus it might be possible to compare multiple physical objects by their numerical properties (figure 1.1).

1.1 Problem Statement

The scientists at the Institute for Molecular Pathology (IMP) are using a confocal microscope to produce volumetric images of brains of the *drosophila melanogaster* (the commonly known fruit fly). After taking these images the scientists conduct several processing steps to collect additional information about the neuronal structures and properties of each sample. They point out cell bodies, neurons or arborizations by segmenting these structures within the raw data. These results of the segmentation are essential for further analysis.

Today the scientists have collected more than 40.000 images [3] and a huge amount of related data. Consequently the scientists often need to compare neuronal structures or query certain images due to given properties. Such problems cannot be solved by simply rendering these objects. This requires new methods to connect the three-dimensional representations to the related data in a clearly structured way.

What is needed is a solution to link three-dimensional images with the related numerical and categorical data.

1.2 Approach

In the mid 90's *information visualization (InfoVis)* became a well known and established research field in computer science [38]. Information visualization uses graphical models to visually represent abstract data. Such proven concepts can be used to visualize the related data of three-dimensional images. This should facilitate the process of comparing or finding similarities within a set of objects. Furthermore it might allow the scientists to gain better insight into their data.

Due to the long existence of the research field of information visualization many definitions can be found in the literature. One says, 'information visualization uses graphical models that may represent abstract concepts and relationships that do not necessarily have a counterpart in the physical world' [22]. In the context of the existing environment the real world objects are 3 dimensional images with related numerical data. Hence, in the biomedical context of this concept the connection between the two sides plays an essential role.

This design concept will be integrated as a new feature into *BrainGazer* [20], an existing visualization environment for visualizing three-dimensional image data of *drosophila* brains in order to study neuronal circuits. *BrainGazer* offers a multi-view user interface where the user can place a set of different views on the screen. So far each view has a predefined functionality, e.g., 3D rendering or database query features. The proposed concept shall extend the existing environment by additional views with two-dimensional InfoVis views and establish links to the 3D renderings of images.

Therefore a framework has to be realized in order to enable this integration. This framework can then be used by software developers to add new views after domain experts have defined use cases.

1.3 Requirements

Due to the implementation in an existing environment the following requirements can be inferred.

- To establish a seamless integration new views are required to support existing user interface features. Therefore it is necessary to develop interfaces between these two entities.
- Take the client server infrastructure into account when implementing visualizations. This allows extending the environment without any modification of the client application.
- Create exchangeable modules for both data sources and visualizations. On the client side, this allows the domain experts to decide which type of visualization is suitable for the given problem.
- Establish a modular developing framework to minimize the implementation efforts for new visualizations.

1.4 Thesis Overview

The next section describes conceptual models and methodologies that are used in this thesis. Chapter 3 introduces related projects that have a certain similarity to this concept. The method chapter will describe, how the techniques and practices can be used and combined in order to establish a solution. The implementation chapter gives a detailed overview over programming and developing techniques. The evaluation in chapter 6 first evaluates the performance of the implementation. The second part introduces the biological background and describes how the proposed framework can be used to solve given use cases. The thesis ends with a conclusion and possible improvements of this concept.

Background

This chapter introduces theoretical backgrounds that form the basis of the presented thesis. Selected conceptual models and methodologies are introduced that are required to understand the methodology of this concept.

2.1 Visualization Pipeline

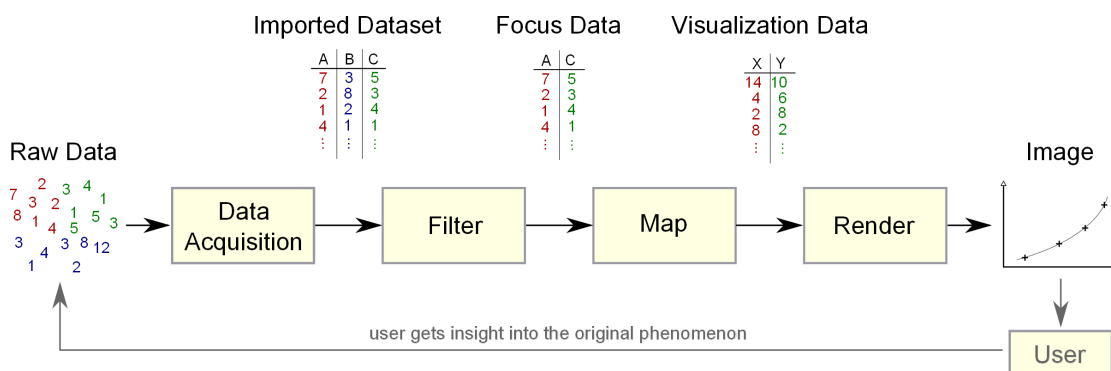


Figure 2.1: The visualization pipeline describes how to transform raw data into a displayable image. Thus, the user gets insight into the original data and, depending on the application, may be able to change the raw data which restarts the process [26].

The *visualization pipeline* (see Figure 2.1) embodies a data flow model describing how data is moved within a collection of computation steps (modules) by a directed graph [35]. Modules can be categorized into three types:

Source A source module produces data by providing it as an output. An example would be a file reader or the result of a database query.

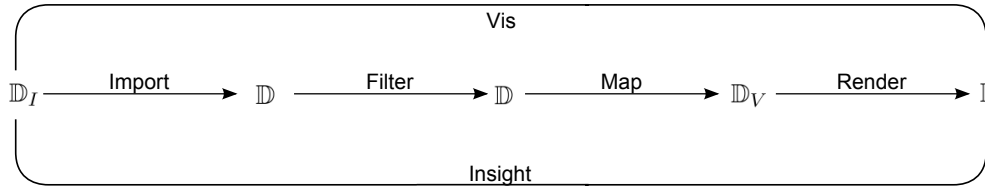


Figure 2.2: The visualization pipeline seen as a composition of formal functions (image source: Telea [44])

Sink A sink module receives data as input. This can be a file writer or the user’s screen which displays rendered images.

Filter A filter module is a combination of a source and a sink. It takes data at least from one input, performs operations and passes the results to the output.

The concept describes in four steps how to generate an image from raw data. The components are connected sequentially representing how data flows between the components. This can be described formally by functions [44].

Formal Description

The visualization pipeline can be seen as a function Vis that maps between \mathbb{D}_I and \mathbb{I} (see Figure 2.2) whereas \mathbb{D}_I is the set of all possible types of raw data and \mathbb{I} is the set of produced images (Function 2.1).

$$Vis : \mathbb{D}_I \longrightarrow \mathbb{I} \quad (2.1)$$

After examining the result image the user should be able to get insight into the input data. Hence, it should allow the user to answer questions about the problem. Therefore a reverse function $Insight$ 2.2 in the opposite direction can be defined which maps the image back to raw data.

$$Insight : \mathbb{I} \longrightarrow \mathbb{D}_I \quad (2.2)$$

To actually create the image from the raw data it is necessary to import the data first. This means, that a representation of the original object has to be found. This representation is then stored in a dataset. Conceptual this means mapping the raw information \mathbb{D}_I to a dataset $\mathcal{D} \in \mathbb{D}$. Here, \mathbb{D} is the set of all supported datasets. Importing data can be modeled by the function 2.3:

$$Import : \mathbb{D}_I \longrightarrow \mathbb{D} \quad (2.3)$$

It describes the process of converting the original information to the data structure that was selected to represent the original phenomenon. In the best case this is a one-to-one mapping or copying which usually involves reading data from an external storage or a measuring device.

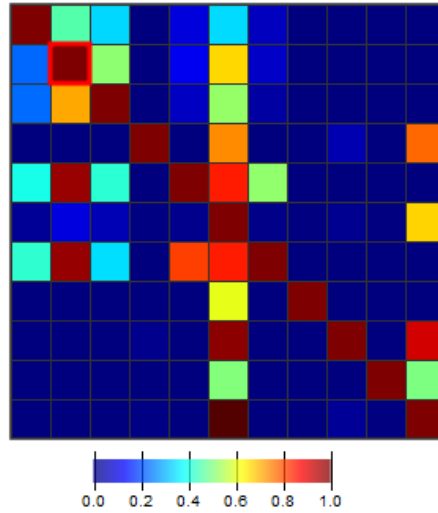


Figure 2.3: For a heatmap matrix numeric input values have to be converted to a certain range. This focus dataset is mapped to a suitable color scale. (image source: own work)

Depending on the application it might be necessary to select specific features in the imported dataset or to eliminate non relevant features.

$$\text{Filter} : \mathbb{D} \longrightarrow \mathbb{D} \quad (2.4)$$

The filtering process is a strict data manipulation operation. Thus, input and output of the function (see Function 2.4) are both datasets but usually it is not necessary to visualize the entire dataset. Therefore properties of the dataset might have to be limited or the size has to be reduced. Formally, this is a selection of a subsets of interest which can be done in the spatial domain, the attribute-value domain or in a combination of both (function 2.4).

The result of the filtering operation is called the *focus dataset* which should represent all features of interest. The next step is to map this features into a visual domain (Function 2.5).

$$\text{Map} : \mathbb{D} \longrightarrow \mathbb{D}_V \quad (2.5)$$

This function maps the focus dataset to a visual domain like distances, colors, sizes or positions. The elements of this visual domain are what the user gets presented on the display.

The components

Due to Moreland [35] the visualization pipeline compromises three primary components.

- Modules
- Connections
- Execution

Modules

Modules are the functional units of the pipeline. They are responsible for:

Import (Data Acquisition) Importing is the process of collecting data from measurements or simulation. This step collects raw data and stores it in specifically designed data structures. The type of data structure strongly depends on the application. For smaller applications a file might be sufficient but in larger environments relational data bases are essential to store the imported dataset.

Filter Filtering is necessary because the imported data set may contain more information or features than the user needs. Therefore methods are needed to extract subsets which are related to the problem. Here decisions are made, which subset of the stored data is relevant for the visualization. This subset is called the focus data.

Map The mapping step transforms the focus data set to a visual domain. This means that data values are mapped to visual features. This set of features defines the type of plot.

Render The rendering process generates the final visualization and provides the image to the user interface which allows the user to get insight into the original phenomenon.

Connections

Connections between the modules are responsible for the data flow between two modules. They allow the modules to transmit their output to the next input port. The distance of this transmission may depend on the topology of the application. If two modules are part of a single software component the transmission can be done via the systems memory but if they are distributed over multiple machines may require different and more complex transmission techniques.

Execution

The execution can be defined as the process which initiates the computation step of each module.

For certain scenarios a feedback loop is necessary for the pipeline. The user primarily gets insight into the original phenomenon but, due to a back channel, is able to change the raw data of the pipeline. This allows engaging with the measurement or simulation process often in order to get immediate feedback about the change.

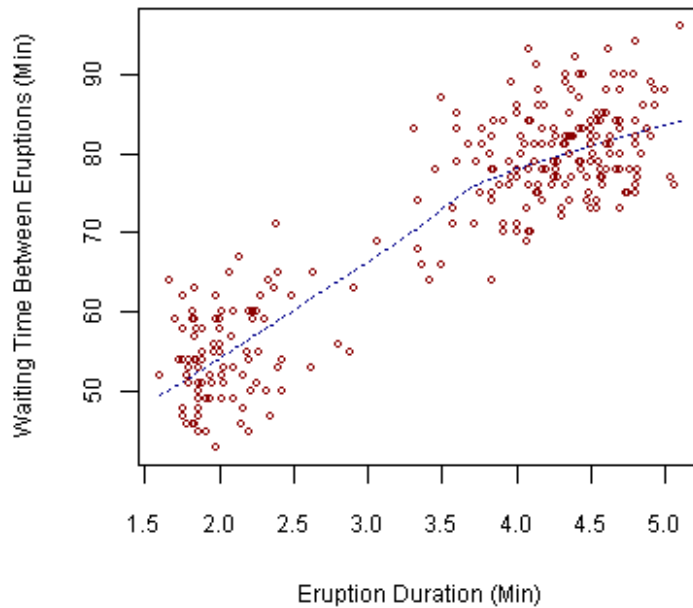


Figure 2.4: A scatter plot is a simple example for information visualization. This plot displays the duration and the waiting time between eruptions of the "Old Faithful Geyser" in Yellowstone National Park [10].

2.2 Types of Data Visualizations

Information Visualization

The research area of *information visualization (InfoVis)* uses graphical methods to produce visual representations of *abstract data* to reinforce human cognition [41]. Abstract data might be the result of measurements, simulations or various operations that generate output - datasets that do not necessarily have a physical representation.

The aim of information visualization is to assist users in understanding the structure of data by "forming a mental model or image of something" [43]. Chi [21] defined information visualization as "visualizations applied to abstract quantities and relations in order to get insight in the data". Therefore a proper visualization might help to recognize clusters, patterns or relationships and enable the user to make decisions or infer knowledge. Typical examples for information visualizations are:

- Scatter plot: A scatter plot uses a 2 dimensional cartesian coordinates to display pairs of values (see Figure 2.4).
- Heat map matrices: A heat map can be used to visualize a $M \times N$ matrix of values (see Figure 2.3).

- Parallel coordinates: They can be used to display a multivariate dataset (see Figure 6.5).

Datasets for information visualizations usually do not contain spatial dimensions but in some cases they have a semantic relationship to a physical object or a location.

Scientific visualization

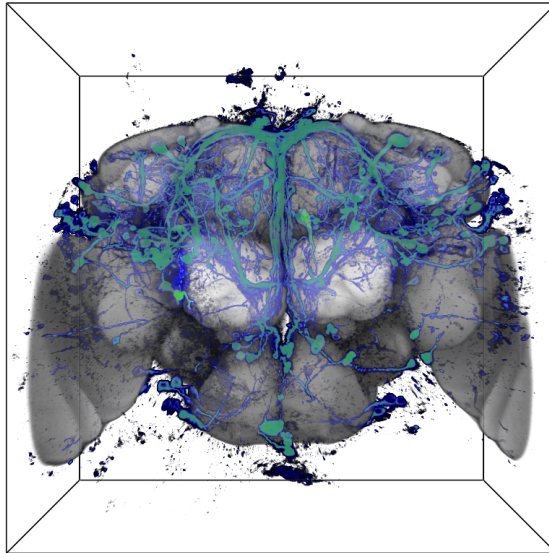


Figure 2.5: A volume rendering of the neuronal structures of a fruit fly is a typical example for a scientific visualization (source: rendered with BrainGazer [20]).

The focus of *scientific visualization (SciVis)* is the presentation of data that has a direct relation to a real world object or phenomenon. It is "primarily concerned with the visualization of three-dimensional phenomena, where the emphasis is on realistic renderings of volumes, surfaces, illumination sources" [24]. Hence, scientific visualizations often require a special 3D rendering environment with different interaction methods. The purpose is to allow scientists to gain insight into phenomena or objects. Samples are often captured by scanners, microscopes, sensors or they might be the result of simulations.

Scientific visualization is like information visualization a mature research field. A typical application is volume visualization where the focus lies on rendering of three-dimensional images taken from objects (see Figure 2.5).

2.3 Multiple Coordinated Views

Multiple coordinated views are a user interface technique to display a single conceptual entity by two or more distinct views. Figure 2.6 shows a screenshot of an application which has a

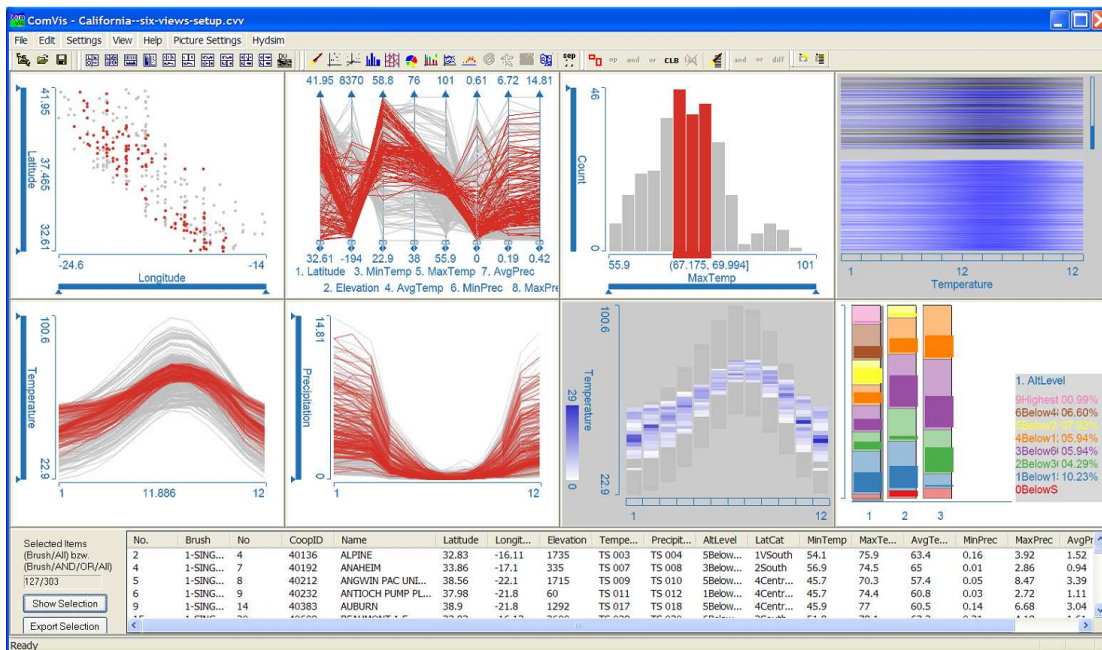


Figure 2.6: A screenshot of ComVis [34], an information visualization application which offers a customizable user interface.

multi-view user interface. In this context distinct views allow the user to learn about different aspects of the data [45]. For example the user interface offers an overview view and more detailed views. The main view is used for the whole dataset while the detail views display smaller parts with greater accuracy [39].

Today’s scientific applications often deal with multi-dimensional entities. The use of only a single visualization might not be sufficient to answer a certain question efficiently. Therefore multi-view user interfaces were introduced to enable different perspectives on the dataset. Certain interaction techniques allow coordination between views.

User interface designer have to decide how the user may interact and manage a set of different views. Too many or wrong arranged views might have in a negative consequences for the users perception [39]. An indicator for the need of multiple views is if different views bring out correlations and/or disparities (rule of complementarity) [45]. This supports the investigation of multiple entities because the user does not need to remember components he wants to compare.

Coordination

As the name implies multiple views require coordination techniques among the views. Therefore coupling functions are required which specify a mappings on how changes from one view are propagated to the other views (propagation model). Very common interaction techniques are

brushing and linking [45].

Brushing

Brushing is a highlighting technique in multiple coordinated view environments. If the user selects an element in one view, the same (or related) element gets simultaneously highlighted in all the other views by the coordination system.

Linking

Linking, or *navigational slaving*, stands for the propagation of navigational actions from one view to linked views [39]. This can be used to enable synchronized scrolling in distinct views or to propagate filter settings as described in Figure 2.7.

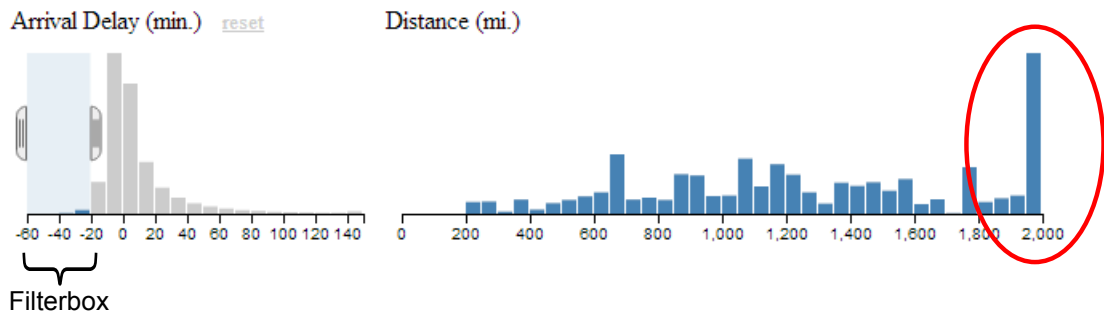


Figure 2.7: Airline on-time performance: In this example two linked views are used to display the arrival delay and the distance of flights (image source: adaptation from Crossfilter [4]). The plots allow users to set filters in the left plot to limit the arrival time. The coordination system applies the same filter to the right plot. As a result it can be seen that longer flights are more likely to arrive earlier as scheduled.

2.4 Model View Controller Pattern

The model-view-controller principle is a software engineering design pattern. The purpose of this concept is to keep a clean separation between user interactions, the data model and view on the data. [32]. This pattern is based on the functionality of the *observer pattern*. Basically it is being used to allow trigger procedures if the state of an observed data structure changes.

As the name suggests it consists of three separated components (see Figure 2.8).

- The **model** is an object. It represents the data that has to be presented in the view. It is independent from controller and view.
- The **view** is responsible for obtaining the data from the model and presenting it to the user. It is aware of the model and the controller

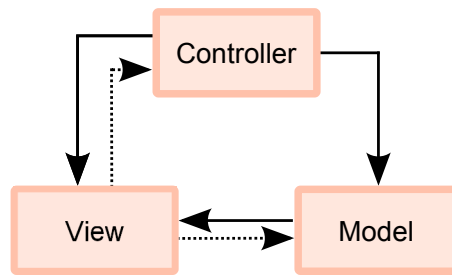


Figure 2.8: The three components of the model-view-controller design pattern. The solid lines represent direct associations. Dotted lines illustrate a relationship via an observer pattern.

- The **controller** receives user interactions to manipulate the model.

If the user changes the model via a view, the controller manipulates the data. Due to an observer pattern, the view gets notified about the change and is able to update the view. A passive approach would be if the controller and view keep checking the model for changes (polling).

Both the model-view-controller and the observer pattern are well known and commonly used design patterns in software engineering. Today they are an integral component of many programming languages including web technologies [32].

Related Work

This chapter introduces related work. It starts with literature about fundamental basics and techniques of this thesis. The second part focuses on projects that deal with the visualization pipeline. Each of these projects has a certain similarity to either the existing architecture or the methodology of this implementation. The last section introduces other scientific software environments that include information visualizations.

3.1 Data Visualization

Visualization in general means to "form a mental vision, image, or picture of (something not visible or present to sight, or of an abstraction); to make visible to the mind or imagination" (original source: Oxford English Dictionary, 1989). Due to accomplishment in computer graphics data visualizations evolved to a mature research area in the last 20 years [18, 28].

In this time many definitions of *information visualization* were created. Gee et al. [25] stated that "information visualizations attempt to efficiently map data variables onto visual dimensions in order to create graphic representations", a statement which relates graphic representations with data. As defined by many other researchers, the area of information visualization concentrates on the visual representation of abstract data [29, 38].

Due to the "Guidelines for Using Multiple Views in Information Visualization" by Wang and Baldonado [45] a view is defined as the combination of a set of data together with specifications on how to display the data. These specifications define a technique like a scatter plot or a heatmap to represent the data in an appropriate way. A single view can represent the data by the use of one particular type of visualization. Multiple views obviously allow users to use different types simultaneously while coordination techniques can be introduced to support the investigation of conceptual entities (see Figure 3.1).

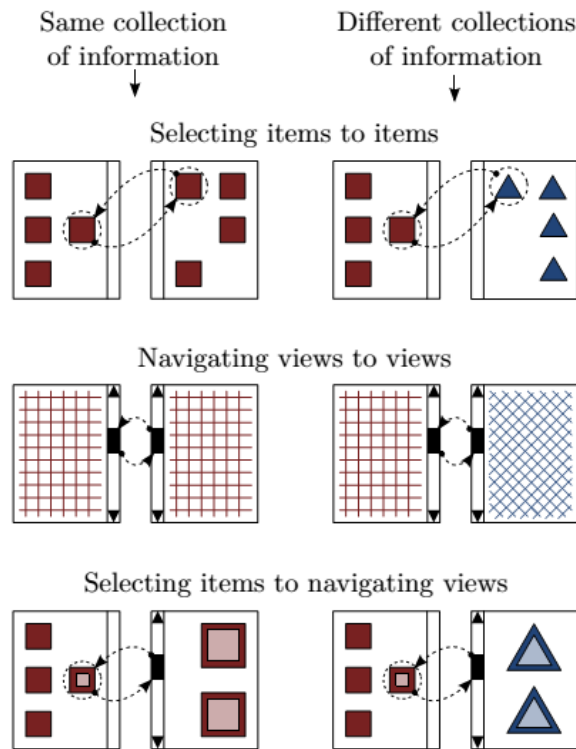


Figure 3.1: The taxonomy of coordination mechanisms between different views [36] (image source: Multiple and Coordinated Views in Information Visualization [39])

3.2 Applications of the Visualization Pipeline

”The conceptual model of the visualization pipeline” introduced by Haber and McNabb describes the process of how to create a visual representation of data in three major transformations. ”The goal of these transformations on the data, is to convert the information to a format amenable to understanding by the human perceptual system while maintaining the integrity of the information.” [26] The reference projects in this section use the visualization pipeline as developing basis.

Schroeder et al. introduce the Visualization Toolkit [40], a library for the development of 3D graphic and visualization applications. As the name proposes it can be used to build complex applications from small components. Therefore it is essential to keep this pieces well defined and establish simple interfaces between them.

A modular design enables the developer to introduce different programming languages for components. Programming languages can be categorized into two groups. There are compiled languages offering performance advantages at runtime and interpreted languages which offer a greater flexibility to developers. As a result the core components of the toolkit were implemented in a compiled language to gain performance. Interpreter languages were used for high level

components.

Another reason to introduce different programming languages is to maintain portability. The use of different technologies enables components to run on multiple platforms and emphasizes the use of consistent interfaces to allow a standardized data exchange.

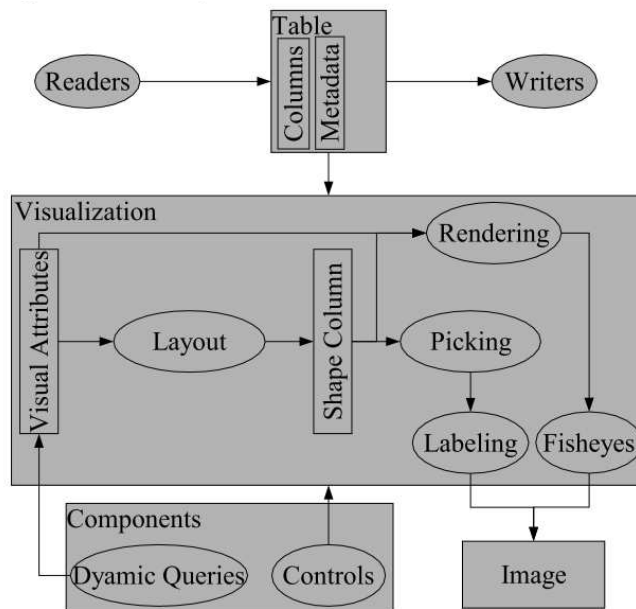


Figure 3.2: The InfoVis Toolkit offers an internal structure consisting of data structures (squares) and functions (ellipses) (image source: The InfoVis Toolkit [23])

Fekete et al. introduced the InfoVis Toolkit [23]. Here, the same approach was chosen by introducing a framework of components but with a focus on two dimensional data visualizations. The proposed toolkit offers a large set of predefined visualizations and control elements to simplify the implementation of visualizations. The aim is to accelerate the development process in order to "achieve a fast action/feedback loop required by dynamic queries". As can be seen in Figure 3.2 it offers components for different stages of the visualization pipeline starting from the data acquisition readers to rendering controls.

Prefuse [29] takes the next step by introducing a user interface for crafting interactive visualizations. The focus is to simplify the developing process in order to allow users to create novel visualizations without having knowledge about programming languages or mathematical algorithms. It introduces finer-grained building blocks than the InfoVis Toolkit [23] (see Figure 3.3) to allow users to create visualizations that are specifically tailored for their use case.

IRIS Explorer [27, pages 633-654] breaks down the pipeline into its components. It offers different modules for each part of the pipeline and lets the user construct the data flow according

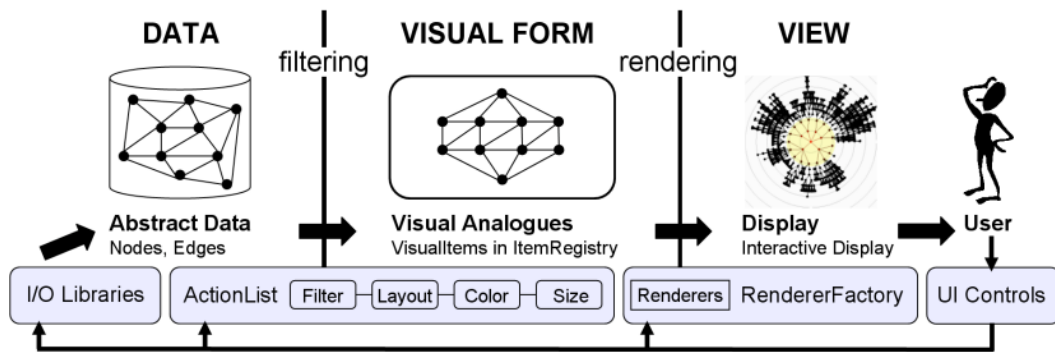


Figure 3.3: The visualization pipeline in Prefuse. It illustrates the different states of data within the pipeline where the developer can modify the process through parameters. (image source: Prefuse [29])

to the given problem. Thus the first module is responsible for reading data from a file, a database or some other application that runs simultaneously. The next step is to select an appropriate filter module followed by the transformation step that transforms the data into displayable geometric objects. The last step for the user is to set up an output module which either renders the image on the user's screen or writes the data into a file. Each module can be adjusted by a set of parameters. For example the user has to set the name of the input file or has to select the number of used contours. For the data exchange between the modules the user has to define data types by using a specifically designed data typing language. Due to its modular design IRIS Explorer can be extended by user defined modules. Application interfaces are provided that even support the distribution of module instances to execute the pipeline in a networked environment.

VisTrails [17] extends the concept of visualization pipelines by proposing *visualization trails* (see Figure 3.4). It allows a much more detailed specification of data flows than the previously introduced pipeline by Haber and McNabb [26]. The user interface supports the development of a trail by connecting modules in the development stage. Depending on the type of module each offers input and output ports. This variability allows querying data from different sources, define data flows to enable preprocessing steps and generate one or more output visualizations. Due to the flexibility of this modular system this principle can be used to map scientific workflows.

Zsu et al. [47] discusses how the visualization pipeline can be mapped to network nodes in order to archive an efficient execution (see Figure 3.5). In this approach the nodes are connected via a wide area network. Due to the geographical dimensions of such a network the transfer times between the nodes play an essential role. To optimize the total delay of the pipeline the problem is formulated analytically and takes computation times of modules and the transfer delays between them into account. The proposed algorithms use these latencies to enable an optimal mapping of the visualization pipeline to the nodes for high performance visualizations

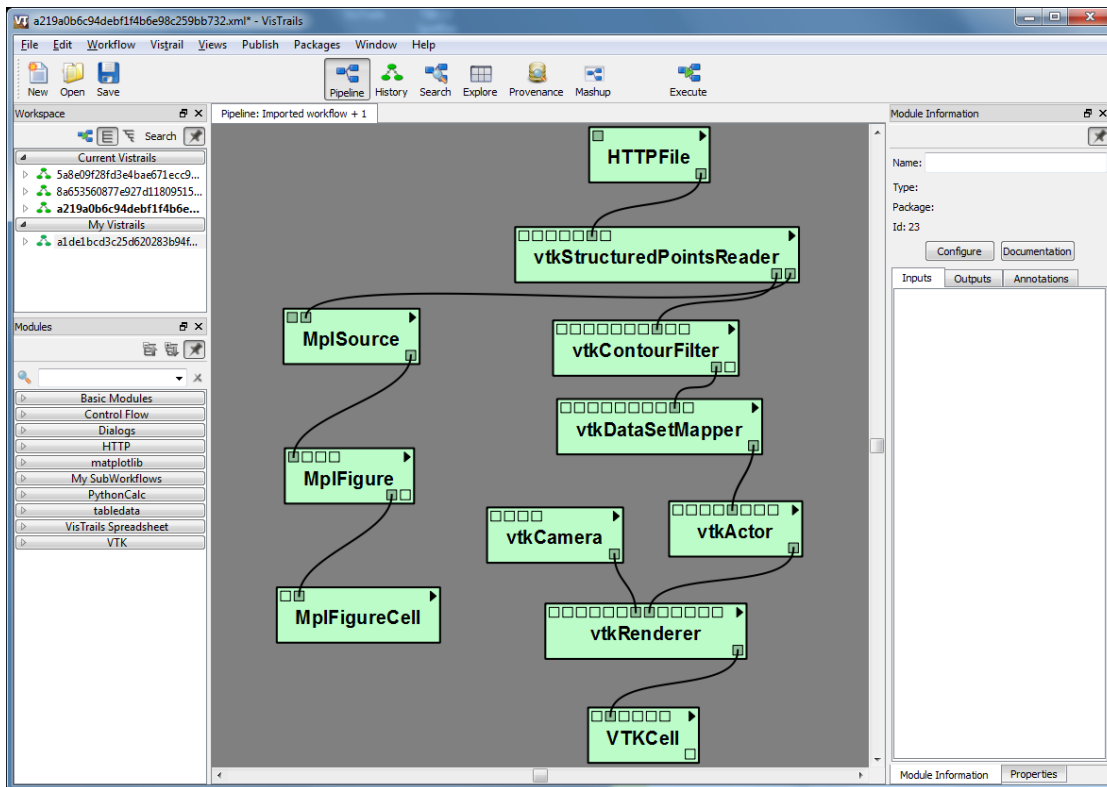


Figure 3.4: A screenshot of the user interface of VisTrails [17]. Here, different modules can be plugged together to allow a detailed specification of the visualization pipeline.

of large datasets.

Wu et al. [37] proposes a similar approach but introduces mechanisms to dynamically distribute pipeline steps to nodes according to visualization needs as well as time-varying network and node conditions. This self adapting method aims to use the available resources as optimal as possible. The aim is to establish a pool of resources which can be used by many users for different applications. Due to the growing size of datasets and system resources which are connected via the internet and located around the world a self adaptive system is needed. Cost models are introduced which take processing and transfer times into account to compute optimal pipeline configurations.

Ahmed et al. [16] distributed the components of the visualization pipeline over a grid-computer based environment for scalability reasons. Due to limited processing power or memory desktop computers provide insufficient performance to visualize large medical datasets. Today this is often the case when datasets from medical detectors or simulations have to be visualized. Therefore Ahmed et al. proposes a method to integrate a scientific visualization application in a grid-computing environment. Mainly this is supposed to use the performance of a grid environ-

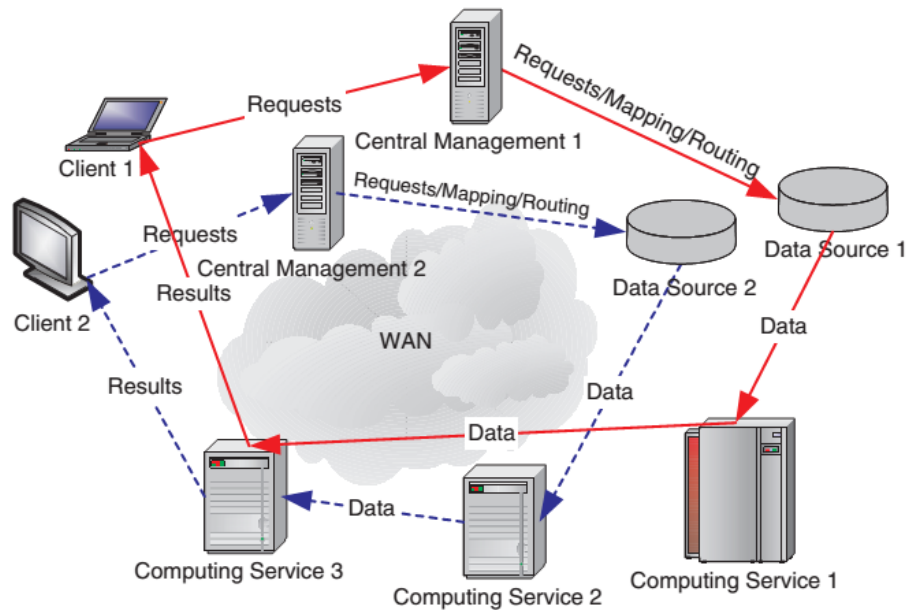


Figure 3.5: The proposed architecture "Distributed Remote Intelligent Visualization Environment (DRIVE)" of Zsu et al. [47]. It consists of a number of *virtual service nodes* which work together in order to minimize the end-to-end delays of the pipeline.

ment to do distribute processing steps to different node. Due to the implementation visualization operations get broken into small sub tasks and get distributes to nodes where they get processed by modules of the Visualization Toolkit [40] as previously introduced.

3.3 Conclusion

As can be seen by the number of publications in the last two decades the research area of information visualization became bigger and bigger. Today many applications rely on these well established concepts, e.g. the visualization pipeline. A variety of visualization applications are based on this concept while using it in many different ways. The Visualization Toolkit [40] was one of the first approaches to implement the steps in separated modules. Fekete et al. [23] took the next step and specializes the pipeline for information visualizations by introducing the Info-Vis Toolkit. Prefuse [29] improved the usability by allowing the users to put together the components in a graphical user interface without having programming skills. VisTrails [17] allows establishing a much more detailed series of preprocessing steps in order to map a whole scientific workflow into the application. It offers features to acquire data from different sources and provides mechanisms that allow to output multiple visualizations. Due to the growing amount of data especially in scientific scenarios desktop solutions became insufficient. Ahmed et al. [16]

therefore introduced a solution by implementing steps of the pipeline in network base grid for performance feature.

Today the visualization pipeline has been used as a model in application for more than 20 years. "The combination of simplicity and power makes the visualization pipeline still the most prevalent metaphor encountered today." (Moreland [35]) It is the basis of many visualization solutions and therefore an essential part of this thesis.

Due to the stated publications the following conclusions can be derived:

- Depending on the application it may be advantageous to develop the modules of the pipeline in different programming languages.
- The pipeline does not have to run on a single machine. It can be distributed on multiple machines that are connected via a network.
- If each step is computed on specifically designed hardware, resources may be used more efficiently and may result in a better overall performance.
- The output of the pipeline does not have to be displayed only in a single view. Multiple visualizations can be used to display data simultaneously.

As a result it can be mentioned that the literature provides approaches that can be used to define a solution for the previously defined problem (see Section 1.1).

Software Design

This chapter describes the existing environment in the first part. The second part describes how the defined techniques from the background chapter (see Section 2) can be put together and applied to the existing environment to realize this software design concept.

4.1 Overview

This thesis proposes an additional feature for BrainGazer [20], an existing software environment for biomedical image analysis, mining and retrieval. BrainGazer offers rendering features for volumetric images and aims to support scientists in the research of neuronal structures of the *drosophila melanogaster*. The user can operate with the environment by two different interfaces (see Figure 4.1):

- **Web-based database query interface:** A webpage that offers different textual search features. Here, the user can search for images or neurobiological objects in the database.
- **BrainGazer:** A client-sided desktop application that offers rendering features for volumetric images.

The user can access the web-based query interface by any browser. Exacly the same web interface can be accessed by BrainGazer. Thus, the user is able to use the same search interface from within BrainGazer. Here, the user can save instances of the results directly in the application for faster access and easier handling.

4.2 Existing Client-Server Environment

The existing environment components (see Figure 4.1) can be grouped into a client and a server side. The server components are single entities whereas the number of clients is not limited. The server side consists of:

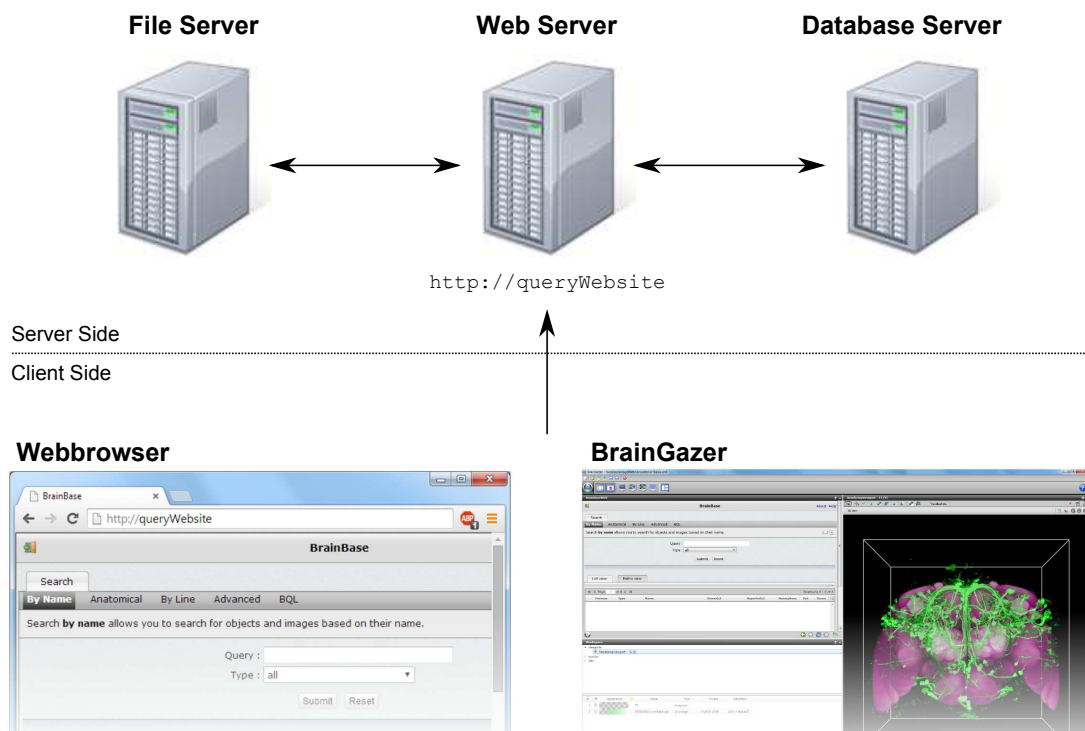


Figure 4.1: Client server structure showing the basic components of the environment. A web server hosts the web interface (see Section 4.2), a database server and a file server. This web site allows the user to query the database and enables user to download files form the file server.

- Web Server: Hosts the database query interface (web site)
- Database: Contais data records with relations to images
- File Server: Stores image files

When the user browses the query web site the web server runs the queries on the database. Depending on the object type the user can download results from the file server.

Web-based Query Interface

The web server provides a web-based query interface, an ordinary web site to allow the user to search for volumetric images and neuronal structures (objects). The query web site offers several query features on different pages to enable searches by specific neurobiological properties. Furthermore it enables the user to browse objects based on the relation between them. If the user intends to search for an object the following steps are performed.

1. The user specifies a query in the search interface.
2. The search request gets send from the web site to the web server.

3. The web server performs a query on the database server and receives the result.
4. These results get sent back to the web site where they are listed on a result page.

This web interface is embedded as dedicated window (see Section 4.2) in BrainGazer, the client application of the environment. As a result the user is able to use the same search features of the web interface from within the application.

Client integration

The web interface can either be accessed by any web browser or by using BrainGazer, the client application of the environment. If it is accessed by a web browser it offers the previously described (see Section 4.2) query features to browse the database.

If the web interface is accessed by BrainGazer the query features of the web interface get extended by advanced rendering and visual query features [20] - features that cannot be realized by web technologies efficiently today. The web interface enables the user to save references of search results in a client-side data structure called the *workspace*. The web site offers features to transfer object references from result or detail pages to the workspace.

4.3 Existing User Interface

BrainGazer offers a multiple-coordinated view user-interface and supports a linking mechanism between the user interface elements (see Section 2.3). The windows inside the application are called *viewports*. Each viewport serves a specific purpose and can be adjusted in size and position according to the user's needs (see Figure 4.2).

So far BrainGazer offers the following types of viewports:

- Database: Web-based query interface (see Section 4.2) to perform queries on the database. The results can be added to the workspace.
- 3D View: Provides rendering features for three dimensional object data.
- NeuroMap [42]: Graph visualization to illustrate relations between arborizations.
- Heatmap: Used to visualize how much a set of neurons overlap each other.
- Workspace: Enables the user to manage images and neurobiological objects in lists.

These existing views can be used to browse the database for images and neurobiological objects, import *instances* of them in the local workspace and visualize them in a rendering viewport.

The architecture of BrainGazer "is based on a flexible plug-in mechanism which allows independent modification or even replacement of system components." (Bruckner et al. [20]) This work aims to extend the environment by additional data visualization viewports respectively *InfoVis viewports*. The user interface supports multiple-viewport instances of a certain type. Therefore new InfoVis viewports can be used to display different visualizations simultaneously.

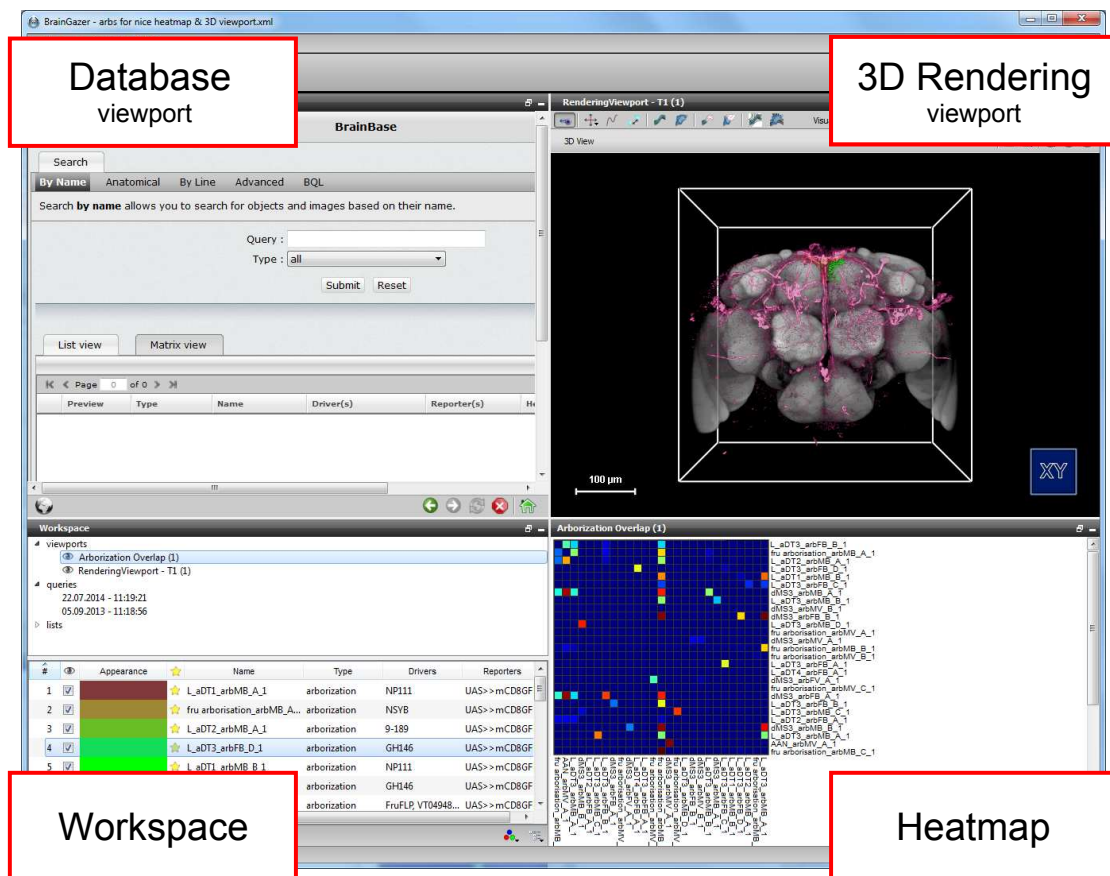


Figure 4.2: A screenshot of BrainGazer [20] showing multiple viewports in the existing user interface.

Workspace Concept

The *workspace* (see Figure 4.3) is a special section of the user interface of BrainGazer (see Figure 4.2). Here, the user is able to manage viewports, queries and organize items in lists. The workspace concept can be seen as a combination of the user interface and the specifically designed workflows.

As described in Section 4.2 the user starts an investigation with BrainGazer by searching for images or neurobiological objects in the embedded web-based database query interface. From here the user is able to select a set of images or neurobiological objects and load them as *instances* into BrainGazer's workspace. By doing this the selected instances appear as list entries in a new *query list*. If the user selects one of this lists the contained items are displayed in the list below, the second part of the workspace. (see Figure 4.4)

Initially the set of instances of a viewport is empty. The user has to associate instances to the viewport manually by using mouse actions. By dragging a set of instances from e.g. a query list

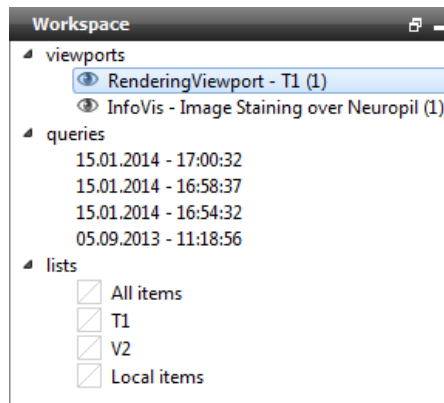


Figure 4.3: A screenshot of the top section of BrainGazers workspace. It contains listings of viewports and query lists. The 'lists' section allows the user to group data items to user defined lists or filter data items due to predefined filters.

#	Appearance	Name
1	<input checked="" type="checkbox"/>	L_aDT1_arbMB_A_1
2	<input checked="" type="checkbox"/>	fru arborisation_arbMB_A_1
3	<input checked="" type="checkbox"/>	L_aDT2_arbMB_A_1
4	<input checked="" type="checkbox"/>	L_aDT1_arbMB_B_1
5	<input checked="" type="checkbox"/>	dMS3_arbMB_A_1
6	<input checked="" type="checkbox"/>	fru arborisation_arbMB_B_1
7	<input checked="" type="checkbox"/>	dMS3_arbFB_A_1
8	<input checked="" type="checkbox"/>	L_aDT3_arbFB_B_1
9	<input checked="" type="checkbox"/>	dMS3_arbMB_B_1
10	<input checked="" type="checkbox"/>	L_aDT3_arbMB_A_1
11	<input checked="" type="checkbox"/>	fru arborisation_arbMB_C_1

Figure 4.4: A screenshot of the viewports item list. The checkbox controls the visibility of items in the viewport whereas the color boxes can be used to open a color selection dialog.

and dropping them on a viewport instances get associated to the viewport. After dropping the instances the view gets updated automatically. The set of instances then appears in the workspace as separated *viewport item list* (see Figure 4.4). To enable a better visual separation between different instances each viewport list item supports individual visual properties (color, appearance, visibility). To share a set of instances and their visual settings among multiple viewport BrainGazer supports *viewport groups*.

A vital component of the workspace concept and important feature for this thesis is the *linking mechanism* between the views. All the viewports and the workspace support brushing [45]. When the user selects a certain instance or set of instances, the same instances get highlighted in every other viewport where an instance of the same object is present in the item list.

4.4 Architectural Design Overview

This section describes how the visualization pipeline can be used to extend the existing application and its environment. To enable the integration both client and server side need to be taken into account.

Adapted Visualization Pipeline

This design concept takes advantages of the client-server architecture by implementing the individual modules of the visualization pipeline (see Section 2.1) on both the client and server side as illustrated in Figure 4.5.

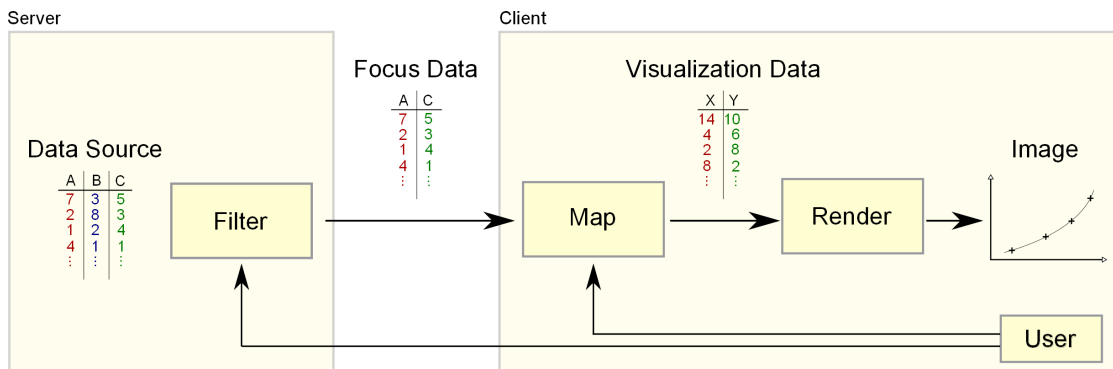


Figure 4.5: This figure illustrates how the three modules of the visualization pipeline (see Figure 2.1) get distributed on the existing client server structure. The data acquisition step is not part of this design concept. The filtering takes place on the server side while clients are responsible for mapping and rendering step. The user is able to control the filter on the server side and the mapping on the client side.

This separation has the following advantages:

- It enables the distribution of pipeline modules to different components of the architecture (see Figure 4.1). The filtering process takes place on the server side while both mapping and rendering are performed by a client. Servers usually provide more memory and processor resources while client PCs often contain better graphics hardware.
- The server is able to provide its services to multiple clients.
- Implementing the visualization pipeline in a modular way allows developers to exchange certain parts. A single server can host multiple filter modules. Each individual module provides different datasets. Here, the main data source is the database but other data sources can be considered as well.
- On the client side it allows the user to change the type of visualization.

The data acquisition is not part of this proposed design concept. For this thesis it is assumed that the relevant data is stored in accessible data sources like databases or file servers. Due to the lack of the import step the pipeline gets reduced to three modules. Consequently the reduction of modules reduces the number of connections as well. Due to the distribution to client and server-side special care has to be taken on the connection between filter and map module.

Pipeline Course

A client initiates the pipeline by sending a user defined request to the server. The server is responsible to answer the request and return the focus data to the client. Therefore it is necessary to ensure that clients receive individual focus datasets according to their requests. This connection is essential in order to allow users to control the filter module.

If the filter module receives a request from the client it has to perform a query on the database. Therefore the server-side environment usually provides a query language (e.g. SQL) to retrieve the stored information. This result dataset, the focus data, gets now sent back to the client.

After receiving focus data client-side procedures start the mapping step. The user may control the module either by adjusting workspace item properties (see Section 4.6) or by changing visualization specific parameters (e.g., color scale for a heatmap). The map procedure ends with the rendering process which is performed by the viewport. Depending on the used visualization techniques the viewport is responsible for the rendering procedures.

One goal of this software design concept is to develop a modular framework. This framework requires both the implementation of modules as well as a software design for an infrastructure to hosts the pipeline modules. Consequently implementations are required on both the server and on the client-side.

4.5 Server features

This section concentrates on the server-sided software architecture. First, the design of both infrastructure and filter modules according to the visualization pipeline is introduced. The subsequent sections introduce relevant implementation details.

Overview

The server is responsible for the filtering step of the visualization pipeline (the importing step is not part of this thesis). Due to the modular design it is required to host a set of different filter modules. Therefore a specifically designed infrastructure is required. The user on the client side creates a request by using BrainGazer which is sent to the server to initiate the pipeline process.

To allow users to select a specific filter module an addressing scheme is necessary. Therefore the request must to contain identifier information to allow the *dispatching mechanism* to decides which filter module to use (see Figure 4.6). After the corresponding filter receives the request it queries data sources and returns the focus data to the client.

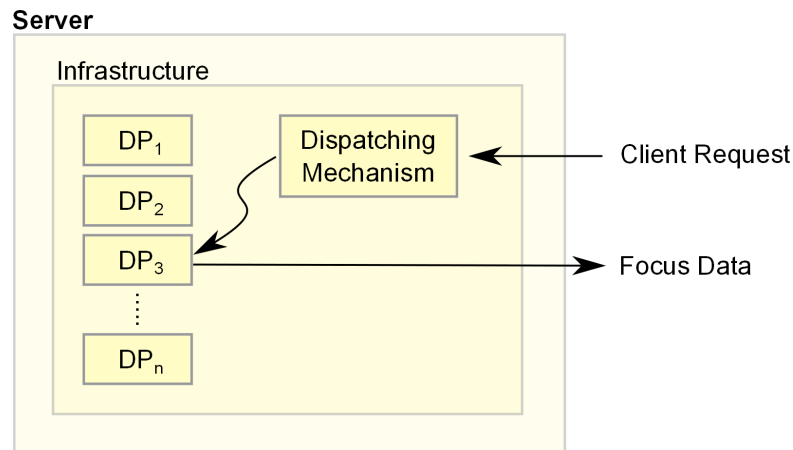


Figure 4.6: The server-sided infrastructure provides a dispatching mechanism which enables developers to implement a set of different data providers (DP). According to an identifier that has to be part of a request the inquiry gets directed to the specified data provider to start the filtering process.

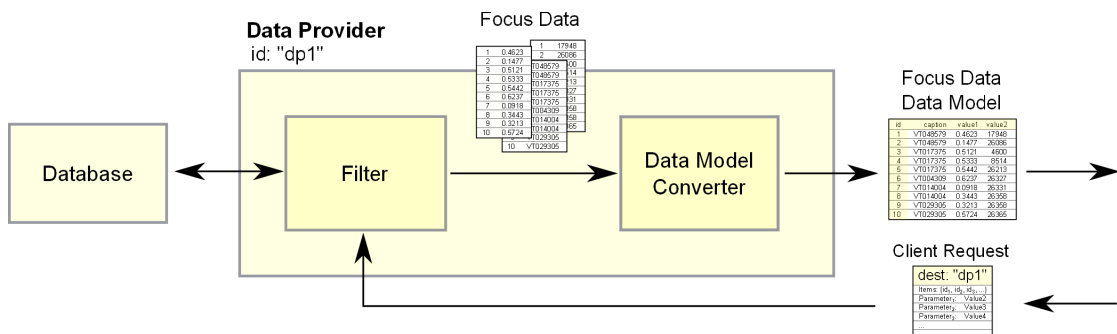


Figure 4.7: The server sided components of the implemented visualization pipeline. The client initiates the pipeline by sending a request. According to this input the filter queries the database and the *data model converter* converts the results to a *standardized data model*.

As introduced the filters do not only have to query data sources, they have to accept and process requests as well. For this additional functionality this thesis defines the term *data provider*. A data provider is an filter instance which is responsible for (see Figure 4.7):

1. Accepting client requests
2. Performing queries on different data sources (the filtering step), e.g. databases
3. Return focus data back to client

Technically they can be seen as server sided filter instances designed to query one or more data sources.

The data provider and the hosting infrastructure can be (but don not have to be) hosted on the same a web server that is being used to host the database query interface (see Section 4.3). The server functionality (web site hosting, see Section 5.1) can be extended to enable the filter step. If using the same web server no additional server is required.

Data Provider Infrastructure

Figure 4.7 points out all the components to host data provider modules. On arrival of a request the dispatching mechanism decides which data provider to contact. In order to allow making this decision requests must contain addressing information. After dispatching the corresponding data provider examines the request and performs queries on at least one data source, i.e. a database.

A single data provider prepares and returns focus data for a specific use case. Different use cases may require data in individual formats. To enable the modular approach on the client side the focus data has to be returned in a consistant *data model*. The aim of this model is to be compatible to different types of visualization on the client side. Focus data often comes as tabular data but more complex use cases might require the visualization of graph or network structures. Hence, it is essential for the modularity of the concept to convert responses into a standardized data model before sending it to the client.

Most visualizaton tools support tabular data models [22]. Due to the biological context of this thesis different data structures might have to be visualized. Therefore different types of data models can be possible:

- Table Model: A two dimensional table with a fixed number of columns and an open number of rows.
- Matrix Model: Similar to the table but with equal number of rows and columns.
- Network Model: A data model containing nodes and vertices.

In the corse of this thesis a standardized table data model is implemented (see Section 5.2).

Web Environment

Due to the existence of the web server in the environment (see Section 4.4), communication technologies based on web protocols are used to establish the link between filter and map module. These protocols are capable to transfer requests and focus data between the server and a client. Thus, it is mandatory to introduce them on the client side as well.

In general, webpages are implemented as HTML files and stored on a web server. Here, the files can be located by using a combination of the servers address (URL) and their filename. A client requests these files by using a web browser. The web browser downloads the file and renders the webpage. In the context of this thesis, a *visualization webpage* is used to realize a data visualization. Here, the visualizations are implemented by using a combination of HTML and additional web technologies (see Section 5.3). These files get requested by BrainGazer and rendered in a specifically designed viewport.

Initially clients do not have information about the set of HTML files on the server. This requires an additional mechanism, a *visualization info service*, on the server side in order to allow clients to obtain information about the hosted visualization webpages.

4.6 Client Features

Overview

As illustrated in figure 4.5 the client side is responsible for the mapping and rendering parts of the visualization pipeline. Additionally the user is able to control certain steps. To continue the modular approach the client-side parts of the pipeline need to be exchangeable as well. Hence, from a single focus dataset more than one visualization can be realized (see figure 4.8). This enables the user to observe multiple visualizations of the same focus data simultaneously.

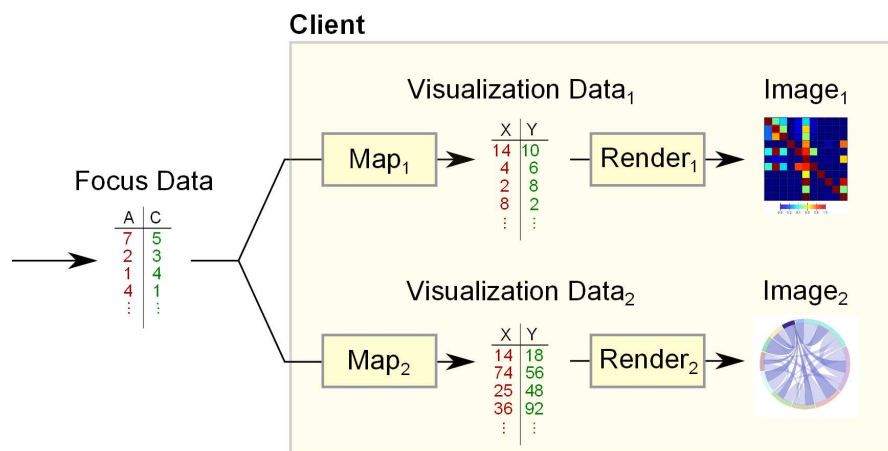


Figure 4.8: Client-side components of the visualization pipeline. A single focus dataset can be represented by multiple visualizations types simultaneously. The user may choose which type is the best to solve the given problem.

The mapping module gets the focus dataset from the filter and converts the data into a visual scale. Examples for visual scales are a position in a coordinate system, color or transparency. This step strongly depends on the type of the used visualization, e.g. for a heatmap matrix data

values have to be converted into colors. Depending on the type of visualization the user is able to control certain mapping parameters, e.g., the user may choose the color scale used by a heatmap.

The rendering is done by a specific BrainGazer viewport. These visualization viewports are configured to serve as web browser for visualization webpages. They are responsible for both downloading a single HTML file from the server and rendering the webpage in their view window. Users can set up the applications user interface by placing the viewports on the screen as desired (see figure 4.2).

Visualization Worksheets

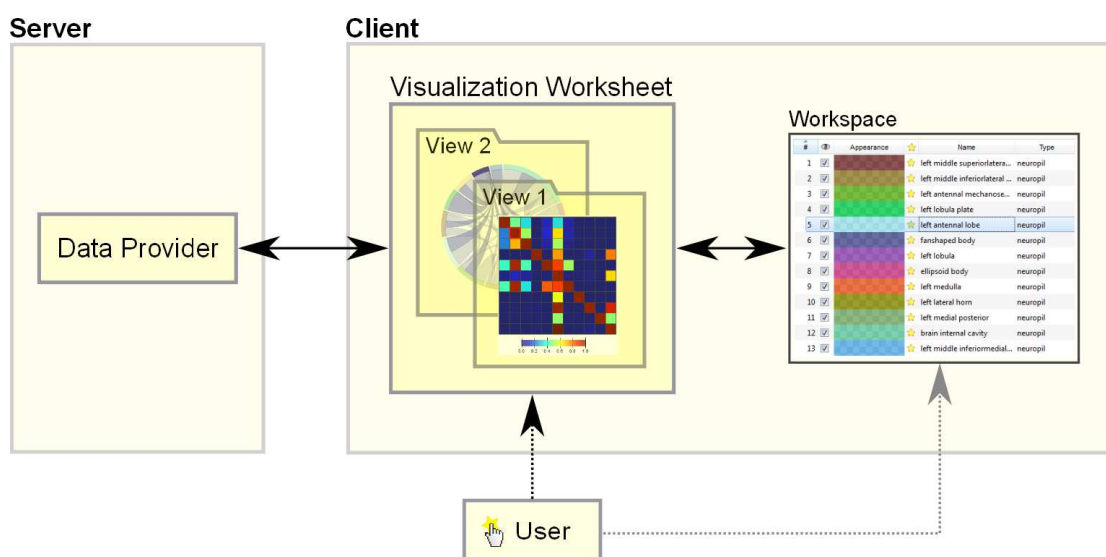


Figure 4.9: The links of a *visualization worksheet* to other system entities. A worksheet receives object identifiers from the workspace, has to request data from a data provider, host one or more visualizations and links the views to the workspace.

In this software design concept a *visualization worksheet* is a data visualization in a that aims to support the user in solving a certain use case. It is implemented as a single webpage and gets displayed in a BrainGazer viewport. If the user associates a set of instances to the visualization viewport (see Section 4.3) their object identifiers get included in a request and sent to a data provider in order to acquire a focus dataset (see Section 4.5). A visualization worksheet is responsible for:

- **Data Provider Connection:** Connect the client side to a data provider to acquire focus data according to the request.
- **Multiple Visualization Types:** Host one of more types of visualizations to display the focus dataset in multiple ways.

- **Workspace Integration:** Connect to BrainGazers workspace to enable the exchange of object identifiers as well as the linking and brushing mechanisms.

Figure 4.9 points out the relations between the entities. Consequently the developer has to decide which type of visualization is more appropriate to solve the given problem. The introduced visualization webpages (see Section 4.5) or worksheets have to be designed to fulfill the requirements of BrainGazers workspace concept to enable a seamless integration (see Section 4.3. The following section will describe these three points in detail.

Data Provider Connection

This requirement connects the worksheet, where the mapping takes place, with the filter part of the visualization pipeline. This module is responsible to acquire enriched datasets from a data provider on the server. A worksheet is bound to a specific use case and acquires enriched data from at least one data provider.

To start the pipeline process a request must be defined by the client (see figure 4.5, the user controls the filtering step). Therefore the user has to drop instances on the viewport. Consequently the *request* is created and sent to the server. It has to include an object identifiers, individual filter parameters and an identifier to allow the dispatching mechanism.

Multiple Visualization Types

To host multiple visualizations on a single visualization webpage a client-sided infrastructure is required. If the worksheet receives focus data the infrastructure has to provide the received dataset to all implemented visualizations. Furthermore the infrastructure is responsible for transmitting user interactions to the visualizations.

Workspace Integration

To fully integrate the visualization into BrainGazers workspace concept it is required to extend the existing linking and brushing mechanisms to the new visualizations. Therefore a bridge mechanism is needed to enable a data exchange between the viewport and the worksheet. This bridge mechanism has to transmit:

- **Workspace item identifier:** If the user drops instances on the viewport the corresponding object identifiers have to be transmitted to the worksheet. If the focus dataset contains related object identifiers a transmission to the viewport is necessary.
- **Viewport item settings:** The item settings of each individual viewport list item have to be transmitted to the worksheet (color, transparency, visibility).
- **Selections:** Selected objects have to be transmitted to the views and vice versa.

To address each item individually it is necessary to introduce a reasonable addressing scheme for instances between viewport and worksheet. Each object can be categorized to a certain type. Additionally an instance has an identification number. The combination of number and the type is a unique identifier and can be used to address instances in the workspace or visual elements in views, representing these instances.

Implementation

This chapter describes implementation details of all the methods introduced in the previous chapter. The first part describes how the server-side components are being realized. The second part concentrates on the client-side elements as well as the integration into BrainGazer.

5.1 Existing Environment

BrainGazer is the desktop application of the environment which is realized by C++ and the *Qt library* [9]. It implements instances of the *Qt WebKit* to embed websites into a desktop application. The WebKit allows developers to implement a *bridge interface* between the application and the embedded website to enable a communication between the two sides.

The web server is an *Apache Tomcat server* [2] which hosts web applications that are implemented by using the *Google Web Toolkit (GWT)* [12]. The GWT is used to implement both websites and server-side services in Java [8]. The GWT compiler creates an AJAX applications for the server together with websites including optimized JavaScript code for the client side.

5.2 Server Features

Overview

As introduced in the previous chapter (see Section 4.5), the server is mainly responsible for the filtering step of the visualization pipeline (see Figure 4.5). The server hosts a set of data providers where each provider includes an individual filter. The set of data provider instances is running on the server waiting for client requests. The processing of a data provider request can be separated into the following steps.

1. The server receives a request from a client.

2. The dispatching mechanism forwards the request to a data provider according to the identifier string.
3. The data provider examines the request and creates a data source query according to the included parameter.
4. In case of multiple involved data sources the results have to be joined.
5. The data model builder converts the result of the query, the focus dataset, to a standardized data model.
6. The data model Java object has to be converted to a string.
7. The method returns the data model string to the client.

To implement the server sided infrastructure *Java Servlets* are introduced. A Servlet is an instance of a Java class that provides a callback function which allows to respond to client requests. It can be implemented to host the set of data providers. The callback function gets processed when a request arrives to the server. Here, the dispatching mechanism is implemented.

Data Provider

As described in Section 4.5 a data provider contains a filter mechanism which requires certain input parameters. These parameters are essential to define the database query. They must include a set of object identifiers and, depending on the use case, filter specific settings.

To acquire data from a relational database queries have to be defined by the use of SQL (Structured Query Language [15]), a programming language designed to manage and query relational databases. This string based language allows to retrieve specific contents from the database. Therefore a SQL string has to be dynamically constructed to include all necessary request parameters. The next step is to execute the query and wait for the results.

After receiving the acquired results from the database server the focus dataset gets created. Depending on the use case it might be necessary to query more than just one database. In such a case the focus dataset is the combination of multiple result sets of the individual database queries. If the database queries are completed the standardized data model has to be created. Depending on the use case this conversion is an individual process that has to be implemented for each data provider separately.

Listing 5.1 shows the Java source code of a simple data provider. Depending on the use case raw data may exist in different data sources. Hence, if a data provider instance has to query multiple data sources it is necessary to merge the result to create a single data set. Finally this result set has to be converted into a data model which can be sent back to the client.

```
1 public class ExampleProvider implements IInfoVisDataProvider {  
2  
3     // unique identifier string  
4     public static String identifier = "DataProviderID";
```

```

5
6 @Override
7 public String getIdentifier() {
8     return identifier;
9 }
10
11 @Override
12 public void provideData(HttpServletRequest request, HttpServletResponse
    response) {
13
14     // convert HttpServletRequest to a request
15     DpRqxObject requestdata = convert(request);
16
17     // create queries
18     String query1 = "SELECT * FROM ... WHERE ... " + requestdata.parameterX;
19     String query2 = "SELECT * FROM ... WHERE ... " + requestdata.parameterY;
20
21     // establish connections to data sources
22     DatabaseConnection con1 = new DatabaseConnection(source1);
23     DatabaseConnection con2 = new DatabaseConnection(source2);
24
25     // perform queries
26     ResultSet results1 = con1.execute(query1);
27     ResultSet results2 = con2.execute(query2);
28
29     // join the result sets to get the enriched dataset
30     ResultSet finalResult = joinResults(result1, result2);
31
32     // convert focus dataset into a data model structure
33     DataModel dm = buildDataModel(finalResult);
34
35     // create JSON string from data model
36     String resultJSON = new Gson().toJson(dm);
37
38     // send data model back to client
39     response.write(resultJSON);
40 }
41 }

```

Listing 5.1: This example shows the Java source code of a simple data provider. The method *provideData(...)* gets executed directly after the dispatching step.

In the first line of the code example (Listing 5.1) the class is declared and the data provider interface is implemented. The interface requires the implementation of the *getIdentifier* method (line 7) which returns the identifier string for the data provider host. The filtering and data model conversion is implemented in the *provideData* method (line 12). The query procedure starts by examining the client request (line 15) and creating database queries in consideration of the request (line 18). The SQL queries are composed by item references and additional parameters included in the request. This is how database records related to specific neuronal objects can be retrieved from data sources. After performing the queries (line 26) the two result sets have to be joined (line 30) to get a single focus dataset which can be converted into a data model (line 33). This data model is an instance of a java class and needs to be converted into a string in order to

allow the transmission to the client. The result of this procedure is a string which gets returned to the client (line 39).

Data Model

The focus dataset that needs to be transferred to clients have to follow certain criterias. First the dataset can only be transferred as a string. Only a string based data structure can be processed by the web technologies used on the client side. Furthermore the dataset has to be converted in a standardized format to enable the proposed modular approach on the client side as well. Therefore a Java class is implemented to simplify the conversion to a string. This class is capable to store a tabular shaped dataset and provides methods to create a dataset (see Table 5.1).

<code>void addColumn(String id, String datatype, HashMap<String, Object> attributes)+</code>	Adds a new column to the data structure and append attributes. Required for initialization.
<code>void addColumn(String id, String datatype)</code>	Adds a new column to the data structure without any attributes. Required for initialization.
<code>int getColumnCount()</code>	Returns the column count of the instance.
<code>addRowData(String id, Object[] data, HashMap<String, Object> attributes)</code>	Adds the elements of the data array to the instance. The elements data type must correlate with the data type descriptors of the columns.

Table 5.1: The methods of the implemented table data model. The purpose of this data model is to allow a simple conversion to the JSON format. Update or delete methods are not required.

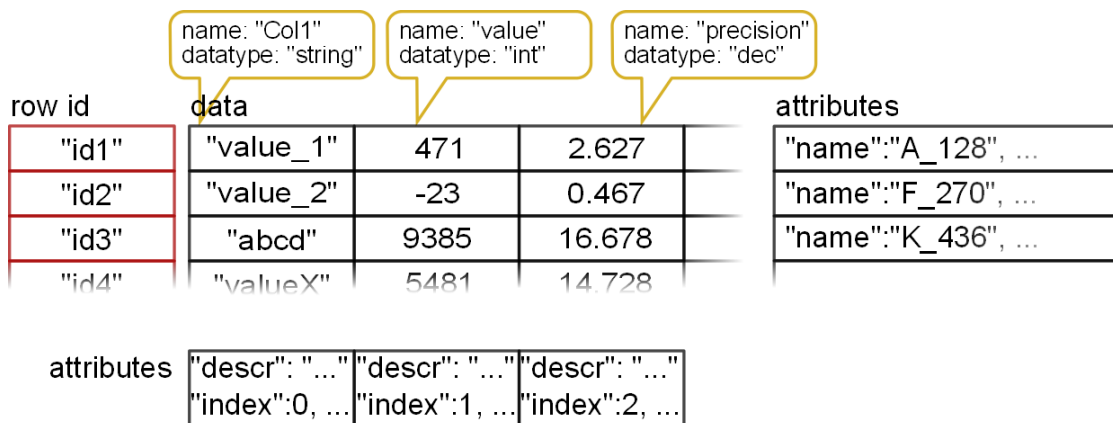


Figure 5.1: An illustration of the implemented table data model. It allows the developer to define various columns for different data types and supports additional attribute elements for each row and column.

The implemented table data model (see Figure 5.1) is capable to store a two dimensional array of data values with additional attributes for each individual row and column. Before using the table model, the developer has to define the data type (integer, float, string) of the column. This has an effect on the conversion to a string.

```
1 {
2   columns : [
3     {"id":"col1", "datatype":"string", "name":"Col1", "attr2":34},
4     {"id":"col2", "datatype":"int", "name":"value", "attr2":21},
5     {"id":"col3", "datatype":"dec", "name":"precision", "attr2":15},
6   ],
7   rows : [
8     {"id":"id1", "name":"A_128"},
9     {"id":"id2", "name":"F_270"},
10    {"id":"id3", "name":"K_436"},
11  ],
12  data : [
13    ["value_1", 471, 2627],
14    ["value_2", -23, 0.467],
15    ["abcd", 9385, 16678]
16  ]
17 }
```

Listing 5.2: The representation of the previously introduced table data model (see Figure 5.1) as JSON string. The focus dataset is sent to a client via this format.

JSON [13] is a string based and human readable text format. It can be simply generated and is supported by many programming languages. To convert an instance of a Java class to a JSON string the GSON class library is introduced. GSON [5] is an open source Java library that supports conversions from Java objects to JSON strings. The result of a conversion is illustrated in Listing 5.2.

An instance of the GSON class library is used by the data provider class to convert the focus dataset into a JSON string (see Listing 5.1, line 36) which gets returned to the client as response of the request.

Data Provider Host

The *data provider host* is responsible for hosting a set of data provider instances. As described in Section 5.2 the host is implemented as GWT Servlet class. Instances of this class can be contacted by *HTTP requests* using *remote procedure calls (RPC)* from clients across a network. The client addresses servlets by using a composition of the server domain name, servlet name and the data provider identifier string (see Figure 5.2).

To realize this mechanism a *map data structure* is introduced. A map can be used to store a set of key-value pairs. While a key has to be unique by definition the value can hold any data structure of a certain predefined type.

The data provider host is implemented as *HTTP servlet class*.

Server URL Servlet

`http: // myserver / infovisdata / DPx`


DataProvider identifier string 

Figure 5.2: An URL (Uniform Resource Locator) is a string to identify and locate a resources in a computer network. This way, clients are able to address the server. Additional to the servers URL data providers can be addressed in this string as well. The dispatching mechanism in the Servlet examines the data provider identifier string to direct the request to the requested data provider.

The data provider host separates the identifier from the URL string and uses it to address the corresponding data provider instance in the map data structure. The filter method of the data provider gets called with the request (see Section 4.5) as parameter. If the identifier string cannot be found in the map the dispatching mechanism responds with an error message.

Visualization Infrastructure

To host the visualization web pages an ordinary web space is needed to host *.html* files which contain the HTML code of the visualization worksheets (see Section 4.6). No additional server components are required because the rendering of the web pages takes place on the client side. It is essential that this web server is accessible for clients to allow them to acquire these web pages.

5.3 Client Features

Overview

The implementation of the client features consists of two parts. First the web based implementation of the mapping and rendering step will be described as well as the link to the server. The second part explains the extensions to the existing application to allow a communication between the application and the embedded visualization websites (bridge interface).

Client-side Pipeline

This section focus is the assembling and 'wiring' of the client-side components. Examples implementations with visualizations will be described in the evaluation section of this thesis (see Section 6.2).

Mapping

As previously introduced the mapping of the visualization will be realized by web technologies like HTML and JavaScript. HTML is used to create a basic web page layout while JavaScript is required to realize visualizations and interaction methods. Here state-of-the-art JavaScript

libraries can be introduced to establish the worksheet infrastructure and realize views. This client-side infrastructure has to provide possibilities to implement more than one data visualization on a single web page.

Rendering

The rendering is performed by *InfoVis viewports*, a BrainGazer viewport configured to display a single web page. It processes the HTML and JavaScript code of the involved HTML files and renders the web page. The final result of this step is displayed in a viewport. This step can be compared to the rendering of a web page in a browser but without navigation features (address bar, bookmarks, ...). An InfoVis viewport is set to display a specific visualization website providing one or more different data visualizations.

Client-side Infrastructure

As stated in Section 4.4 the goal of this thesis is to develop a modular framework. The pipeline components have to be modular to allow a flexible usage in worksheets (see Section 4.6). Especially views must be designed so that they can be applied to different datasets. Therefore it is mandatory to introduce the *model-view-controller design pattern (MVC)* (see Section 2.4) for views.

To avoid the implementation of the needed design patterns from scratch existing JavaScript frameworks can be introduced. AngularJS [1] is an open source JavaScript library that encourages the implementation of the model-view-controller design pattern in web applications. The next section will introduce selected features of AngularJS. These features are needed in order to describe the implementation of the model-view-controller in this thesis.

Introduction to AngularJS

AngularJS [1] is designed to establish a clean separation between the three involved components: the model, the controller and the view. The *model* can be any kind of data structure while a *view* is a representation of the model, e.g. a table on a website. A *controller* is necessary to bind the model to the view.

Directives are additional markers for HTML, i.e., attributes or tag names. AngularJS introduces a series of built-in directives to simplify the implementation of common tasks, e.g. displaying a list (see Listing 5.3).

```
1 <ul ng-controller="MyCtrl">
2   <li ng-repeat="person in employees">
3     {{person.firstname}} {{person.lastname}}
4   </li>
5 </ul>
```

Listing 5.3: This example creates a simple list of names included in the 'employees' scope variable by the use of *ng-repeat*. This directive can be used to repeat a *li* element for each person

in the list of employees. In this example the variable `employees` is the model and the HTML list is a *template* for the view.

To enable the model-view-controller functionality a *controller* is necessary to glue the model and the view together. The actual view is realized when the snippet gets finally executed in a web browser. AngularJS calls the set of data structures in the model the *scope* (see Listing 5.4).

```
1 function MyCtrl($scope) {  
2   $scope.employees = people;  
3 }
```

Listing 5.4: A simple definition of a controller named 'MyCtrl' with a scope variable '\$scope.employees'. Here, the scope variable gets initialized by the content of the JavaScript object named 'people'. The controller gets assigned to the view by the *ng-controller* directive (see Listing 5.3).

As a result every time when the scope variable is changed the view in the browser window is updated to the new set of items. AngularJS supports event handler functions called *watch statements* (see Listing 5.5) which can be bound to scope variables. As a result such procedures are executed whenever the value of the supervised scope variable gets changed.

```
1 $scope.$watchCollection('employees', function(newSet, oldSet) {  
2   // do something  
3 });
```

Listing 5.5: This snippet shows a watch statement that monitors the scope variable 'employees'. Here 'watchCollection' is used to monitor the number of elements in the scope variable 'employees'.

User defined directives

AngularJS introduces multiple built-in directives for commonly used functions but allows developers to define individual directives. Thus they can simplify the HTML code of the web page for specific tasks. For example the code of Listing 5.3 can be adopted into a template and declared as directive.

```
1 <userlist listdata="employees"></userlist>
```

Listing 5.6: The HTML code of Listing 5.3 can be converted into a template to simplify the usage. The input data for this directive gets handed over via an attribute.

Usage of AngularJS

For the implementation of this software design concept AngularJS is primarily used to simplify the implementation process of new visualization worksheets. AngularJS features enable developers to minimize the implementation efforts. Here, AngularJS is specifically used for:

- The implementation of visualizations can be simplified by creating user defined directives for each visualization. The focus dataset or mapping information can be handed over by

attributes (see Listing 5.6). These directives can be used to implement different data visualizations as individual modules. As a result they can be included in different worksheets.

- Certain data sets can be stored in scope variables. Watch statements (see Listing 5.5) can be introduced to react to changes of certain data structures like item lists, focus datasets or mapping parameters (see Section 4.6). Furthermore scope variables can be used to share data sets among multiple directives.
- AngularJS encourages the implementation of reusable modules. The same modules can be used in different worksheets.

The worksheet infrastructure (see Section 5.3) depends on these aspects.

Worksheet Infrastructure

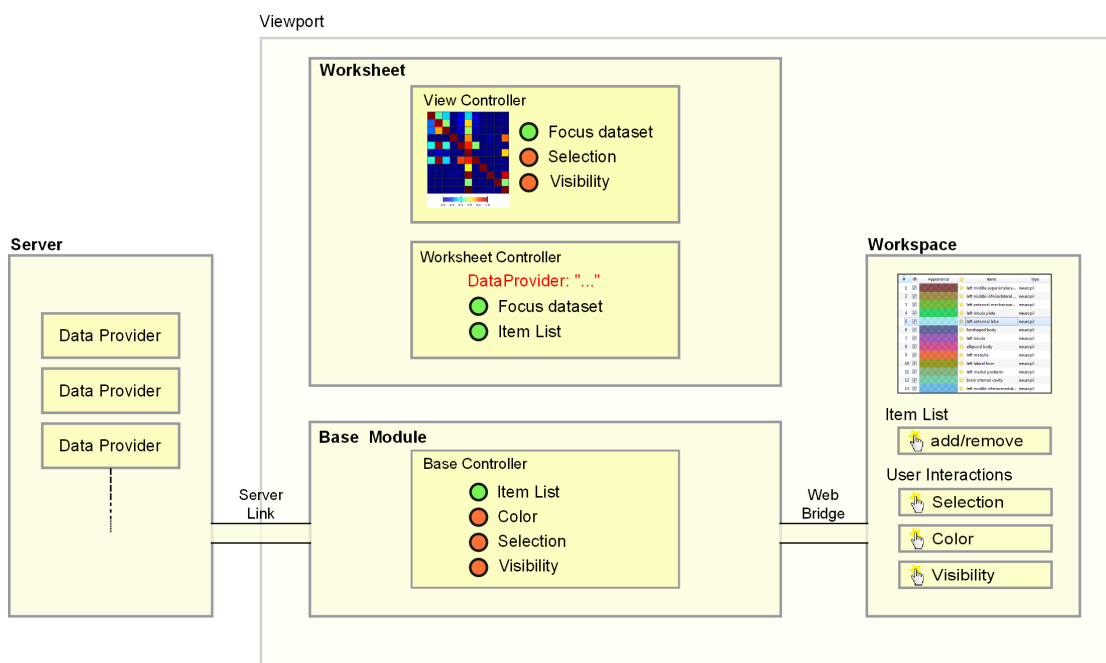


Figure 5.3: The framework of a visualization web page. The orange and green cycles illustrate scope variables in different modules. If the values of these variables change, watch-procedures (see Listing 5.5) is executed. Green dots represent variables responsible for item lists and the focus dataset, while red dots point out workspace settings.

As can be seen in Figure 4.5 the user is able to control both the filter and the mapping step of the implemented pipeline structure. Therefore connections between the modules and control mechanisms are a vital part of this software design concept. This section describes the implemented AngularJS infrastructure for visualization worksheets which is responsible to enable the

proposed pipeline functionality on the client-side. As illustrated in Figure 4.6 a worksheet requires links to the workspace and the server. Both are necessary to fulfill the requirements (see Figure 5.3).

- **Server Connection:** For the link between a visualization worksheet on the client side and data provider on the server side both web-based, e.g. the servers URL and worksheet specific parameters like the involved data provider are required.
- **Workspace Connection:** The link to the applications workspace concentrates on the integration into the existing workspace concept (see Section 4.6). To exchange object identifiers and linking and brushing parameters a fixed set of functions is required to transmit item references and their individual settings.

All the static parameters for both links are summarized in a specifically designed module. This module represents the basis for the connection functionality and is designed to be reusable for all worksheet implementations. It defines scope variables for the workspace integration and provides a template for the server connection (AngularJS factory).

The base module must be implemented by every worksheet. Consequently worksheets inherit the set of scope variables as well as the connection template. As a result the scope variables of the base module can be watched by the worksheet in order to react to changes. This mechanism works in both ways. The base module monitors the same scope variables as the worksheet to transmit changes to the workspace.

Due to the combination of the base module and a worksheet module the effort to implement a new worksheet is reduced to a minimum. The developer has to monitor the base module's scope variables by watch statements and hand over updates to views. Hereby it is not necessary to watch all scope variables from the base module. Depending on the visualization type the developers have to decide which features can be supported by the view.

Server Connection

When the user adds items to the viewport, the item references get sent to the item list scope variable of the base module (see green arrow in Figure 5.4). The worksheet controller detects this update via a watch statement and creates a request including the new set of items.

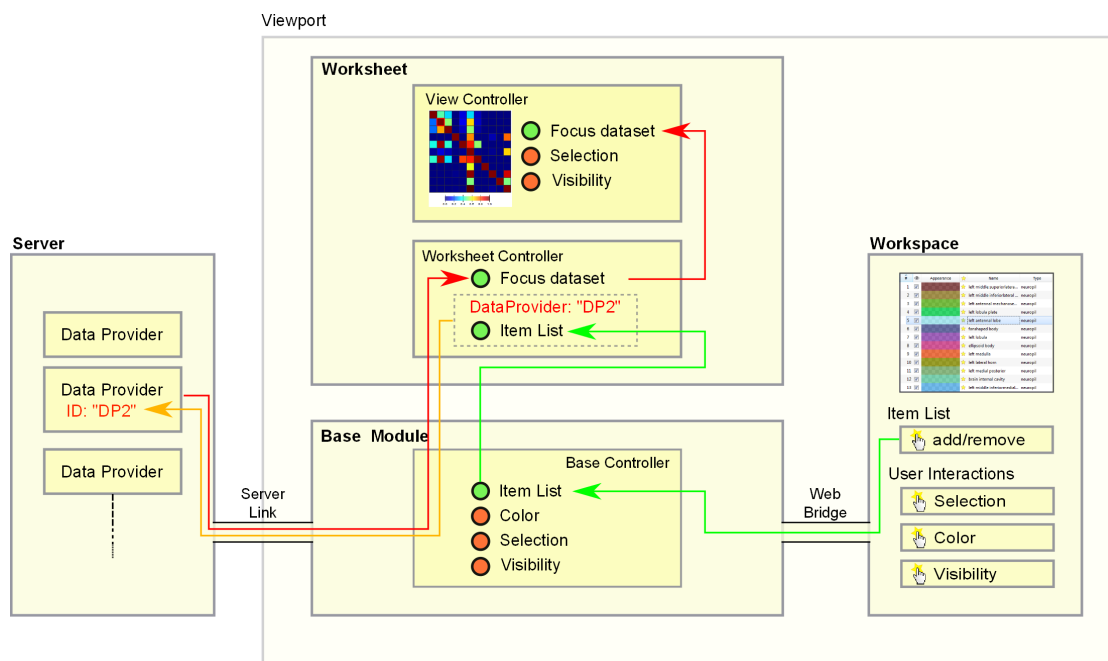


Figure 5.4: When the user adds or removes items from the viewport, the list of items in the base module gets updated as well as the item list of the worksheet controller (green arrow). The worksheet controller defines the request and creates a data provider service with a data provider id and the request as parameter (orange arrow). The focus dataset is received by the worksheet controller. This triggers the view controller(s) to update the visualizations.

```

1 angular.module('NgInfovisBase', [])
2   .factory('dataProvider', function ($http, bgBridge) {
3     return function (dataProvider, rqxdata, success, error) {
4
5       var serverURL = getServerURL();
6       var url = serverURL + "infovisviews/data/" + dataProvider;
7
8       $http.post(url, JSON.stringify(rqxdata))
9         .success(function (data) {
10           success(data);
11         })
12         .error(function (data) { error(data); });
13     };
14 })

```

Listing 5.7: Definition of a data provider factory in the base module 'NgInfovisBase' module. This factory can be used by worksheets to initiate connections to the server

The link to the server is established by using the connection template of the base module (AngularJS factory). To use this factory to create a service the following parameters are required (see Listing 5.7, line 3):

- A data provider identifier string.
- The payload of the server request.
- A success function which is executed if the server successfully returns the requested focus dataset.
- An error function in case of a problem, e.g., the server is not available.

The request gets sent to a specific data provider on the server (see orange arrow in Figure 5.4) by a *HTTP post request* (line 8). By definition this type of request does not restrict the length of its payload [7]. Therefore it is suitable to transfer requests without limiting the amount of item references or parameter.

```

1 angular.module('WorksheetModule', ['NgInfovisBase', 'NgHeatmap', ...])
2   .controller('WorksheetController', function ($scope, dataProvider,
3     dataTypes) {
4     // declare scope variable for the focus dataset
5     $scope.focusDataset = {};
6
7     // watch the item list scope variable of the base module
8     $scope.$watch('bridge.itemList', function (newVal, oldVal) {
9
10      // prepare request
11      var rqxObject = prepareRQxObject();
12
13      // initiate a data provider request
14      dataProvider("DataProviderID", rqxObject,
15        function (ServerResponseData) {
16          // load server response to scope variable
17          $scope.focusDataset = ServerResponseData;
18        },
19        function (error) {
20          // error handling
21        }
22      );
23    }, true);
24  });

```

Listing 5.8: This example of a worksheet implementation shows how a data provider request is initiated. A change of the 'itemList' variable triggers the request. The response is saved in the 'focus dataset' scope variable.

The scope variable of the focus dataset is assigned to the visualization directive by a parameter in the HTML code of the worksheet. Hereby it is essential for the function of the binding to use the same controller (see 'WorksheetController' in Listing 5.8 and Listing 5.9).

```

1 <body ng-controller="WorksheetController">
2 <heatmap table='focusDataset' selection='bridge.selectedItems'></heatmap>
3 </body>

```

Listing 5.9: The table attribute in line 2 connects the scope variable 'focusDataset' to the heatmap. This is how data is loaded into a view. It is essential for the functionality that both the HTML markup and the worksheet module declare the same controller. The selection attribute will be introduced in Section 5.3.

Workspace Connection

This section describes how user interactions from the application or the workspace are transmitted to worksheets. As can be seen in Listing 5.8 (first line) the declaration of a module may contain dependencies to other modules. As a result scope variables can be passed via HTML parameters to directives (see Listing 5.9).

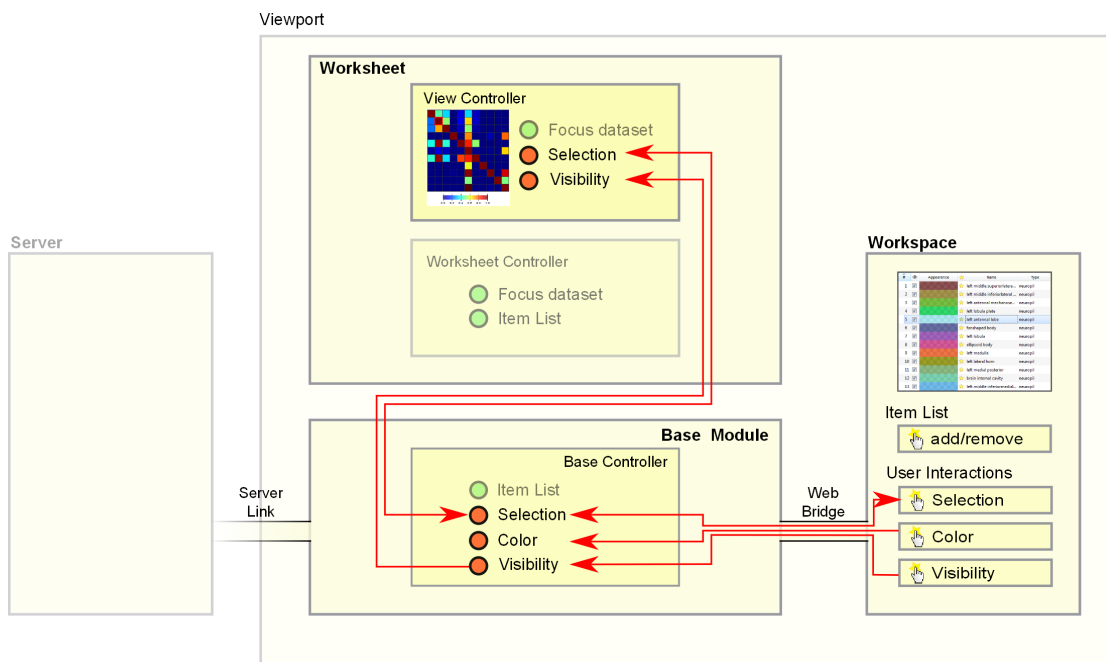


Figure 5.5: If the user makes changes in the workspace the scope variables of the base module are changed via the web bridge (red arrows). The scope variables for the color and the visibility can only be changed from the workspace whereas the selection can be changed from the worksheet as well. The worksheet scope variables may watch these variables for changes to adapt the view.

Qt WebKit Bridge

As stated in the previous section the base module declares scope variables which are bound to application signals via the so called Qt WebKit Bridge. This class enables the JavaScript environment of worksheet websites to access application functions and vice versa. The Qt WebKit Bridge instance creates a 'bridge object' for the JavaScript environment. This JSON object contains a set of functions and slots. Functions can be called to send data to the application. Slots can be connected by the developer with JavaScript functions to allow the application to execute JavaScript code (see Figure 5.6).

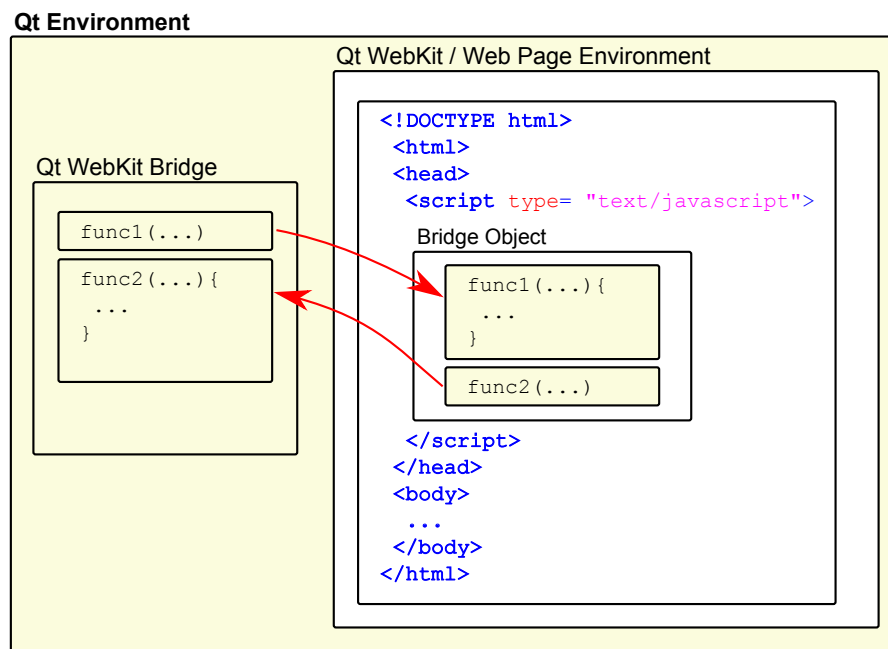


Figure 5.6: The bridge object enables a communication between the web page environment of the website and the parent application. If a JavaScript function is connected to a slot of the Bridge Object the function can be called from the Qt environment (red arrow). Functions of the Bridge Object can be called from the

The data that needs to be exchanged are JSON strings containing sets of item references with additional parameter, e.g. color or visibility settings. These strings have to be created and interpreted on both sides.

Implementation of the Qt WebKit Bridge

The Qt WebKit declares a 'bridge object' in the JavaScript environment of the displayed web page. The slots of this object are connected to application functions. They are triggered when the user changes the item list of the viewport (see Section 4.3) or an item properties (see Sec-

tion 4.6). The base module implements JavaScript functions and connects them to these slots (see Listing 5.10, line 2).

```
1 if (typeof bridgeObject.selectItems !== "undefined") {
2   bridgeObject.selectItems.connect(function (itemReferenceList) {
3     $rootScope.$apply(function () {
4       bridge.selectedItems = $.parseJSON(itemReferenceList);
5     });
6   });
7 }
```

Listing 5.10: This code snippet connects the slot 'selectedItems' of the bridge object to a local function which copies the the item list to the scope variable 'selectedItems'. As a result selections from the application are sent to the visualization.

The functions of the bridge object (see Listing 5.11) can be utilized to send signals from the worksheet to the application.

```
1 $rootScope.$watch('bridge.selectedItems', function (newVal, oldVal) {
2   if (typeof bridgeObject.setSelection !== "undefined") {
3     var items = bridge.selectedItems;
4     bridgeObject.setSelection(JSON.stringify(items));
5   }
6 });
```

Listing 5.11: This code snippet executes a function of the bridge object and passes a list of item references to it. As a consequence the application receives the list and is able to change the selection in the application.

Bridge Object Features

Table 5.2 introduces the implemented slots and functions of the implemented bridge object. They are by the application to communicate or transfer data between the visualization worksheet and the application. The parameter of these functions and slots are used to transfer item references (identification number and type) and workspace properties like appearance and visibility.

Function	Description
<pre>itemsChanged(added, removed) added = [{ type: "1", id: "1234", appearance: {r:255, g:255, b:255, a:255}, visibility: true }, ...] removed = [{ type: "1", id: "1234" }, ...]</pre>	<p>Add or remove data from the visualization when the user drags objects from the workspace into the viewport. 'added' is an array of JSON objects containing item identifiers and properties.</p>
<pre>itemAppearanceChanged(states) states = [{ type: "1", id: "1234", appearance: {r:255, g:255, b:255, a:255} }, ...]</pre>	<p>Changes the appearance of the specified objects when the user adjusts settings (color and transparency) of one or more objects in the workspace. 'states' is an array of JSON objects containing item references and their appearance properties as an array of color and alpha values.</p>
<pre>itemVisibilityChanged(states) states = [{ type: "1", id: "1234", visibility: true }, ...]</pre>	<p>Changes the appearance and the visibility of the specified objects. 'states' is an array of JSON objects containing items and their visibility properties as boolean values.</p>

Table 5.2: This table introduces the implemented slots of the bridge object as well as the parameter or return values. To use slots they have to be connected to local JavaScript functions (see Listing 5.10).

Workspace features can be triggered by calling the JavaScript function from the bridge object. By adding parameters to this functions data can be exchanged between the two sides. Table 5.3 introduces the functions of the bridge object. These functions can be called from the JavaScript environment of the web page.

<pre>setSelection(selection) selection = [{ type: 1, id: "1234" }, ...]</pre>	Sets the selection in the workspace when the user selects objects in the visualization. 'selection' is an array of JSON objects containing item identifiers.
<pre>addItem(newItems) newItems = [{ type: 1, id: "1234", }, ...]</pre>	Adds items from the visualization to the worksheet. Can be used when the focus dataset contains references to related items. 'newItems' is an array of JSON objects containing item identifier.

Table 5.3: This table introduces the implemented functions of the bridge object as well as the parameter. These functions can be directly called via the bridge object.

Interpretation as Model View Controller Cascade

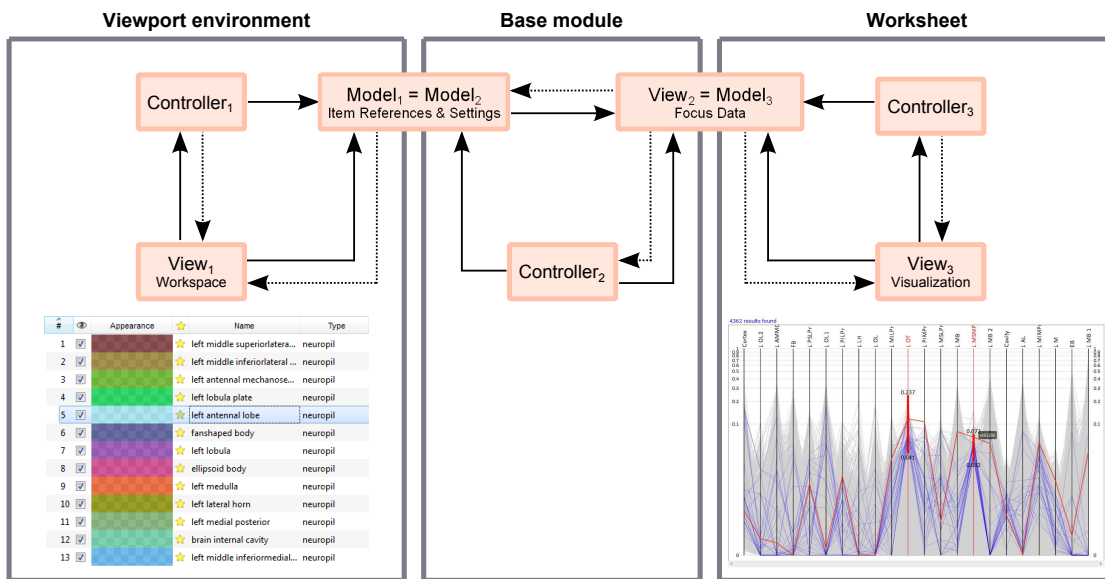


Figure 5.7: The usage of at least three different controllers establishes multiple model-view-controller cycles (see Figure 2.8).

Due to the implementation of at least three controllers per worksheet multiple model-view-controller cycles are realized (see Figure 5.7).

The first cycle is integrated in the viewport environment. Its model contains item references and settings. The application presents this list as workspace to the user. This list offers features

to adjust both the list items and various settings for each item. When the user changes these settings the controller updates the model from the viewport. This model is shared between the viewport and the worksheet, i.e. the base module.

Via the bridge object the item list is shared with the worksheet. The worksheet controller 'watches' this list for changes to request a new focus dataset. A change of this dataset triggers the view to update the visualization(s).

Due to the ability of worksheets to add items to the workspace the sequence can be reversed. If the user decides to add a new item from the worksheet to the workspace the shared model gets updated. This change triggers the viewport to display the new item and the base module to acquire a new focus dataset.

Evaluation

6.1 Objective Evaluation

This section describes the performance of the proposed concept by presenting latency measurements of the pipeline. The purpose of this performance evaluation is to measure the efficiency of the pipeline components. After these measurements it can be evaluated if the concept performs sufficiently in the existing environment by determining the efficiency of each individual step.

Evaluation Strategy

The evaluation was conducted with a fixed set of 20 workspace items. The references of these items were hard coded into the source code of the base module and were included in the request. This results in a JavaScript object of 212 bytes which was sent to the server and resulted in an focus dataset of 2960 bytes. The size of the JavaScript objects was determined by introducing *sizeof.js* [11].

Latencies are measured by logging time stamps at several places between the involved components (see Figure 6.1). The differences between these time stamps allow to determine latencies, e.g., the processing time of a data provider. Therefore log messages were added to the source code of both the worksheet and the server components to return the system time at different stages of the pipeline. These messages log the system time in milliseconds as integer. Therefore it's possible to determine the latencies by simply subtracting consecutive values. The rendering step of the pipeline is not considered in this evaluation because it strongly depends on the type of the used visualization. This section concentrates on the delivery of the focus data.

Time measurement points where located before the following positions:

- T_0 : Initializing the data provider request (worksheet)
- T_1 : Arrival of the request at the server, before dispatching (server)

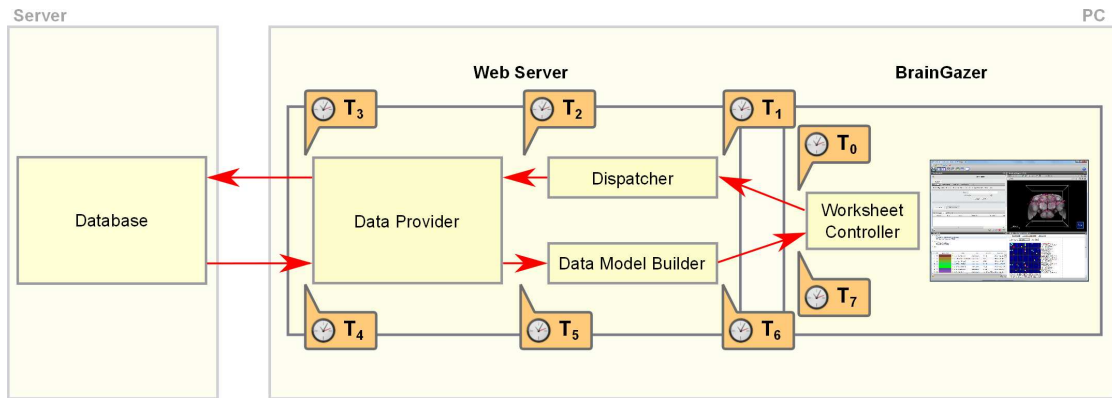


Figure 6.1: This illustration points out the places in the pipeline where time measurements are made. A request is initiated in the worksheet controller at T_0 and ends at T_7 .

- T_2 : start of data provider (server)
- T_3 : data base request (server)
- T_4 : merging result sets from different sources (server)
- T_5 : data model builder (server)
- T_6 : returning data model back to worksheet (server)
- T_7 : data provider response (worksheet)

After placing this measurement points 20 runs where performed. The time spans between the logs can be interpreted as the time consumption of the processing steps.

Infrastructure

The measurements were performed on a PC running Windows 7 (64 Bit) with 16 GB RAM and a 3.4 GHz CPU with four cores¹. The web server components were running on the same PC for the evaluation. Only the database is located on a different machine running Ubuntu 11.04 (32 bit) with 2 GB RAM and a 3 GHz processor². In the evaluation environment the PC and the database server both are connected by GBit Ethernet in a local area network. A ping test revealed a latency of less than 1 ms.

Results

As can be seen the longest time is spent between initializing a request and receiving it on the server (176 ms). The time spent for dispatching the request is lesser than 1 ms. This time cannot

¹Intel Core i7-2600 @ 3.4 GHz

²Intel Pentium 4 @ 3 GHz

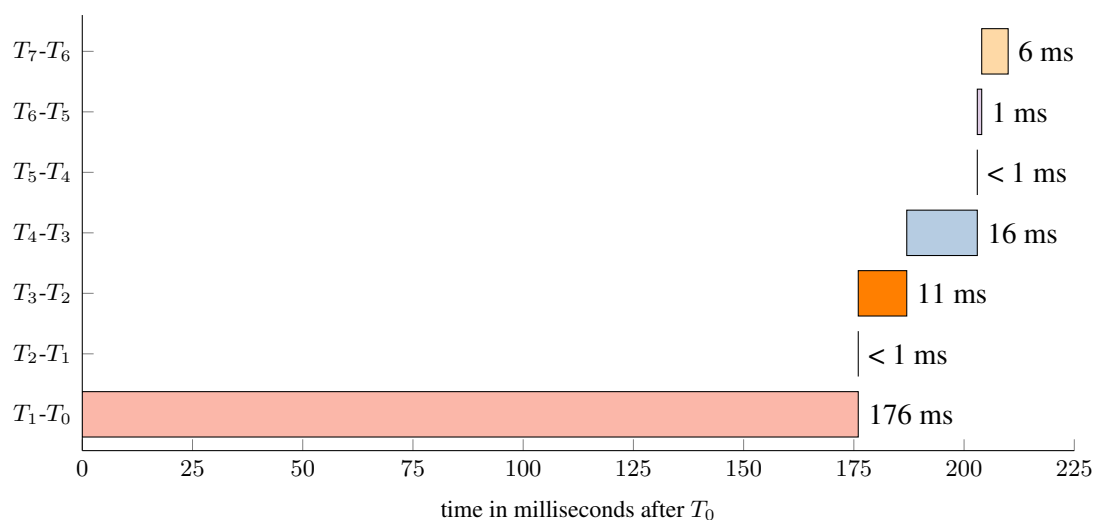


Figure 6.2: An illustration of the processing delays caused by the implemented components. These are averaged values after 20 measurements.

be measured exactly because the resolution of 1 ms does not allow an accurate measurement. More time is required for creating a database request (11 ms) and waiting for the response (16 ms). The time to process the response as well as converting the response to a data model takes less than 1 ms. Sending back the reply to BrainGazer takes significantly longer.

Discussion

As can be seen in Figure 6.1 the longest delay occurs between initializing a data provider request and receiving it on the web server. This delay might be the result of a bad JavaScript processing performance of BrainGazer. As stated in Section 5.1 the Qt WebKit is used to embed the websites. To prove this assumption a benchmark comparison between the embedded web browser and a desktop browser was conducted. It revealed a significant lack of performance of the embedded browser (see Table 6.1).

Benchmark Website	Firefox 31.0	Qt WebKit	Difference [%]
browsermark.rightware.com	4238	140	97%
peacekeeper.futuremark.com	1159	707	39%
www.speed-battle.de	1172	78	93%

Table 6.1: Benchmark results comparison between the embedded web browser of BrainGazer (Qt WebKit, see Section 5.1) and a standard desktop browser on the same PC (for specifications see Section 6.1). The numbers are the results of website benchmark tests. A higher value indicates a better performance.

To strengthen the hypothesis of the bad performing Qt WebKit the same measurements

where performed by a standard web browser (see Browser statistics [14] for a list of standard web browsers). In this scenario the measurements revealed latencies between T_0 to T_1 of only 15 ms. As a result it can be stated that the delays are caused by the embedded Qt WebKit.

It can be seen that dispatching of the request is performed very quickly (T_2-T_1). Due to the small number of data providers (9) in the map data structure the dispatching can be finished very fast.

To create the database query the request object needs to be processed. This depends on the size of the request and causes a small delay of 11 ms. Consequently the query needs to be executed (T_4-T_3).

In this evaluation use case a post processing of the result set is not required. Therefore no time is spent for this step (T_5-T_4). This delay might be significantly higher when multiple data sources are involved. This would require a merging procedure of the different data sets.

Creating the data model requires to convert the result set into a different data structure (T_6-T_5). Therefore the complete dataset needs to be rebuilt. It can be said that the duration of this procedure depends on the size of the dataset.

Back on the client side the performance lack of the Qt WebKit occurs again. After receiving the response the dataset gets copied into a scope variable. Unfortunately the bad performance of the Qt WebKit in comparison to a standard web browser has a significant impact on the proposed concept. As a result it can be said that the decision to introduce web technologies on the client side (see Section 4.5) simplified the implementation of worksheets but led to a bad usability in the existing environment.

6.2 Case studies

This section will describe two use cases and how the proposed concept can be used to solve the given problems. The first section will introduce the biomedical background that is necessary to describe the use cases.

Biomedical Background

The aim of circuit neuroscience is to understand the computational function of neuronal circuits. Scientists try to reverse-engineer biological circuits to gain knowledge about their structure and logic to understand their computational algorithms [46].

An often used modeling organism is the *Drosophila Melanogaster*, the commonly known fruit fly. Due to the high degree of stereotypy in insect nervous systems it has become an important modeling organism in circuit neuroscience in order to create brain atlases [20]. There is a direct homology between *Drosophila* genes and genes that affect human diseases. A significant proportion of the known human genes are similar to *Drosophila* genes. On the other side a large number of *Drosophila* protein sequences has similarities to those of mammals [33]. Thus, knowledge gained from *Drosophila* brain studies might be helpful to form hypotheses about the human brain.

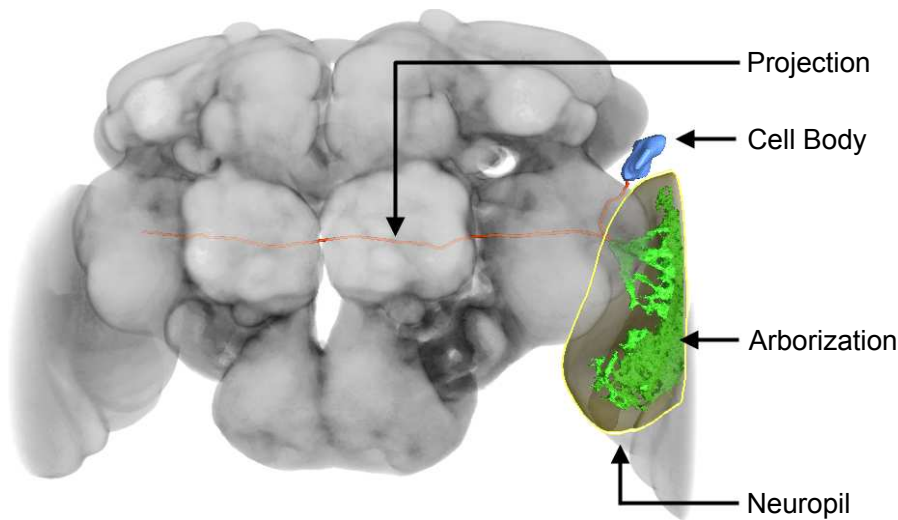


Figure 6.3: Segmented anatomical structures in the context of a *Drosophila Melanogaster* brain [42]

A *neuron* consists of a composition of cell body, projections and any number of arborizations (see Figure 6.3). The cell bodies are blob-shaped structures containing the nucleus of the cell. *Arborizations* are tree like terminal branches that connects the neuron to other neurons to enable a communication via synapses. The connections between cell bodies and arborizations are called *projections*. Additionally to this object classification the brain and the associated ventral nerve cord (VNC) is divided into functional or spatial sub regions of the nervous system called *neuropils* [42].

Data Acquisition

To study the nervous system of the *Drosophila Melanogaster* scientists have acquired a large number of confocal microscopic image stacks of individual *Drosophila* brains [3]. They are using molecular genetic techniques [19] to highlight specific neurons in the samples. This technique results in a fluorescence effect which makes the targeted neurons stand out from the remaining tissue. The volumetric images of brain tissue and the highlighted structures can be generated by registering the scans from the first channel to a standard brain template. This template is a standard image generated by averaging a set of representative scans to a reference scan. The neurons are visible on the second channel of the microscope. The final image is the result of the two channels combined with the template.

By creating binary masks and geometry files the scientists point out interesting structures and store the segmented neurons separately. The generated mask and geometry files are saved on a file server whereas both references to images and their neuronal structures are stored in a relational database [42].

Then the generated files run through a series of preprocessing steps to detect overlapping structures. The generated percentage values indicate how much two individual structures overlap each other. These percentage values are stored along with references to both objects in a specifically prepared database table [20]. As a result overlapping or staining values can easily be accessed for further analysis.

Use Case: Image Staining per Neuropil

Finding similar anatomical structures in a set of individual images is an often required step in the workflow of an experiment. This is how the scientists are able to find out different shapes of structures in genetically identical groups (lines).

Visualization Approach

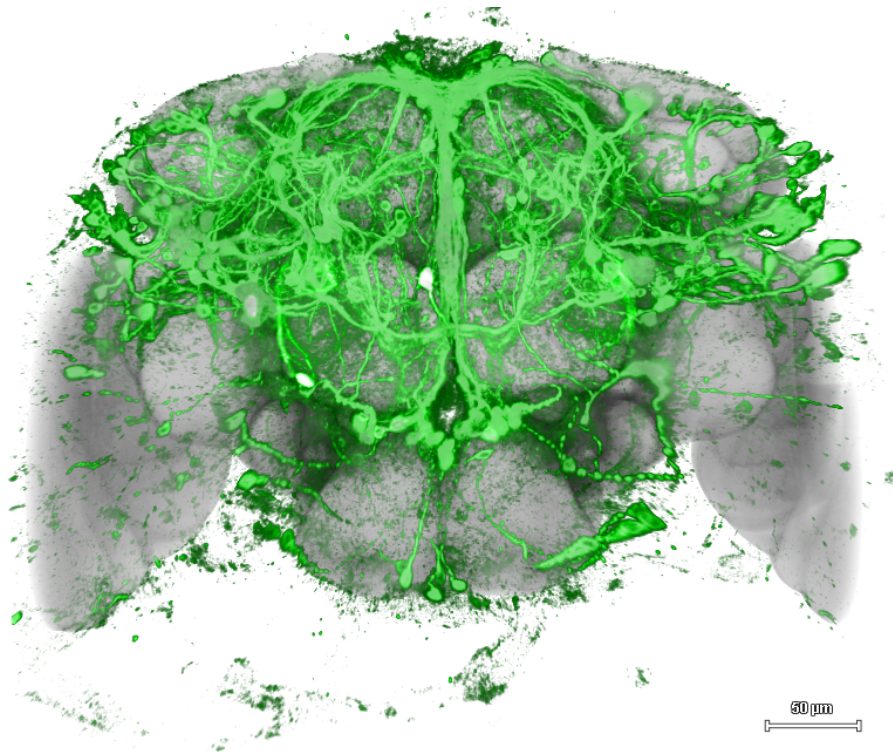


Figure 6.4: 3D rendering of a volumetric image data of a *Drosophila Melanogaster* brain. The green channel shows the fluorescence staining of neurons. The gray channel represents a standard brain template.

The number of highlighted (stained) voxels in a certain region of the brain is an important value to compare a set of images concerning similar neuronal structures (see Figure 6.4). High staining values in a certain area might indicate the presence of neurons. During the preprocessing of the images the number of stained voxels is counted and stored as an integer in the

database. These values can easily be used to introduce a visual comparison method.

All the values are stored in a single database table with references to images and neuropils. Hence the content of this data table can be categorized as multivariate data and the involved object types define the input data types of the visualization.

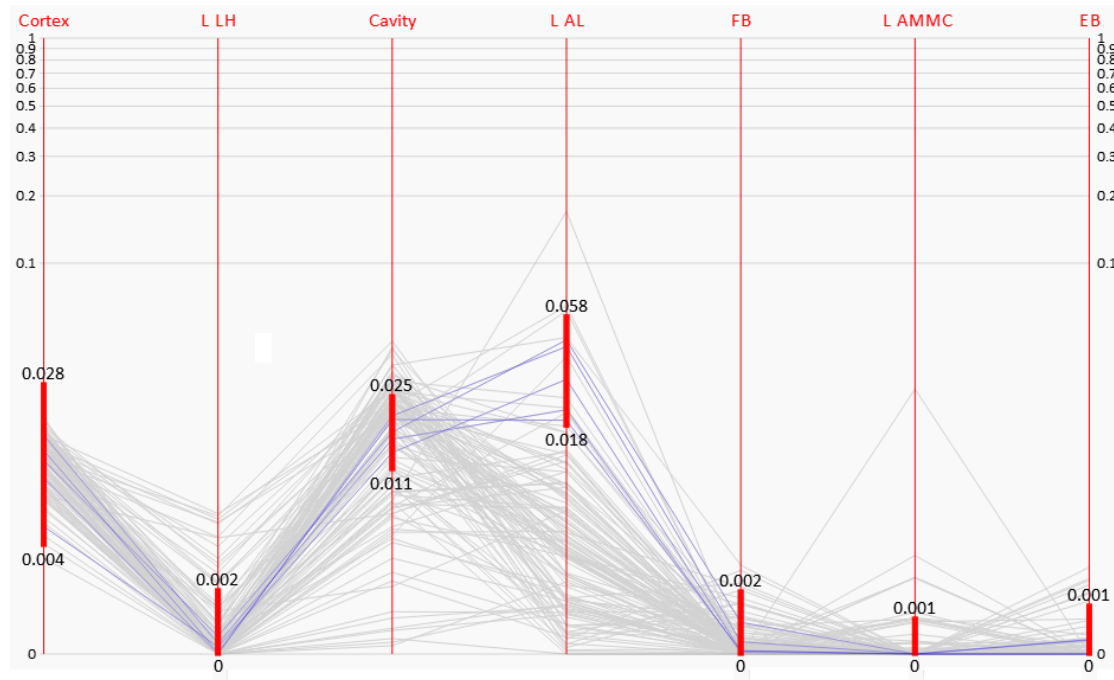


Figure 6.5: A screenshot of an interactive parallel coordinate system as implemented in Listing 6.1. The thick red lines on the vertical axes represent filter selections to allow the user to pick only samples which fit into a certain range (blue lines). The filtered data can then be accessed by a scope variable.

Parallel coordinate systems are a widely used visualization method for multivariate data and have become well-known for exploratory data analysis [30]. They can be implemented with interaction features, e.g. brushing to allow the user to select a certain subset of samples. A selection technique in an interactive visualization is essential for allowing the user to pick samples which fit into a certain range.

To visualize the image staining the vertical axes of the parallel coordinate system can be used to represent the neuropils. The images are then represented by lines between the axes which connect the staining values of each individual neuropil. Figure 6.5 is an example of the visualization showing the staining values of a set of images (lines) on seven neuropils (vertical axes). In this example the user has set a filter on each axes to pick a subset in order to find similar images.

Before the investigation starts the user has to select both a set of images and neuropils by using the database search interface. From there the results are added to the workspace and in the next step to the novel infovis viewport. After dropping the items the worksheet loads the needed data for the visualization and renders the parallel coordinate system. If this worksheet is used in combination with a 3D viewport, the user is able to visually compare the neuronal structures of the results of the worksheet investigation. Therefore the user loads the volumetric images of possible candidates into a 3D viewport and compares the areas of interest by overlaying the segmented neuronal structures to the images.

Implementation of the Visualization Worksheet

The visualization instance can be embedded as AngularJS directive which is used within the HTML code of the worksheet website (see Section 5). Through the attributes of this directive data can be exchanged with the view. In the developing stage of the worksheet, the directive of the parallel coordinate system has to be connected with one or more data provider and the application by using scope variables of the server and the workspace connection (see Section 5.3).

The following listings are representative code segments from the implementation to describe the proposed concept.

```
1 <body data-ng-controller="AppController">
2   <div id="viscontainer">
3     <parallelcoordinates
4       table="imageNeuropilStaining"
5       dataselection="bridge.selectedItems"
6       axisselection="bridge.selectedItems"
7       filtereddata="bridge.filteredItems">
8     </parallelcoordinates>
9
10    <itemstoworkspacebutton
11      items="bridge.selectedItems">
12    </itemstoworkspacebutton>
13  </div>
14
15  <loadinganimation status="bridge.isLoading"></loadinganimation>
16
17  <script type="text/javascript">
18    // data provider definition, see below
19  </script>
20 </body>
```

Listing 6.1: The HTML skeleton of the parallel coordinates visualization.

Listing 6.1 is the HTML skeleton necessary for a single visualization and its integration into the application. The tag `<parallelcoordinates ...>` in the third line is a user defined AngularJS directive that introduces the plot. Its attributes link the plot to the scope variables of the Qt WebKit Bridge and the data provider without the usage of additional event handler functions. This is how the linking mechanism of BrainGazer gets extended to the visualization. The following list introduces the scope variables of Listing 6.1 necessary to connect the parallel coordinate system to both the server and the application (see Figure 5.3):

- *table*: The input data for the plot (focus data) which results directly from the data provider response (see Listing 6.2).
- *dataselection*: A set of identifiers representing the selected data lines of the plot.
- *axisselection*: A set of identifiers representing the selected axis of the plot.
- *filtereddata*: A set of identifiers representing filtered data lines.

The *table* attribute is assigned to *imageNeuropilStaining* a scope variable which is filled by the data provider (see Listing 6.2). data selection and axis selection both are lists of identifiers representing selected axis or lines. These two attributes can be used bidirectionally in order to propagate selected items from the visualization to the application and vice versa.

The code segment of Listing 6.2 is the definition of the server connection to connect the visualization to the corresponding data provider on the server. First, the worksheet dependent scope variable '\$scope.imageNeuropilStaining' is declared which contains the focus dataset after a request.

```

1 <script type="text/javascript">
2 angular.module('NgImageStaining', ['NgBase', 'NgParallelCoordinates'])
3   .controller('AppController', function ($scope, dataProvider, dataTypes) {
4
5     $scope.imageNeuropilStaining = {}; // scope variable declaration
6
7     $scope.$watch('bridge.itemList', function (newItems, oldItems) {
8
9       var requestObject = {...}; // initialize and prepare the request object
10
11       $scope.bridge.isloading = true; // show loading animation
12
13       // data provider definition
14       dataProvider("imageNeuropilInstanceStaining", requestObject,
15         function (table) { // successfully received server response
16           $scope.imageNeuropilStaining = table; // visualization data
17           $scope.bridge.isloading = false; // hide loading animation
18         },
19         function (error) {...} // error handling
20       );
21     });
22 });
23 </script>

```

Listing 6.2: Controller definition and data provider functionality

The watch statement (Listing 6.2, line 7) gets triggered every time the list of associated items of the viewport is changed. Hence, if the set of items changes, a data provider request is sent to the server and updates '\$scope.imageNeuropilStaining' in order to update the visualization.

The scope variable 'imageNeuropilStaining' is connected to the view controller (see Listing 6.2 line 5). Therefore it can be watched by the view controller to re-render the visualization in case of a data update (line 14).

```

1 angular.module("NgParallelCoordinates", [])
2 .directive('parallelcoordinates', function () {
3     return {
4         restrict: 'E', // directive option
5         scope: {      // scope variables = HTML parameter
6             table: '=', // enriched dataset
7             axisselection: '=?' // selection
8         },
9         template: "<div id='parallel-coordinates'></div>",
10
11        link: function (scope, element, attrs) {
12            var pcl = new ParallelCoordinates('parallel-coordinates');
13
14            scope.$watch('table', function (newVal, oldVal) {
15                pcl.clearPC(); // reset view
16                pcl.update(newVal.data, newVal.rows, newVal.columns); // update view
17            });
18
19            // user select items in the workspace
20            scope.$watch('axisselection', function (newVal, oldVal) {
21                if (pcSelection.axis !== undefined)
22                    pcl.highlightItems(newVal);
23            }, true);
24
25            // user selects items in the view
26            pcl.AxisSelectionChanged = function (selection) {
27                scope.axisselection = _.map(selection, function (d) {
28                    return { "id": d.id, "type": d.type };
29                });
30            };
31        }
32    };
33 });

```

Listing 6.3: This simplified listing describes how a view is 'wired' with the base module. The watch statements detect changes of the base modules scope variables and perform changes to the view. If the user selects items in the view a scope variable of the base module is changed to trigger the watch statements in the base module.

The HTML and JavaScript source code of both Listings 6.2 and 6.3 enable a fully functional visualization. Due to the design modules responsible for the visualization and the bridge can be reused without modifications in other worksheets. The next use case demonstrates the usage of multiple visualizations in one worksheet.

Use Case: Arborization Overlap

As mentioned in Section 6.2 the connectivity of neurons plays an essential role in understanding how information is transmitted and processed in the brain. Hence, neuroscientists try to gain more knowledge about the wiring of neurons.

A necessary but not sufficient condition for the existence of a connection is a spatial overlap between the arborizations of neurons (Neuromap [42]) (see Figure 6.6). At first the user estab-

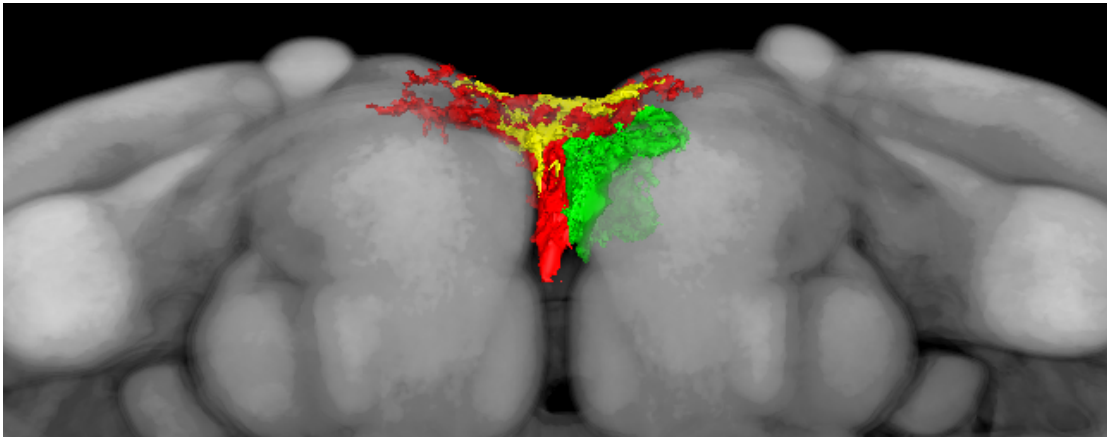


Figure 6.6: 3D rendering of three overlapping arborizations (red, green, yellow). An overlapping indicates that these structures might be synaptically connected at certain regions in order to transmit information.

lishes a set of possible arborizations in the workspace and associates it to the InfoVis viewport.

A data visualization of the overlap values can be helpful as first step of a workflow although the spatial context of overlapping regions might get lost in a 2 dimensional visualization. But if the visualization is used in combination with a 3D view the spatial context can be reestablished by the linking mechanism. This allows the user to visually observe the overlapping regions in the 3D view and to determine if the overlap indicates a synaptic connection.

Visualization Approach

In a processing step of the segmentation procedure overlapping values are computed automatically. Along with references to arborizations they get stored in the database. As a result a record contains two references to arborizations and an overlap value. If using a set of arborizations as input, a matrix of overlapping values can be put together.

For the illustration of matrices multiple visualization types are suitable. A very common method is to use a heatmap (see Figure 2.3). Hereby color scales are used to represent numerical values of a certain range. Each data value is mapped to a certain color according to the scale.

Another method is to use a chord diagram (see Figure 6.7) to display the data values. It displays the data as segments of a circle and displays the relations by connecting pairs of segments.

As a result two different visualization types are suitable to support the user at dealing with this use case. This requires methods to display multiple views and mechanisms to switch between them. In the context of this use case a heatmap is better suited to find out pairs of overlapping structures and enables to visually compare overlapping values. Both visualization types have advantages and disadvantages. A chord diagram is better suitable to answer the question

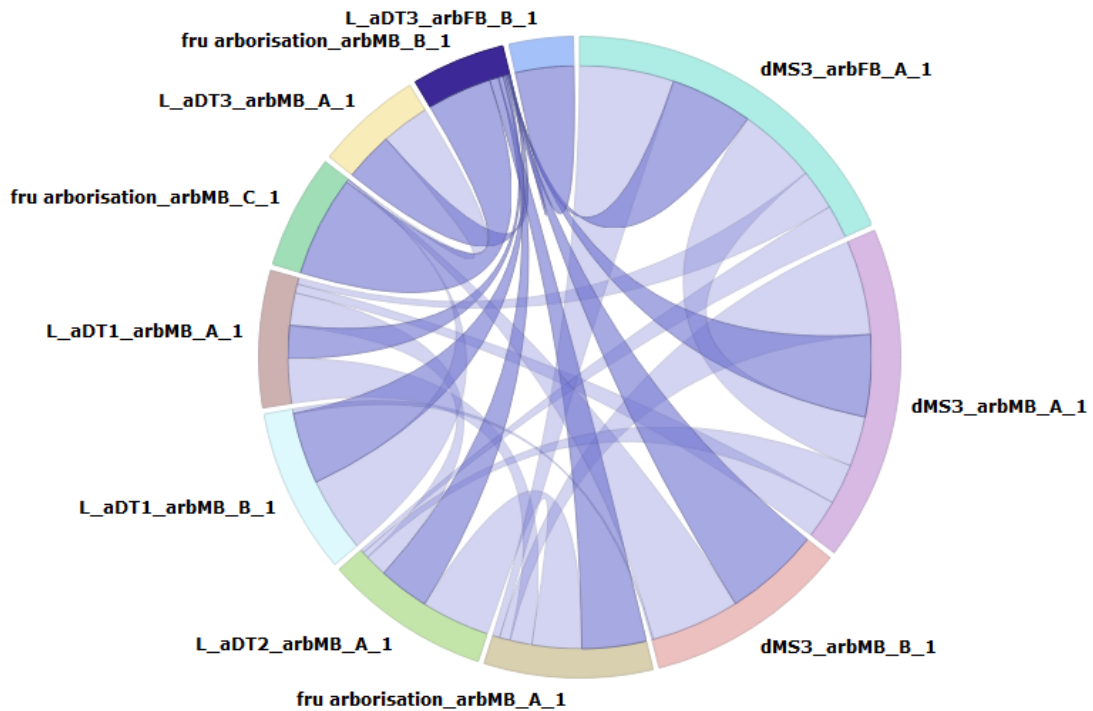


Figure 6.7: A chord diagram representing the overlapping between a set of arborizations. A connection of two segments represents an overlapping. By highlighting individual segment and the connections the user immediately sees if the selected arborization overlaps other arborizations.

whether a single arborization overlaps certain other structures. Therefore in a chord diagram multiple segments or connections can be selected to find out the overlaps among the selected arborizations.

The user starts the investigation by setting up a collection of relevant arborizations in the workspace. After associating these arborizations to the InfoVis viewport the two visualizations are rendered (see Figure 7.1).

Implementation of the Visualization Worksheet

```

1 <div class="tab-container">
2 <div id="tab1" name="Heatmap" class="tab active">
3 <heatmap
4   table='overlapTable'
5   selection='bridge.selectedItems'>
6 </heatmap>
7 </div>
8 <div id="tab2" name="Chord Diagramm" class="tab">
9 <chorddiagramm

```

```
10 table='overlapTable'  
11 selection='bridge.selectedItems'  
12 appearance='bridge.itemList'>  
13 </chorddiagram>  
14 </div>  
15 </div>
```

Listing 6.4: The HTML skeleton of a multi-view worksheet. Both visualizations use the same scope variables.

Listing 6.4 uses two visualization directives. Both directives are using the same scope variables for the enriched dataset (*overlapTable*) and the selection. The heatmap does not support appearance settings because it uses colors to represent values. Therefore the appearance is only connected to the chord diagram. To enable the switching mechanism the two directives are enclosed by *div* elements and assigned to CSS properties³. The switching mechanism adjusts these CSS parameters if the user decides to change the view. As a result the views can be displayed separately or simultaneously.

Figure 7.1 shows the result of the implementation in BrainGazer. The user can choose between two visualization types. For this use case the heatmap is better suitable to find out the overlapping between two arborizations whereas the chord diagram can better be used to find out if two or more arborizations have overlapping regions.

³Cascading Style Sheets, style sheet language to define visual features of web pages.

Conclusion and Future Work

7.1 Conclusion

This thesis proposes a visualization framework that aims at extending an existing scientific visualization environment by additional two dimensional data visualizations (see Figure 7.1). It describes the technical background of the implementation starting from the database backend and ends with the integration on the client-side application.

This software design concept aims at integrating new views into an existing environment and at hiding the infrastructure from the user. Due to the workspace integration users are able to interact with the InfoVis viewport like any other viewport.

7.2 Future work

This chapter describes possible improvements to the proposed concept. So far this proposed software design concept is implemented to enable new possibilities to compare three dimensional object representations by using two dimensional information visualizations. The main focus of the implementation was to create a prototype which is, as the objective evaluation shows (see Section 6.1), far away from being perfect. The following illustration of the data flow illustrates possible bottlenecks in the pipeline structure. Potential improvements are described in the next sub sections.

Figure 7.2 illustrates the basic architecture of the proposed pipeline concept. These components and the connections between them are potential candidates for improvements. Of course the performance of the system could be improved by better hardware components but the following sub sections focus on implementation details to improve conceptual issues as well as to eliminate performance bottlenecks.

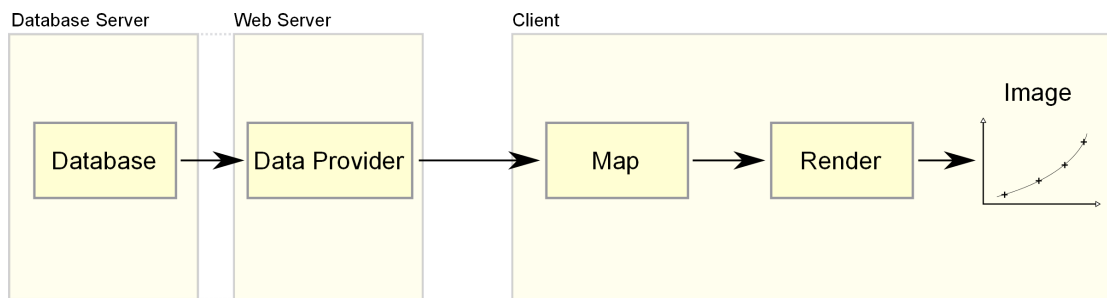


Figure 7.2: The main components and connections of the pipeline between the database and the visualization.

Server-side caching

The web server is responsible for performing database queries. This is done by sending a query string (i.e., an SQL statement) to the database server. This query contains a set of use case specific parameters to get exactly the wanted datasets. This set may contain object identifiers or commands to collect data from different database tables. As a result more complex queries might lead to a higher workload of the database.

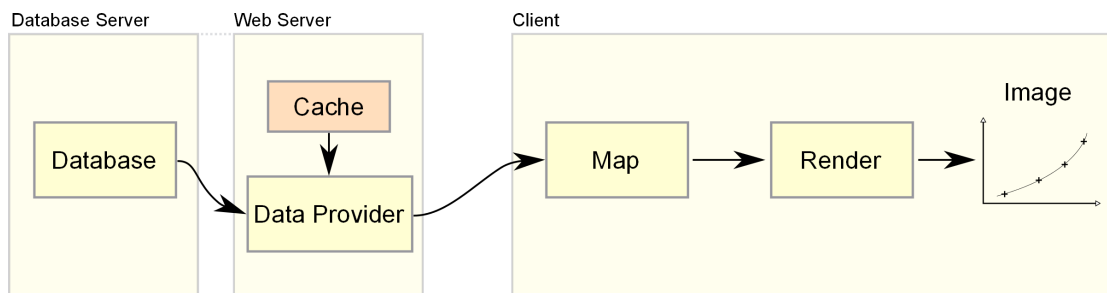


Figure 7.3: The data flow with a cache on the web server. If the client sends a request the server checks if the cache contains the results. Only if the cache does not contain a valid entry (cache miss) a database query is performed.

An approach to this problem would be a server-side caching mechanism that temporarily stores query results (see Figure 7.3). If a client requests data, the results are stored in the cache and sent back to the client. If the web server receives the same request again it can easily respond to the request with the cached data. Consequently this would lead to a load removal at the database server. On the other hand this feature requires caching techniques and additional memory. A caching system would be necessary to manage the cache entries. A drawback of this solution would be a heavier workload of the web server.

Data compression

'JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate' [13]. An object consists of a set of key/value pairs. Especially JSON object arrays of the same type contain the same names for each array element. This leads to overhead which might cause a longer server response time.

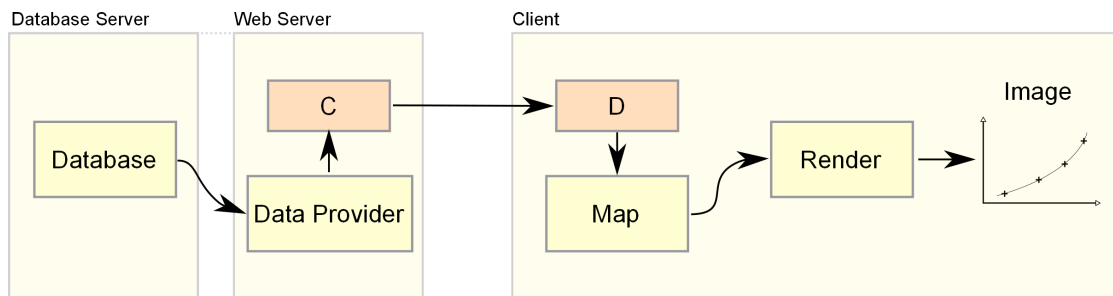


Figure 7.4: The data flow with a compression mechanism on the web server and on the client side. The compression function (C) on the web server compresses the JSON object and the client decompresses it (D).

A compression mechanism on the web server reduces the size of data that has to be transferred (see Figure 7.4). This would lead to a faster transmission but requires additional processing power on the web server and on the clients machines. A simple way to achieve this compression solution is to use a common built-in compression technique of web servers like GZIP. It's an often used algorithm which is supported by all major web browsers and servers [6]. Another compression technique can be introduced for the worksheets like HTML and JavaScript files. This method reduces the file size and improves the loading time of the website in the browser.

Incremental Updates

So far, every time the user adds a new instance item to a viewport the entire dataset of the visualization is updated. Valid data that already exist on the client side is deleted and is replaced by the same information plus the data from the new instances. This problem does not only concern the transmission of the data. The database query and the preprocessing of the enriched dataset require calculation efforts for the web server.

The solution of the problem would be the introduction of incremental updates. This means that only the visualization data is required from the server that is related to the newly added instance items of the viewport. The server sends the partial update and the client merges it with the already existing visualization data.

Improve Implementation of Data Provider

The server components of this software design concept are implemented as Java classes by the use of the Google Web Toolkit [12]. Therefore any part which is implemented within the Java environment requires a full re-compilation of the web server. As a consequence the user cannot extend the server by new data providers without the knowledge about the implementation.

To solve this problem a more modular approach is needed. A more dynamic way of adding would be an additional API to allow a dynamic integration of new components to the server. An alternative solution to this problem might be a more generalized data provider concept as described in the next section.

Generalized data provider

Another way of expanding the functionality is to reduce the complexity of data providers on the server side. In the proposed pipeline concept database queries are defined on the server side. On arrival of a client request the query is extended by item references and is executed. A more generalized approach would be to define the queries on the client-side. This way, only a single data provider would be necessary. The dispatching step would be obsolete. On the other hand the client becomes responsible for a part of the filtering. The client needs to transform the results to the standardized data model (see Section 4.5). This would change fundamental parts of the proposed concept (see Section 4.4). The implementation of a single generalized data provider would lead to a relief of the web server. On the other hand the data traffic would increase because the results of database queries are transmitted as raw data.

By acquiring contents from different data sources the client has to join the results to preserve the focus dataset. This enables a new more dynamic approach but leads to more complex visualization worksheets. Furthermore precautions must be considered in terms of database security.

Bibliography

- [1] AngularJS. <http://angularjs.org/>. Accessed: 2014-02-26.
- [2] Apache Tomcat. <http://tomcat.apache.org/>. Accessed: 2014-05-06.
- [3] BrainBase Database. <http://brainbase.imp.ac.at/bbweb>. Accessed: 2014-11-28.
- [4] Crossfilter - Fast Multidimensional Filtering for Coordinated Views. <http://square.github.io/crossfilter/>. Accessed: 2014-11-28.
- [5] google-gson - A Java library to convert JSON to Java objects and vice-versa . <https://code.google.com/p/google-gson/>. Accessed: 2014-02-18.
- [6] HTTP Compression. <http://www.http-compression.com/>. Accessed: 2014-02-18.
- [7] HTTP Methods: GET vs. POST. http://www.w3schools.com/tags/ref_httpmethods.asp/. Accessed: 2014-02-26.
- [8] Java. <http://www.oracle.com/technetwork/java/javase/downloads/index.html>. Accessed: 2014-07-31.
- [9] Qt Project. <http://qt-project.org>. Accessed: 2014-02-26.
- [10] Scatterplot of Old Faithful Geyser Eruptions. <http://commons.wikimedia.org/wiki/File:Oldfaithful3.png>. Accessed: 2014-07-21.
- [11] sizeof.js. <http://code.stephenmorley.org/javascript/finding-the-memory-usage-of-objects/>. Accessed: 2014-09-16.
- [12] The Google Web Toolkit Project. <http://www.gwtproject.org>. Accessed: 2014-02-26.
- [13] The JSON Standard. <http://www.json.org/>. Accessed: 2014-05-13.
- [14] w3schools.com - Browser Statistics. http://www.w3schools.com/browsers/browsers_stats.asp. Accessed: 2014-02-26.
- [15] Information Technology - Database Language SQL. <http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt>, 1992. Accessed: 2014-07-31.

- [16] A. Ahmed, K. Latiff, M. S. A .and Abu Bakar, and Z. A. Rajion. Visualization pipeline for medical datasets on grid computing environment. In *International Conference on Computational Science and its Applications 2007*, pages 567–576, Aug 2007.
- [17] L. Bavoil, S.P. Callahan, P.J. Crossno, J. Freire, C.E. Scheidegger, C.T. Silva, and H.T. Vo. Vistrails: enabling interactive multiple-view visualizations. In *Visualization, 2005. VIS 05. IEEE*, pages 135–142, Oct 2005.
- [18] B. B. Bederson and B. Shneiderman. *The craft of information visualization: readings and reflections*. M. Kaufmann, 2003.
- [19] A. H. Brand and N. Perrimon. Targeted gene expression as a means of altering cell fates and generating dominant phenotypes. *Development (Cambridge, England)*, 118(2):401–15, June 1993.
- [20] S. Bruckner, V. Soltészová, M. E. Gröller, J. Hladuvka, K. Bühler, J. Y. Yu, and B. J. Dickson. Braingazer - visual queries for neurobiology research. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1497–504, 2009.
- [21] E. H.-H. Chi. *A Framework for Information Visualization Spreadsheets*. Springer Science & Business Media, 1999.
- [22] M.C.F. de Oliveira and H. Levkowitz. From visual data exploration to visual data mining: a survey. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):378–394, July 2003.
- [23] J. Fekete. The infovis toolkit. In *Information Visualization, 2004. INFOVIS 2004. IEEE Symposium on*, pages 167–174, 2004.
- [24] M. Friendly. Milestones in the history of thematic cartography, statistical graphics, and data visualization. <http://www.datavis.ca/milestones/>, 1995. Accessed: 2014-11-28.
- [25] A. G. Gee, M. Yu, and G. G. Grinstein. Dynamic and interactive dimensional anchors for spring-based visualizations. *Technical Report 2005*, 2005.
- [26] R. B. Haber and D. A. McNabb. Visualization idioms: A conceptual model for scientific visualization systems. In *Visualization in Scientific Computing*, pages 74–93. IEEE Computer Society Press, 1990.
- [27] C. D. Hansen and C. R. Johnson. *The Visualization Handbook*. Elsevier Butterworth-Heinemann, 2005.
- [28] H. Hauser, D. Weiskopf, Kwan-Liu Ma, Jarke van Wijk, and Robert Kosara. Scivis, infovis - bridging the community divide?! *IEEE Visualization Conference Compendium*, pages 52–55, 2006.
- [29] J. Heer, S. K. Card, and J. A. Landay. Prefuse: A toolkit for interactive information visualization. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '05*, pages 421–430, New York, NY, USA, 2005. ACM.

- [30] J. Heinrich and D. Weiskopf. State of the Art of Parallel Coordinates. In *Proceedings of the Eurographic conference 2013*, pages 95–116. Eurographics Association, 2012.
- [31] R. Kosara. A little Space, Please - SciVis and InfoVis face off. *American Scientist*, Volume 97, Number 2:150, March-April 2009.
- [32] B Lahres and G. Rayman. *Praxisbuch Objektorientierung: Das umfassende Handbuch*. Galileo Press, Bonn, 2nd edition, 2009.
- [33] T. F. C. Mackay and R. R. H. Anholt. Of flies and man: *Drosophila* as a model for human complex traits. *Annual Review of Genomics and Human Genetics*, 7(1):339–367, 2006.
- [34] K. Matkovic, W. Freiler, D. Gracanin, and H. Hauser. Comvis: a coordinated multiple views system for prototyping new visualization technology. In *Proceedings of the 12th International Conference Information Visualisation*, 7 2008.
- [35] K. Moreland. A survey of visualization pipelines. *IEEE transactions on visualization and computer graphics*, 19(3):367–78, March 2013.
- [36] C. North and B. Shneiderman. A taxonomy of multiple window coordinations. Technical report, University of Maryland, 1997.
- [37] W. Qishi, G. Jinzhu, Z. Mengxia, N.S.V. Rao, H. Jian, and S.S. Iyengar. Self-adaptive configuration of visualization pipeline over wide-area networks. *Computers, IEEE Transactions on*, 57(1):55–68, Jan 2008.
- [38] T. Rhyne, M. Tory, T. Munzner, M. Ward, C. Johnson, and D.H. Laidlaw. Information and scientific visualization: separate but equal or happy together at last. In *Visualization, 2003. VIS 2003. IEEE*, pages 611–614, Oct 2003.
- [39] M. Scherr. Multiple and coordinated views in information visualization. Technical report, University of Munich, 2008.
- [40] W. J. Schroeder, K. M. Martin, and W. E. Lorensen. The design and implementation of an object-oriented toolkit for 3d graphics and visualization. In *Proceedings of the 7th Conference on Visualization '96, VIS '96*, pages 93–ff., Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.
- [41] A. N. Shahrul, Y. Suraya, and S. Shahar. Application of information visualization techniques in representing patients' temporal personal history data. In *Visual Informatics: Bridging Research and Practice, IVIC '09*, pages 168–179, Berlin, Heidelberg, 2009. Springer-Verlag.
- [42] J. Sorger, K. Buhler, F. Schulze, Tianxiao Liu, and B. Dickson. neuromap - interactive graph-visualization of the fruit fly's neural circuit. In *Biological Data Visualization (Bio-Vis), 2013 IEEE Symposium on*, pages 73–80, Oct 2013.
- [43] R. Spence. *Information Visualization*. Addison-Wesley, 2001.

- [44] A Telea. *Data visualization - principles and practice*. A K Peters, 2008.
- [45] M. Q. Wang Baldonado, A. Woodruff, and A. Kuchinsky. Guidelines for using multiple views in information visualization. In *Proceedings of the Working Conference on Advanced Visual Interfaces, AVI '00*, pages 110–119, New York, NY, USA, 2000. ACM.
- [46] R. Yuste. Circuit neuroscience: the road ahead. *Frontiers in Neuroscience*, 2(17):6–9, 2008.
- [47] Mengxia Zhu, Qishi Wu, N.S.V. Rao, and S.S. Iyengar. Adaptive visualization pipeline decomposition and mapping onto computer networks. In *Multi-Agent Security and Survivability, 2004 IEEE First Symposium on*, pages 402–405, Dec 2004.