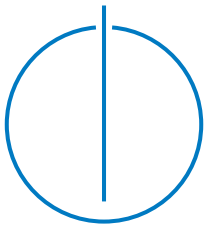
Fakultät für Informatik
Der Technischen Universität München

Bachelorarbeit in Wirtschaftsinformatik

Graphical Interaction on Enterprise Architecture Visualisations

Björn Kirschner



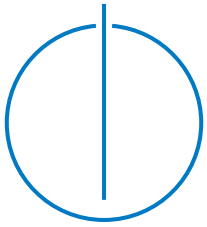
Fakultät für Informatik
Der Technischen Universität München

Bachelorarbeit in Wirtschaftsinformatik

Graphical Interaction on Enterprise Architecture Visualisations

Graphische Interaktion auf Visualisierungen der Unternehmensarchitektur

Supervisor:	Prof. Dr. rer. nat. Florian Matthes
Advisor:	M. Sc. Sascha Roth
Submission Date:	16.04.2012



Declaration

I assure the single handed composition of this bachelor's thesis only supported by declared resources.

München, 16th of April, 2012

Björn Kirschner

Abstract

In the management discipline of Enterprise Architecture, visualisations already are a standard instrument to provide decision support. Yet visualisations of information, particularly within the field of System Cartography, might reach vast dimensions. To analyse such large graphics and benefit from the illustrated information, efficient means of navigation are essential. Nevertheless, sophisticated concepts covering graphical interaction are still rare. Few research or open source components addressing the growing need for functionality which eases the handling of big graphics can be found.

The following work presents design and implementation of an approach to tackle the problem of providing commonly needed navigation mechanisms on Enterprise Architecture visualisations. The implementation is based on an existing web-based Enterprise 2.0 framework enabling collaborative data collection and maintenance in an organisation.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

As the amount of digital data collected and stored in business increases, the question of its effective and efficient utilisation arises. Interpreting and drawing conclusions out of huge sets of raw data may prove to be extremely difficult [BEG+12]. Data is of little value when users do not understand the whole picture it paints.

One way to cope with this problem lies in generating graphics of the data. "Visualisation provides a powerful means of making sense of data" [HS12]. But huge amounts of data lead to large visualisations, which causes usability problems. "Displaying an entire large graph may give an indication of the overall structure, but makes it difficult to understand" [HMM00].

Therefore effective means of navigation (zoom and pan functionality) are crucial in order to benefit from the illustrated information. Navigation allows to examine high-level patterns as well as fine-grained details [HS12].

The chair for Software Engineering for Business Information Systems (SEBIS) at TU Munich has been researching the field of Enterprise Architecture Management for a few years now. After Buckl et al. [BEL+07] had described the complexity of this discipline, the idea of generating graphics visualising certain aspects of the Enterprise Architecture was born [LMW05b]. Currently, a web application for the automated rendering of software maps and other visualisations is being developed. As these visualisations may reach vast dimensions, often several times the size of a normal computer screen, efficient means of navigation thereon have to be found and implemented. This thesis addresses the need for such navigation mechanisms.

Structure

At first, Chapter 2 will introduce several important terms related with title and topic of this thesis. As one goal of the thesis is the development of a navigation component, Chapter 3 will define requirements for the implementation. After Section 4 explaining design and structure of the application, Chapter 5 will describe selected aspects of the implementation in detail. The testing procedure will be listed in Section 6. Chapter 7 will compare this work to other products providing rich navigation user interfaces. Finally, a conclusion and an outlook will be given.

2 Terminology, definitions and basic concepts

The following chapter introduces terms and concepts used throughout this thesis. It starts on a high abstraction level - describing **Enterprise Architecture** as the broader scope the developed navigation functionality is chiefly aimed at. The subsequent section justifies the need for visualisations as part of **System Cartography** in the discipline of Enterprise Architecture Management. Section 2.3 then lays the foundations for conceptually understanding these visualisations. Thereby interaction, particularly **Graphical Interaction** will be defined. Finally, on an implementation level, the technological basis of the generated visualisations - **scalable vector graphics** - will be explained.

2.1 Enterprise Architecture

Most software systems nowadays consist of a huge number of distinct applications. All of those pursue the goal of increasing automation and efficiency, but are very specialised and have a narrow field of application [Mas05]. This diversity increases complexity and, as a consequence, costs for managing it. **Enterprise Architectures (EA)** should provide a "holistic view of the enterprise" [Lan05] and thereby facilitate solving this problem.

The ANSI/IEEE standard 1471-2000 introduces a formal term definition. According to this standard, architecture "is defined by the recommended practice as the fundamental organisation of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution" [IEE00].

Following this definition, Enterprise Architecture leads to a "coherent whole of principles, methods, and models that are used in the design and realisation of an enterprise's organisational structure, business processes, information systems, and infrastructure" [Lan05].

As technology of information systems and business processes change rapidly, Enterprise Architecture has to cope with changing circumstances. Beyond that, EA should be able to influence this evolution [Mas05].

Keller [Kel07] derives the following tasks for an IT enterprise architect out of these definitions:

- Make the IT support the business strategy, even if a vast number of IT systems is involved.
- Provide ways to manage a huge portfolio of expensive IT systems.
- Care about the complexity of the enterprise's IT.
- Facilitate changes of applications and the whole portfolio due to innovation.

Enterprise Architecture frameworks provide guidance for introducing an architecture. They specify "conventions, principles and practices" [IEE11]. Buckl et al., for instance, present a conceptual framework for Enterprise Architecture design [BMR+10a][1].

Stakeholders and viewpoints

A **stakeholder** is an "individual, team, organisation, or classes thereof, having an interest in a system" [IEE07, IEE11]. An Enterprise Architecture has to be focused on the needs of its stakeholders.

Usually, many stakeholders with diverging demands are involved in an IT system. When their expectations cannot be conciliated, different **views** on the architecture have to be created. A **viewpoint** is a specification for a view.

[1] Further information on the current state and process of Enterprise Architecture Management can be found in [BBF+12] and [BMR+10a]

As the creation of new views may be complex and time-consuming, viewpoints define how to reuse views.

Stakeholder-specific views can be found, inter alia, in [BMR+10b] and [BMM+11]. Buckl et al. visualise matrices for data access on business objects.

2.2 System Cartography

As application environments are growing steadily, so does the need for describing them. Lankes et al. [LMW05a] quote that in these architectures with hundreds or even thousands of information systems providing suitable documentation is vital for management and planning of changes in the architecture. To address this need for documentation, the SEBIS chair at TU Munich introduced terms and technologies of cartography in the scope of software architectures.

Wittenburg [Wit07] defines **software maps** as "graphical models for architecture documentation of an application environment." A software map consists of a base map and layers which visualise various characteristics. All layers on top refer to the base map or other layers. Those layers can address different topics while reusing the same base map. For a better overview they can be switched on and off. Since the base map always stays the same, analysts profit from a high recognition factor [LMW05a]. Figure 2.1 illustrates the schematic example of a base map with three further layers on top.

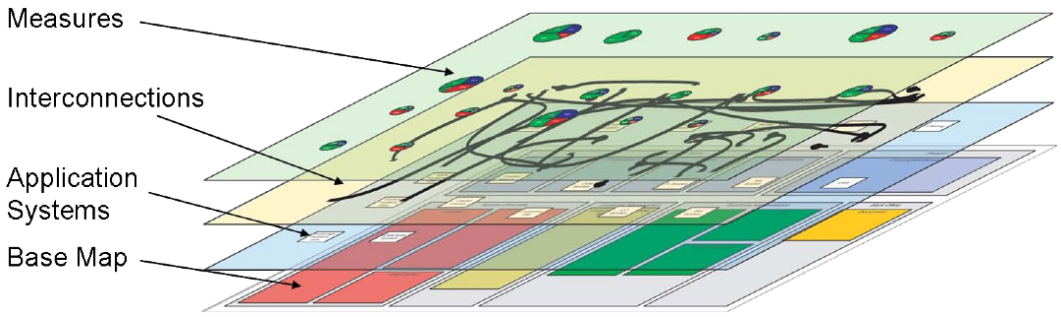


Figure 2.1: Example of layers in a software map [Mat08]

One typical example in software cartography are **Cluster Maps**. Within those, the base map shows logical entities, which are placed according to their hierarchical structure and relationships [LMW07].

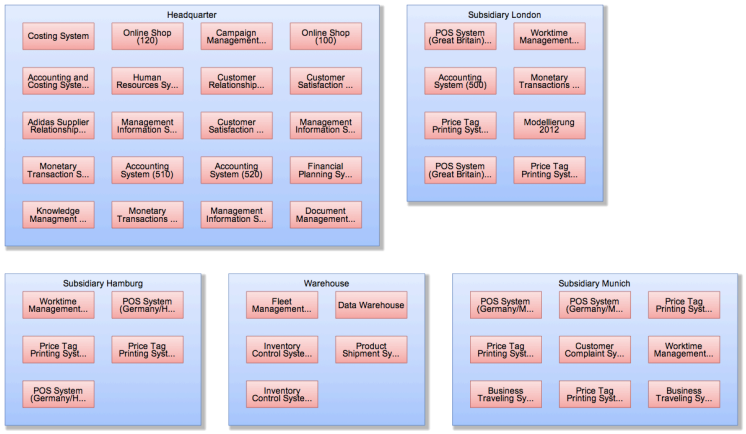


Figure 2.2: Example of a dynamically generated Cluster Map [SEB12a]

But software maps can also exist without base map. **Sugiyama Maps** for example, named after Kozo Sugiyama, visualise hierarchical graphs. They show vertices and edges placed on horizontal lines called layers [Sug02]. Figure 2.3 shows an example. Compared to maps based on a base map, a Sugiyama Map is rendered by a graph layout algorithm. Since no information about the position of elements is know here, nodes might be placed differently every time the map gets rendered. A sophisticated algorithm searches for a layout that, amongst other goals, minimises edge crossings, minimises separation between neighbouring vertices, and cares about parent nodes being placed near their children [STT81].

Information visualisations versus scientific visualisations

In the discipline of Enterprise Architecture Management visualisations are generally based on "large-scale collections of non-numerical information" [Fri08], following a data schema. Such information visualisations are not to be confused with scientific visualisations. The latter are "primarily concerned with the visualisation of 3-D phenomena (architectural, meteorological, medical, biological, etc.), where the emphasis is on realistic renderings of volumes, surfaces, illu-

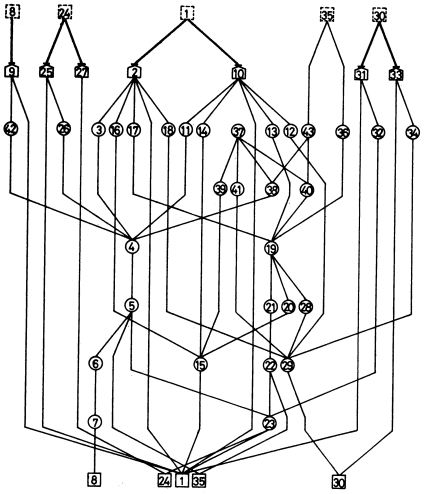


Figure 2.3: Example of a Sugiyama Graph with nine levels [STT81]

mination sources, and so forth, perhaps with a dynamic (time) component" [Fri08].

2.3 Graphical Interaction

Figure 2.4 shows the conceptual model of architectural descriptions according to the ISO/IEC standard 42010 [IEE07]. Every information system fulfils a mission. It may have many stakeholders with diverging concerns. Viewpoints are a concept to satisfy the interests of different stakeholders. A view expresses "a system's architecture with respect to a particular viewpoint" [IEE07].

Schaub et al. [SMR12, HMRS12] took one step beyond mere visualising of software maps. They introduced a conceptual framework for interaction on Enterprise Architecture visualisations.

The *data model* typifies actual data of a data source within an enterprise. "An *information model* is a representation of concepts, relationships, constraints, rules, and operations to specify data semantics for a chosen domain of discourse" [Lee99]. It defines the schema on which the data model is based [BGS10].

The *view data model* is the result of a query on the data model. It retrieves the part of the data needed for the visualisation. It may contain aggregations

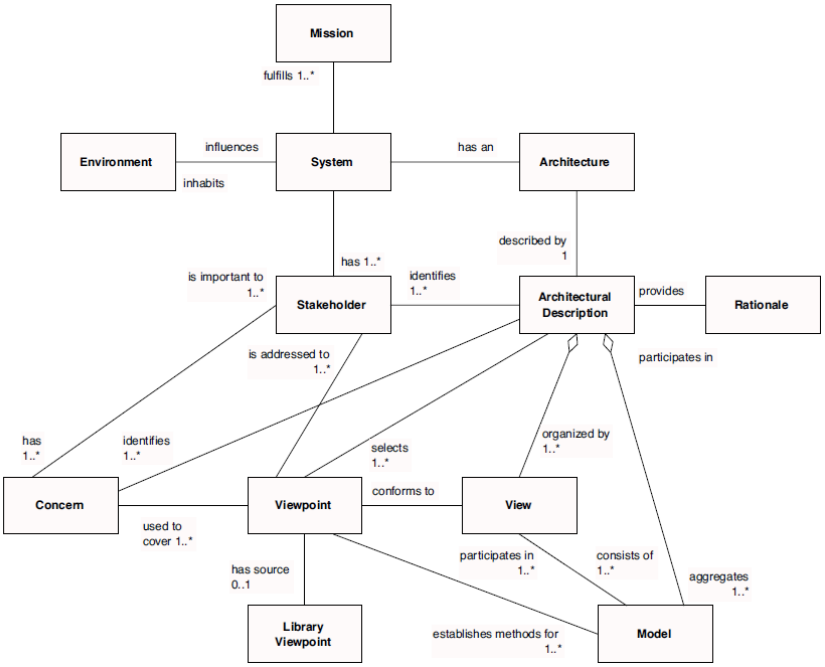


Figure 2.4: Conceptual model of architectural description according to ISO/IEC Std. 42010 [IEE07]

or average values. The *view model* is the schema corresponding to the view data model.

A *visualisation model* holds definitions of visual primitives like geometric shapes or compositions of shapes. Together, view model and visualisation model define a viewpoint following the ISO definition [IEE11]. The *symbolic model* contains all rendered elements visible to the user. Among these elements are the symbols instantiating the objects defined in the visualisation model. Furthermore, the symbolic model includes elements which are not related to a viewpoint, e. g. a general context menu or a bar that shows the zoom level.

According to Schaub, interaction can be defined on each of the models described above. Possible interaction for the visualisation model, for example, includes rules about whether a certain object may be dragged and where it may be dropped. Interaction related to the symbolic model, called *symbolic interaction* by Schaub, provides general interaction functionality independent of a certain viewpoint. A familiar example therefor is zoom and pan capacity.

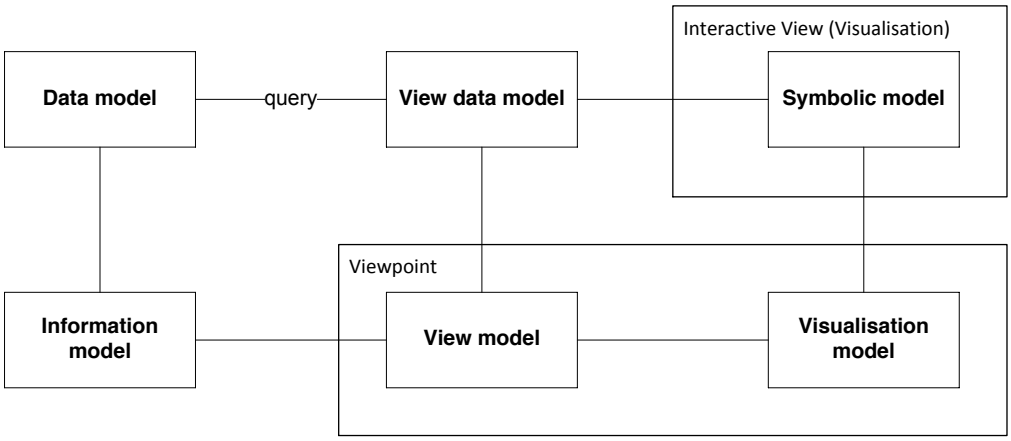


Figure 2.5: A conceptual framework to generate interactive web-based visualisations according to Schaub et al. [SMR12]

Yet the name "symbolic interaction" suggests that its interaction capability depends on or is related to symbols. The Brockhaus encyclopaedia [bro93] defines "symbol" as a visual element with a clear meaning. It emphasises that the term "symbolic" includes a semantic constituent.

However, in Schaub's framework symbolic interaction describes interaction which is common for all graphical elements of a visualisation irrespective of what symbol, form, or semantics they might have. The ambiguity of the term "symbolic" shall be avoided within this thesis. Hence general interaction on the rendered graphics valid on all visualisations regardless of the viewpoint is called **Graphical Interaction** here.

2.4 Scalable vector graphics (SVG)

From a technical point of view, visualisations in the Tricia GraphicalVisualizations plugin are built on **scalable vector graphics (SVG)** technology. SVG is a XML based language for two-dimensional graphics. It is an open World Wide Web Consortium (W3C) standard [W3C11]. SVG offers all advantages of vector graphics especially for web applications. Compared to bitmap graphics, vector graphics can be scaled without any perceived quality loss.

Listing 2.1: Simple SVG example

```
1 <svg width="1200" height="600" xmlns="http://www.w3.org/2000/svg" version="1.1"
      viewBox="0 0 1200 600" preserveAspectRatio="xMinYMin">
2     <circle cx="50" cy="50" r="20" stroke="red" fill="red" />
3     <rect x="10" y="10" width="20" height="40" stroke="blue" fill="blue" />
4 </svg>
```

Listing 2.1 shows how scalable vector graphics are declared in HTML. This simple example consists of a SVG container element which contains a circle and a rectangle. Attributes of the SVG tag define the XML namespace - here the W3C SVG standard - and the used SVG version. *Width* and *height* specify the borders of the element. Especially interesting are the attributes *viewBox* and *preserveAspectRatio*.

View Box

ViewBox provides the possibility to establish a different viewport on the SVG. Its four arguments specify x-coordinate, y-coordinate, width and height of the new viewport. In the given example the view box starts in the top left corner and covers the whole SVG element. Changing the view box parameters to (0, 0, 600, 300) would stretch everything visible within a rectangle from the coordinate origin to the point (600, 300) so that it fits the whole SVG element. With other words, this would enlarge the visualisation to a zoom level of 200% relative to the coordinate origin in the top left corner. Parts of the visualisation not within the new viewport are cut off.

To display other parts of the visualisation, x- and y-coordinates of the view box can be changed. The values (600, 300, 600, 300), for example, would show the bottom right part of the original graphics while keeping the zoom level of 200%.

All zooming and panning functionality in this work is based on setting the view box of a SVG.

Aspect Ratio

The SVG attribute *preserveAspectRatio* specifies how to handle graphics when the aspect ratios of the SVG element and its view box do not match. If it is set to "none", non-uniform scaling is allowed. The graphics then takes up the whole viewport, even if it has to be stretched with different factors in x and y-direction. Consequently, using the value "none" might distort the visualisation.

All further values of *preserveAspectRatio* enforce uniform scaling. Graphics are then always scaled in the aspect ratio of the viewport, regardless of whether it differs to the aspect ratio of the view box. *PreserveAspectRatio* now also has to define the position of the scaled visualisation inside the viewport. "xMinYMin", for example, places it into the upper left corner of the SVG container. X and Y can be handled individually. "xMidYMax" would also be valid and places the visualisation into the lower part of the viewport, but centred in x-direction.

Optionally, either "meet" or "slice" can be added to the options above. "Meet" makes the whole view box visible inside the viewport, while "slice" ensures that the entire viewport is covered by the view box. Figure 2.6 illustrates all options of the SVG attribute *preserveAspectRatio*.

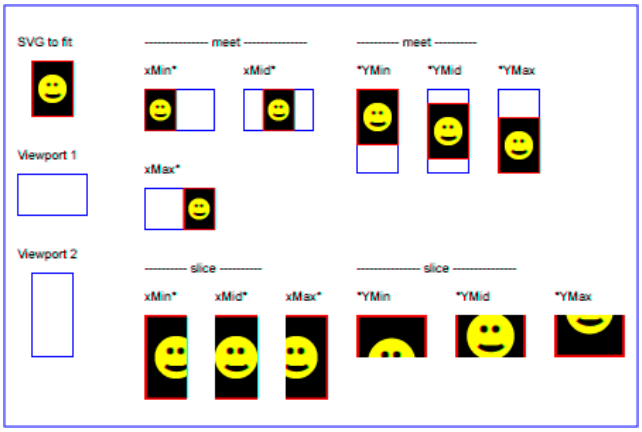


Figure 2.6: Different options of the SVG's *preserveAspectRatio* attribute [W3C12]

Raphaël JavaScript library

As a layer between SVG and custom JavaScript code, Tricia relies on the Raphaël JavaScript library. Raphaël provides a small yet mighty toolset to handle vector graphics on the web, following the W3C SVG standard. The library also assures browser independence.

Creation and manipulation of graphical primitives is all be done using this library.

In the context of Raphaël, each SVG element is referred to as a *paper*.

3 Requirements

This chapter lists the required functionality of a component which handles interaction concerning the navigation. All interaction mechanisms should be well adapted for Enterprise Architecture visualisations.

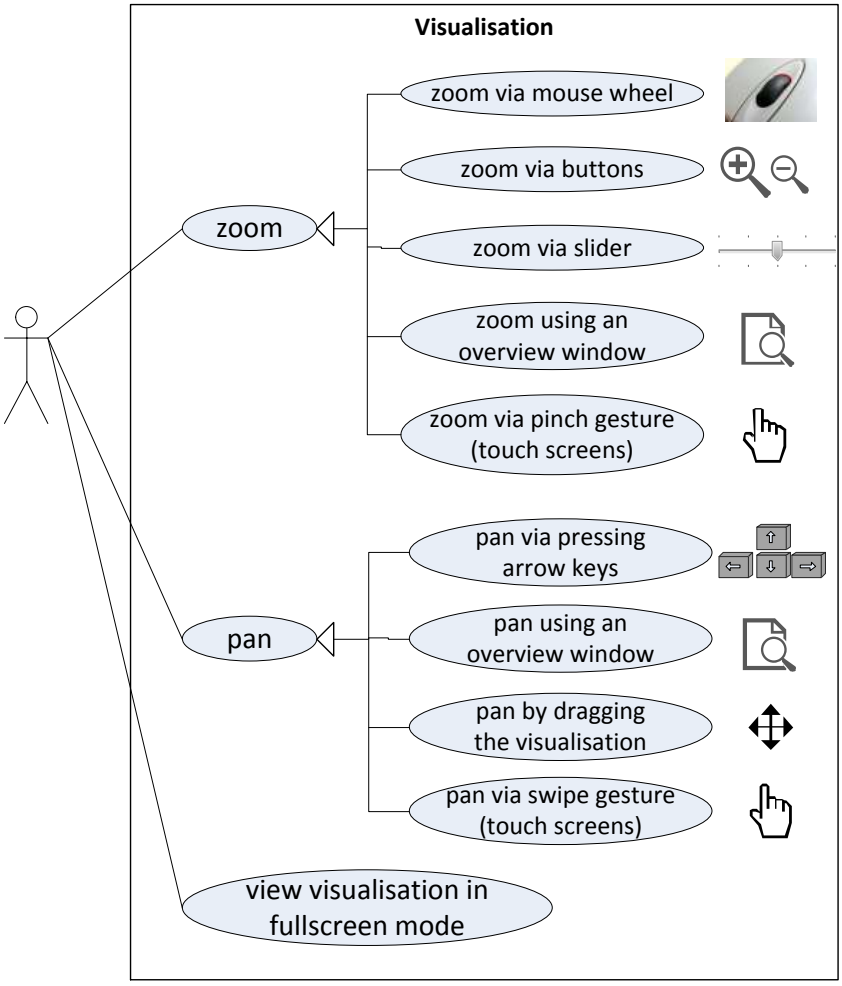


Figure 3.1: Use case diagram depicting functional requirements of a navigation component

3.1 Functional requirements

Use case diagram 3.1 illustrates basic functional needs on stationary computers. For mobile devices like tablet computers or smartphones, sub-requirements differ slightly. On those devices, touch gestures should sufficiently satisfy navigation demands.

Visualisations should be supported by a component which shows a minimised visualisation for a better overview. This module should contain a rectangle called **View Frame** which represents exactly the part of the graphics currently visible. It should help users to keep track of which part of the graphics they have navigated to. Figure 3.2 illustrates a mockup of such an overview window.

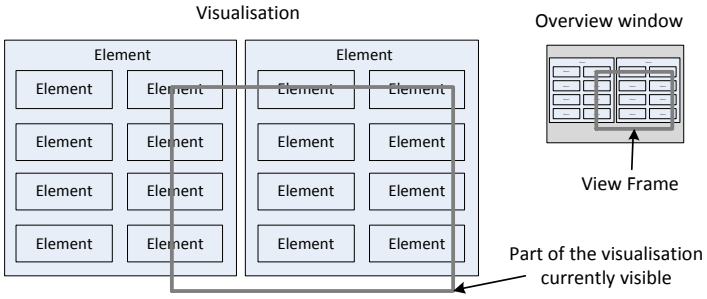


Figure 3.2: Visualisation and overview window with View Frame

3.1.1 Zoom

One functionality essential on large graphics is zooming. Since it is one of the two core functionalities of navigation (along with panning), multiple ways to zoom in and out should be provided. These ways should include interaction via more than one input device. Brown [Bro87] calls this strategy *multiple paths*.

As shown in the use case diagram 3.1, turning the mouse wheel, pressing special buttons, and dragging the handle of a slider should all trigger the zoom process. Additionally, the overview window should provide zooming functionality by stretching a new or resizing an existing View Frame. The overview component then has to set the zoom level according to the size of the new View Frame.

For mobile devices, different interaction mechanisms have to be found. The implementation should support the pinch gesture (increasing or decreasing the distance between the touches of two fingers).

3.1.2 Pan

Panning is the second core functionality of navigation. One way to pan should be achieved by pressing the arrow keys on the keyboard. Apart from that, the visualisation itself should be draggable via mouse interaction. Aditionally, users should be allowed to drag the View Frame of the overview window. The new View Frame position can be translated to new pan position coordinates.

On mobile devices, panning via the swipe touch gesture (stroke with one finger in a certain direction) should be enabled.

3.1.3 Fullscreen mode

A visualisation is usually shown inside a corporate web site. In general, these sites place a menu bar and logos above all content. Content itself is restricted to boundaries of a container element. This leads to the fact that content often cannot use the full width of the browser window. Since a visualisation is part of the content, it is also subject to the named restrictions. This lessens the overview as users have to scroll until the visualisation is shown in the centre of the window. Furthermore, it is impossible to use the whole browser width if the container does not cover the entire width. These drawbacks lead to the following requirement:

Users should be able to view a single visualisation extracted from all context in a “fullscreen mode”. Fullscreen in this work is defined as a visualisation taking up the whole space of a browser window.

The fullscreen mode is to be accessible on mobile devices, too, as navigation there is particularly problematic due to small screens.

3.1.4 Multiple visualisations on one page

The implementation has to be able to cope with multiple visualisations on a single page. Interaction with each visualisation individually, without side effects to any other visualisation on the page, must be possible.

3.2 Non-functional requirements

Non-functional requirements defined for this work include **usability**, **optimisation for all screen resolutions**, **performance** and **extensibility**.

3.2.1 Usability

In his book "Human-Computer Interface Design Guidelines"[Bro87], Brown names *ease of learning*, *ease of use* and *functionality* as key principles for designing good user interfaces.

- *Ease of learning* is "the extent to which a novice user can become proficient in using a system with minimal training and practice."
- *Ease of use* means "the extent to which the system allows a knowledgable user to perform tasks with minimal effort." This includes performing tasks in little time and using few keystrokes.
- *Functionality*, finally, is defined as "the number and kind of different functions the system can perform."

The challenge lies in finding an appropriate balance of those three principles. According to Brown, *ease of learning* is not to be overestimated, as users learn quickly and might soon "outgrow the system". The implementation of this work should follow his guidelines:

- Fundamental functionality should be easy to learn.
- Functionality which is often used should be easy to perform. Short access times and *multiple paths* should be assured therefor. Multiple paths

means, that a single core functionality can be carried out in different ways, for example using different input devices.

- The user interface should encourage experimentation.
- The user should be able to reverse unintended actions.
- Defaults should exist to help minimising the number of user selections.

3.2.2 Optimisation for all screen resolutions

Navigation on visualisations should feel comfortable on a wide range of screen resolutions. Users should be able to benefit from this functionality regardless of whether they display content on a large stand-alone computer monitor, a medium-sized laptop, or even a mobile device like a smartphone. The fact that displays do not only differ in size but also in their aspect ratio has to be taken into consideration, too. Laptops, for example, tend to have wide displays with aspect ratios of 16:10 or 16:9, whereas classic PC monitors are usually built in a 4:3 width to height ratio.

Providing intuitive and efficient handling on very small screens as they can be found on tablet computers and smartphones is highly difficult. Therefore visualisations should be specially optimised whenever they are displayed on such tiny screens. For this optimisation Raneburger et al. [RPK+11] postulate to

- make maximum use of the available space
- with only a minimum amount of navigation clicks
- and minimum scrolling effort.

3.2.3 Performance

One aspect of performance is the response time after user interactions. It should not exceed one second. Dahm [Dah06] states that users only consider response times within one second as immediate.

Miller [Mil68] defines an even stricter time frame and quotes that only computer reaction within 0.1 seconds is felt as instantaneous. During response times between 0.1 seconds and 1 second the user's "flow of thoughts stays uninterrupted" [Nie93] but the delay will certainly be noticed.

For response times of over one second, Nielsen's usability heuristics [Nie93] demand feedback which informs users about what the computer is doing and how it is interpreting user inputs.

3.2.4 Extensibility

If a need for new components arises, a programmer should be able to integrate those with little effort and ideally without changing existing code. Especially visual elements should be easily replaceable and extendable, as they are most likely to be subject to change whenever user preferences change.

3.3 Environment

All development related to this thesis is done within the scope of a tool called **Tricia**. Tricia is a software platform for web collaboration and information management [inf11]. It includes a plugin for visualisations, in particular system maps, called **GraphicalVisualizations**. This plugin is currently being developed at the SEBIS chair in Munich.

GraphicalVisualizations automatically generates the structure of visualisations on the server and passes all graphical elements to the client. The client then interprets these JavaScript elements and renders visible items as Scalable Vector Graphics (SVG). SVG is not accessed directly but via the third-party library Raphaël as an intermediary layer.

All user interaction functionality is implemented in JavaScript and executed in a browser.

4 Design

The GraphicalVisualizations plugin generates, of course, visualisations. The object representing a visualisation is hence the centre of all application logic on the client side.

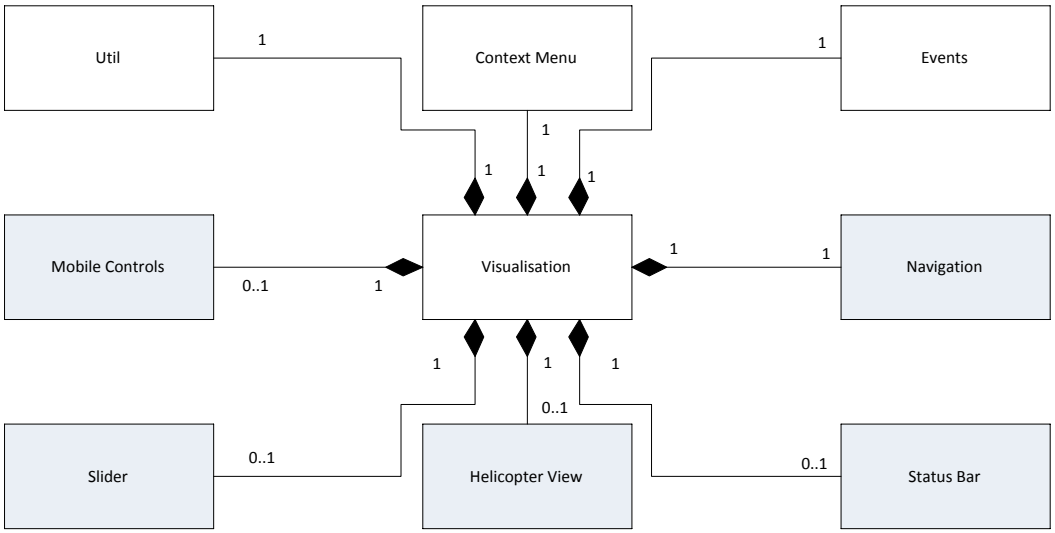


Figure 4.1: Class diagram illustrating the structure of components existing around a visualisation. Components related to navigation are painted blue.

A few essential components exist around a visualisation. The Util class provides generally helpful functionality. Context Menu handles the menu which appears after a click with the right mouse button onto the visualisation. Events, finally, handles communication between components.

4.1 Components for navigating on visualisations

All components needed to navigate on the visualisation are painted blue in diagram 4.1. Note that Mobile Controls are only visible on mobile devices and that the Navigation component does not contain any visible items.

4.1.1 Visible components

All visible navigation tools will be shown inside a container element. This may contain a Status Bar, a Slider and an overview window called Helicopter View. As multiplicities in class diagram 4.1 indicate, all visible components are optional and only shown if the user wishes them to appear.

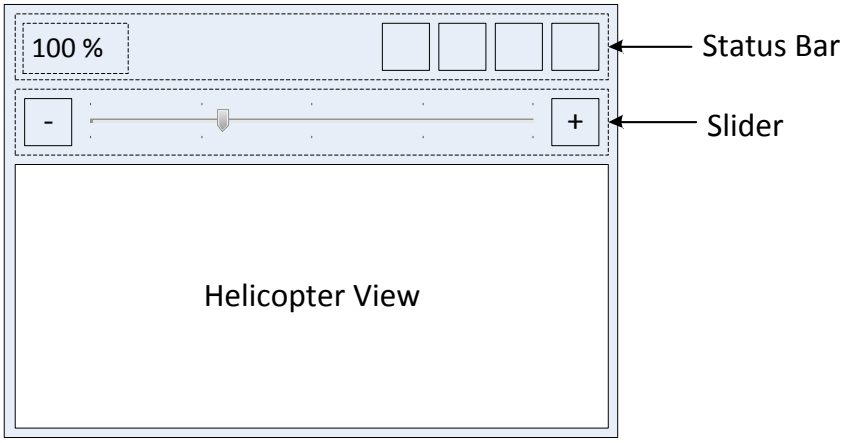


Figure 4.2: Mockup of navigation components on a stationary computer

4.1.1.1 Status Bar

In the Status Bar, the current zoom level will be displayed. In addition, four buttons will provide the following functionality: “fit horizontal”, “fit vertical”, “fullscreen mode” and “open visualisation in a new window”.

“Fit horizontal” stretches and pans the visualisation until it fits into its original boundaries in x-direction. “Fit vertical” does the same vertically, i.e. in y-direction. “Fullscreen mode” provides the possibility to show the visualisation

covering the whole display area. Everything apart from the graphics will be not visible. “Open visualisation in a new window” pursues a similar concept except that it opens a new browser window for showing the graphics. If fullscreen mode is already engaged, the two latter buttons will embed the visualisation into its former context again.

4.1.1.2 Slider

The Slider component contains two buttons to increase and decrease the zoom level. A jQuery UI slider will graphically visualise the zoom level and allow to change the zoom factor by dragging the handle of the slider.

4.1.1.3 Helicopter View

Finally, the heart of the navigation is the Helicopter View.

The term Helicopter View follows the nomenclature of Schweda [Sch06]. In other applications a similar concept is called *Navigator* (Adobe Photoshop) or *Pan & Zoom Window* (Microsoft Visio).

A Helicopter View is a window which shows a miniature view of a visualisation. Its main purpose is to provide an overview. Figure 3.2 shows an example of a visualisation and its Helicopter View.

The Helicopter component will ensure that the View Frame gets updated after every graphical interaction, so that it always represents the currently visible part of the visualisation. When users interact with the Helicopter View, this component will carry out all calculations necessary to transform View Frame’s size and position to a new zoom level and pan position of the visualisation.

4.1.1.4 Mobile Controls

The composition of navigation components will be different, whenever users open visualisations using a mobile device such as a smartphone or a tablet computer.

Tests showed that using a Helicopter View on a touch screen device is very difficult. The interaction with the Helicopter View relies heavily on various mouse events. Especially "mouse over" events cannot be translated to a touch event because as long as the user does not touch the screen (which would correspond to the mouse click event), the device has no idea, where the user's finger is. Hence, using the Helicopter View on a touch device would require a completely new interaction concept.

Apart from this problem, a Helicopter View would occupy too much of the device's screen. As mobile devices tend to have relatively small displays, a first thought was to scale down the size of the helicopter view accordingly. But the Helicopter View then becomes totally unusable, because much space is needed to accurately track touches. Apple's iOS Human Interface Guidelines, for instance, state that UI elements should have a minimum size of 44 x 44 points to be comfortably tappable [App12].

Those two reasons lead to the conclusion that a Helicopter View is not useful on mobile devices. It is also not wise to spend much space for big control bars. Therefore, Status Bar and Slider will not be shown, either. All controls which cannot be renounced are bundled in the Mobile Controls component. This minimal set of controls will be well adapted for touch interaction. Currently, only one button invoking the fullscreen mode is shown.

4.1.2 Navigation component

The navigation's core component is simply called Navigation. It does not have any visible elements. Instead, it bundles all interactions concerning the navigation, and saves zoom level and view box parameters of the visualisation. The Navigation is the only component that is directly linked to the visualisation via a bi-directional association and allowed to set its view box.

Status Bar, Slider, Helicopter View and Mobile Controls only communicate with the Navigation, never with the visualisation directly. Thus, the state of all variables relevant to the navigation can be kept consistent.

If all components directly manipulated the visualisation, notification and synchronisation would be rather difficult and race conditions might occur.

4.2 Communication between components

As already mentioned, the Navigation manages zoom level and view box position. It is the only component which is allowed to set the view box on the visualisation. Every other component which wants to pan or zoom has to interact with the Navigation. If any component manipulates view box or zoom level, all other components have to be informed about the changes. If, for example, a user drags the slider handle, the Slider component has to pass the new zoom level to the Navigation. This one saves the new value and sets the view box on the visualisation accordingly. Now as a last step, all components have to be updated. In this case the Status Bar, so that it displays the new zoom level and the Helicopter View, which must adjust its View Frame to represent the view box on the actual visualisation.

4.2.1 Naive design

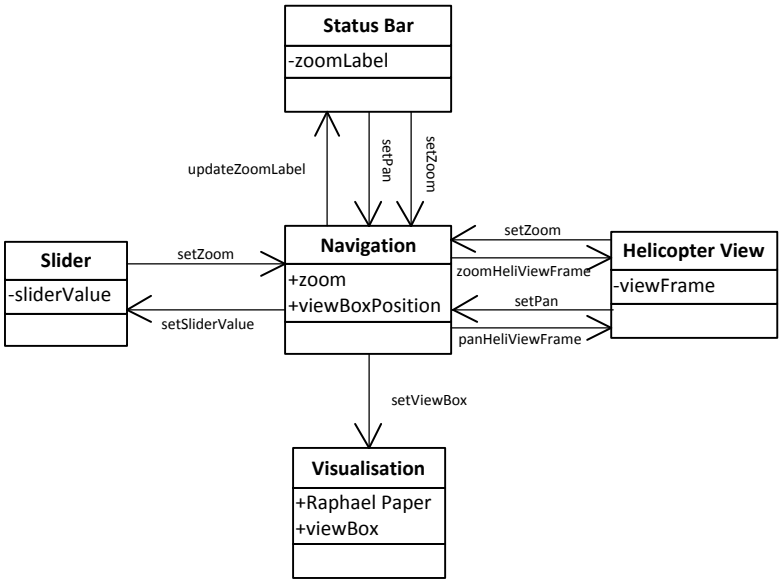


Figure 4.3: Communication between components in a naive design

A naive approach would let all components communicate directly with each other according to their needs. This would lead to a lot of unidirectional associations. Figure 4.3 illustrates the necessary communication between the

components as a class diagram. It obviously results in many circular dependencies. But cycles in dependencies are generally considered a bad software design because of various reasons. The tight coupling makes it difficult to integrate new components as well as to reuse an existing one for a different purpose. Besides, for the developer, circular dependencies are hard to grasp and likely to cause unwanted effects and errors [GHJV95].

4.2.2 Communication via events

Due to the shortcomings of circular dependencies described above, in this work a different design is chosen. The idea is to make all components communicate via events and let an additional abstraction layer handle those events. Every component shall be able to trigger and listen to events. If an event is fired, the event component will automatically forward it to all components which have registered to that particular event. The sender does not need to know who will obtain the message, nor how many will get it. Just as the recipient does not have to know who the sender is. Thus, any circular dependencies can be avoided.

The example described in the previous chapter will now work differently. The Slider whose handle is moved would now simply fire an event with the name "zoom" and a number representing the change of the zoom factor. Navigation, Status Bar and Helicopter View component are registered to the "zoom" event and will all get identical information about the fired event. Figure 4.4 illustrates this flow of communication. After receiving the event, each component acts independently. The Navigation computes the new view box, the Status Bar updates the zoom factor label, and the Helicopter View sets its View Frame.

Event names can basically be everything. But in this work, implementation will follow a few naming conventions. Event names may be structured hierarchically by using a point (".") as separator. Instead of just calling a "zoom" event, one could add information about who triggers the event. The Slider, for instance, would send a "slider.zoom" event. Information refining the event is also possibly useful. "Navigation.pan.start", for example, could be called if processing needs to be done before the panning starts.

With these conventions, event names can be extremely detailed, although a consuming component might not care about additional information at all. Most components probably want to register to all zoom events, regardless of who they were sent from. Therefore, components can use wildcards when registering to an event. Wildcards are represented by a "$*$". A component listening to all zoom events will therefore register to the event "*.zoom".

Fortunately, the used Raphaël framework already provides an event component which meets all desired criteria.

Diagram 4.4 shows an overview of which events the navigation components are typically registered to and which events they fire.

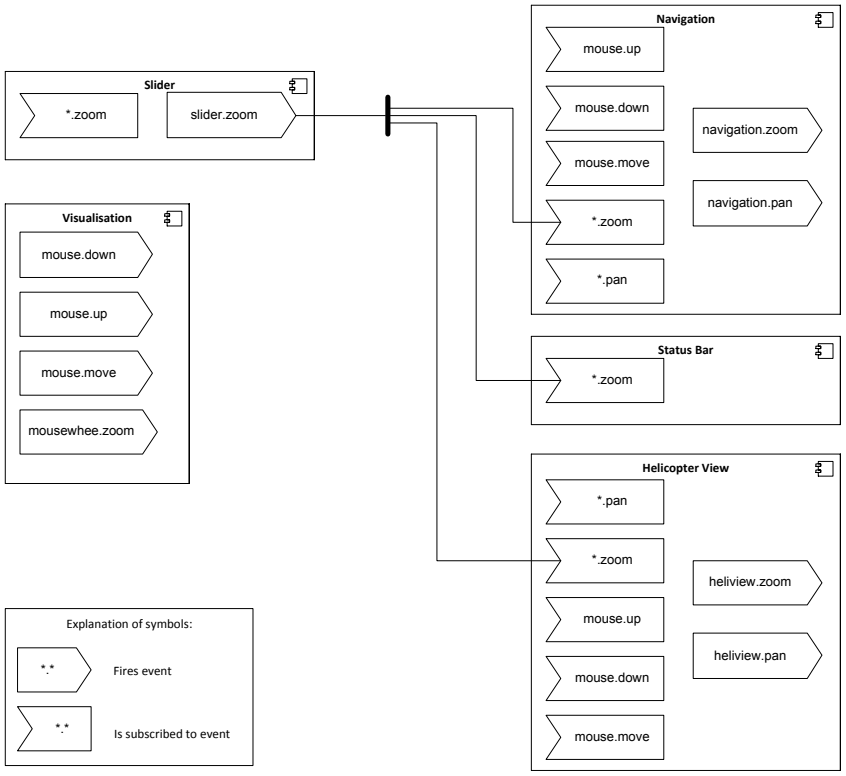


Figure 4.4: Illustration of the events navigation components fire and are subscribed to. The connectors depict an exemplary event flow triggered by a move of the slider handle.

4.2.3 Events on pages with multiple visualisations

According to the requirement described in 3.1.4, multiple visualisations on one web page have to be supported. However, events, as introduced in the previous chapter, would lead to unwanted side effects whenever there is more than one visualisation on the web page.

Since the used event component is part of the Raphaël library, it exists globally. Hence, events intended for one visualisation would affect all visualisations on the page. All events (most importantly zoom and pan) are subscribed by the components of every visualisation. So, if a user zooms in one visualisation, all others will also receive the event. They would mimic the behaviour of the first visualisation and zoom likewise.

To prevent this effect and allow each visualisation to be zoomed and panned differently, events must be bound to a single visualisation. The solution is to add a visualisation identifier to every event name. Every visualisation possesses a unique ID.

To keep matters simple and add this identifier in one central point, the component *Events* was introduced. It is attached to a visualisation and visible to all of the visualisation's other components. Thus, navigation components can register to, unregister and fire events interacting only with their Events component. The navigation components use the notation defined in chapter 4.2.2 and do not need to care about the visualisation ID. The latter is concatenated to the event name when the Events component translates those custom events to Raphaël events.

5 Implementation

This chapter explains important issues of the implementation. It starts with the question of how to fit a visualisation into the Helicopter View and how to navigate inside it. Subsequently, the implementation of interaction functionality on a stationary computer is demonstrated, followed by interaction on mobile devices. Finally, two different facets of the desired fullscreen mode are described, benchmarked and compared.

Before this, an overview over terminology and different container sizes on computer displays is shown.

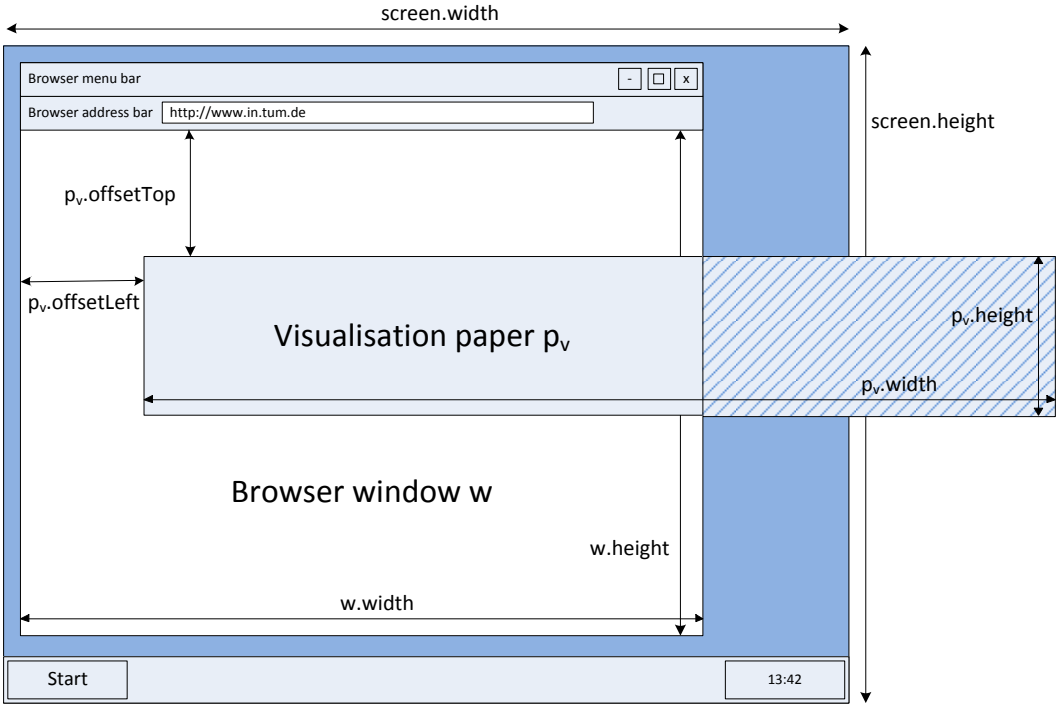


Figure 5.1: Simple mockup of a visualisation on a computer screen

In the most simple scenario, exactly one visualisation is rendered inside a web page. No further content and no navigation controls are shown. Figure 5.1

shows how this scenario might look. Requirements in section 3.2.2 ask for efficient utilisation of the screen size. But since visualisations in this work always run inside a web browser, concessions have to be made. The operating system occupies space e.g. for the taskbar and usually does not allow its applications to cover the whole screen. The browser itself usually shows at least one menu bar and an address bar. The remaining inner size of the window is the maximum which can be used for a visualisation.

But the size of a visualisation is totally independent of both browser and screen dimensions. It might exceed the browser's size. Furthermore, a visualisation can be placed anywhere via the paper's offset attributes. The one in figure 5.1 is quite wide and placed with much offset. As a result, only the part of the paper which is inside the browser window can be seen. The striped part of p_v is not visible. Users would have to use navigation functionality to see this part.

When it comes to navigating, one mighty tool is a Helicopter View. It should give an overview of the visualisation.

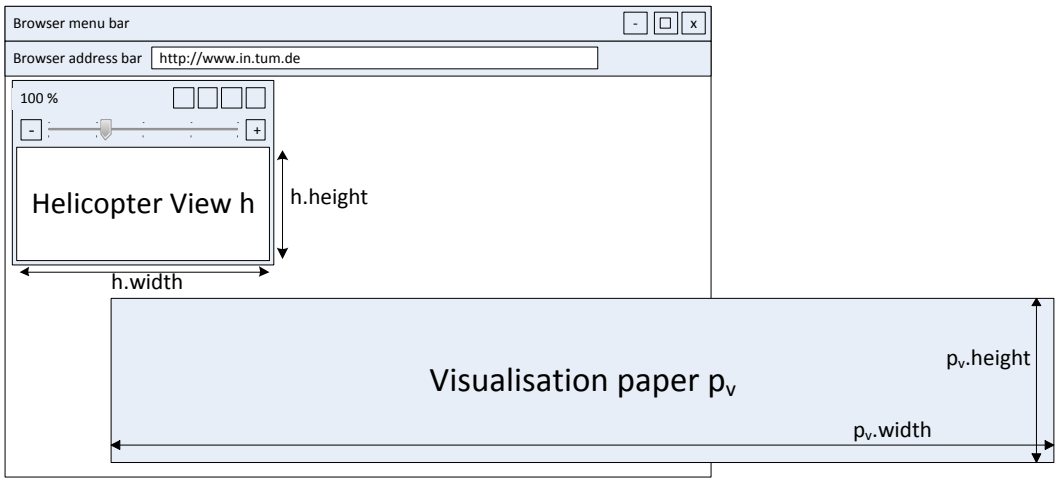


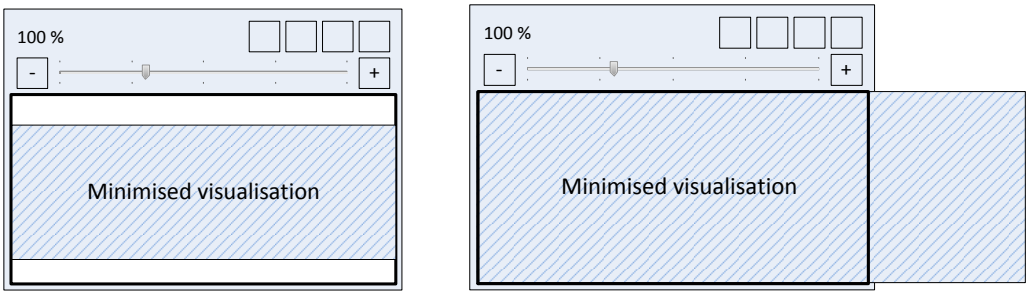
Figure 5.2: Mockup of visualisation and Helicopter View inside a browser

5.1 Fitting the visualisation into the Helicopter View

Users can specify the size of the Helicopter View according to their preferences. However, this size is very unlikely to match the aspect ratio (width to height) of p_v. Which leads to the question of whether to scale p_v with the horizontal scale factor $s_{hor} = p_v.width/h.width$ or using the vertical scale factor $s_{vert} = p_v.height/h.height$.

Essential for answering that question is whether the whole visualisation is to fit into the Helicopter or not. If this is the case, p_v must be scaled with the bigger of s_{hor} and s_{vert}. Otherwise the minimum of s_{hor} and s_{vert} has to be applied.

Using the scale factor s = $Max(s_{hor}, s_{vert})$ forces the whole visualisation into the Helicopter. See figure 5.3a for an example. Depending on the aspect ratios, much space might be left unused and graphics might become very small and hard to interprete.



(a) s = $Max(s_{hor}, s_{vert})$ (b) s = $Min(s_{hor}, s_{vert})$

Figure 5.3: Alternatives of scaling the visualisation to fit into the Helicopter View

Calculating with the other alternative, s = $Min(s_{hor}, s_{vert})$, makes use of all available Helicopter View space. The example of the two illustrations in 5.3 indicates a big size difference. Navigating on the right visualisation which is much larger than in the left picture will be significantly more precise. But, as figure 5.3b also shows, the minimised visualisation is now bigger than the boundaries of the Helicopter View, if the aspect ratios of p_v and the Helicopter do not match.

The first implementation of this work used alternative (a) to fit the whole visualisation into the Helicopter. But it soon turned out that especially on Sugiyama graphs, which in one test scenario had aspect ratios of 5:1, graphics in the Helicopter View were shrunk so much that interaction became nearly impossible. Hence, out of usability reasons, implementation was changed to alternative (b) of scaling with s = $Min(s_{hor}, s_{vert})$.

Literature affirms this approach. According to Plaisant et al. [PCS95] benefits of a Helicopter View are highest when the ratio of overview to detail view lies between five and twenty. If the visualisation was scaled using the formula s = $Max(s_{hor}, s_{vert})$, this ratio would be much higher than twenty in many cases.

After those calculations, an algorithm iterates through all elements of the visualisation paper p_v and copies them to the SVG paper p_h inside the Helicopter View. All objects are divided by the scale factor s. Element attributes such as "stroke width" are also scaled down by applying this factor.

5.2 Navigating inside the Helicopter View

A big problem of the implemented scaling method is how to reach the parts of minimised visualisation which are outside the Helicopter boundaries and, as a consequence, not visible. The only way to solve it is to introduce a further abstraction level which allows navigation on the Helicopter View. As users want to navigate on the actual visualisation and do not navigate only for the sake of using the Helicopter View, this additional abstraction might be deemed an unnecessary obstacle. So it has to be kept as simple and intuitive as possible.

5.2.1 View box on the Helicopter paper

Navigating inside the Helicopter View is made possible by the view box attribute of the paper p_h, which contains the minimised visualisation.

p_h always shares one dimension with the Helicopter View container h. If the aspect ratio of the visualisation a_v is bigger than the aspect ratio of the Heli-

copter a_h, then h and p_h have the same height. If $a_v < a_h$, h and p_h have the same width. If $a_v = a_h$, both width and height of h and p_h are identical.

The view box on p_h represents the part of p_h which is currently visible. So it has exactly the dimensions of the Helicopter View container h. When users navigate to the hidden parts of p_h, only x and y-coordinate of the view box are adjusted. As zooming p_h makes no sense, width and height of the view box are never altered.

5.2.2 Extremes of the coordinate system

Visualisations are often rendered by complex layout algorithms. Some of these place elements using only positive coordinates, while other algorithms make use of the whole coordinate system. The latter case increases the complexity for finding the right start coordinates for the view box on p_h. Setting the view box to (0, 0) would now show the middle part of p_h.

One possibility to find out the coordinates of the top left corner is to iterate over all elements of p_h and search for minimum values. Since this is very time-consuming, code was changed so that these minima, which are already known to the rendering algorithm in Java, get serialised into JavaScript and passed to the Helicopter's constructor.

As a default, the view box can now be set to the upper left corner.

Maximum values of the coordinate system are calculated by adding the paper dimensions to the minima. They are important because it does not make sense to move the view box beyond the maxima.

5.2.3 The interaction layer

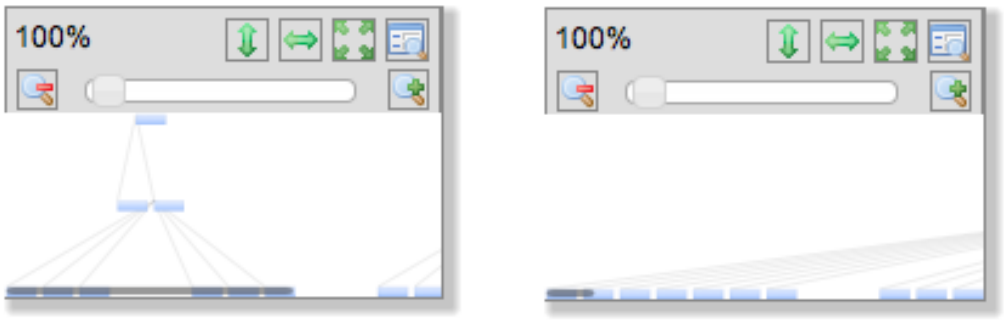
To register mouse interaction on the Helicopter View, an interaction layer is placed on top of the Helicopter. Technically, this is done by creating a further SVG paper p_i, which is placed on top of p_h. Since interaction is restricted to the Helicopter boundaries, p_i exactly matches these boundaries.

The most important task for the interaction layer is to capture mouse events. If a mouse over event close to the borders of the Helicopter View is detected, the view box on the Helicopter is panned towards this border. This Helicopter-internal panning has two speed levels. If the mouse pointer gets fifteen pixels or closer to the border, panning starts. Very close (at most six pixels) to the border, panning quickens considerably. This behaviour, which can be found in many user interfaces, helps users to scroll slowly if they want precision, and allows fast scrolling whenever long distances have to be covered.

After panning the visualisation via other means, for example pressing the arrow keys on the keyboard, p_h is updated, too. It is then placed so that the currently visible part of the visualisation is also shown inside the Helicopter.

Elements of the interaction paper

A scrollbar always shows where the view box is currently placed. It is important for users to keep track of where they are [Tid09]. This scrollbar implicitly also indicates the size of the paper p_h relative to the size of the Helicopter View. This is achieved by initialising the scrollbar size following the intercept theorem. If a_v is bigger than a_h, the Scrollbar length divided by the width of Helicopter View h must equal h.width divided by p_h.width. For rather tall visualisations ($a_v < a_h$), width has to be replaced by height.



(a) Long Scrollbar indicating a short way to scroll

(b) Short Scrollbar indicating a large visualisation

Figure 5.4: Scrollbar length indicates size of the visualisation

Example 5.4a shows a visualisation only slightly larger than the Helicopter View, while the scrollbar of example 5.4b indicates a long way to scroll and thus a very large visualisation.

While the view box on the Helicopter View is being changed, an arrow which points towards the direction being panned to is shown. It gives users visual feedback about their action. Figure 5.5 shows a pan to the right.

Apart from scrollbar and direction arrows, the most important object stored inside the interaction paper p_i is the View Frame. It is a rectangle which always represents the current view box on p_v.

Four more rectangles are laid on top of the corners of the View Frame, shown in figure 5.5. Those tiny elements listen to drag interaction. Thereby they permit resizing the View Frame.

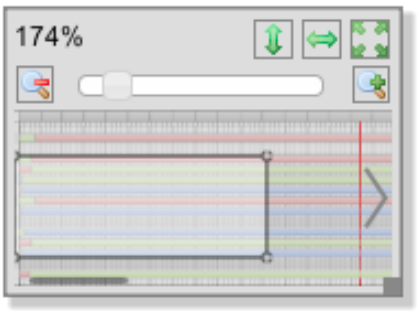


Figure 5.5: Screenshot of the View Frame and an arrow indicating a pan to the right

5.3 Interaction with the Helicopter View

Interaction with the aim of navigating on the actual visualisation is also captured on the interaction paper p_i.

5.3.1 Panning the View Frame

If a mouse down event on the View Frame inside the Helicopter View is recognised, panning mode is activated. The Helicopter View now registers to the mouse move event. The View Frame follows the mouse pointer until a mouse

up event is captured. After unregistering from the mouse move event, panning mode is deactivated, and a new view box on the visualisation paper p_v can be set. To do so, x and y-coordinates of the View Frame have to be translated to the coordinate system of p_v. Since mouse positions are registered on the interaction layer p_i, at first they have to be translated to a position on the Helicopter paper p_h. This is done by subtracting the current view box position on p_h and adding the minimum value of the view box. Figure 5.6 visualises this calculation. Finally, the resulting coordinates are multiplied with the scale factor s to set a new view box on p_v.

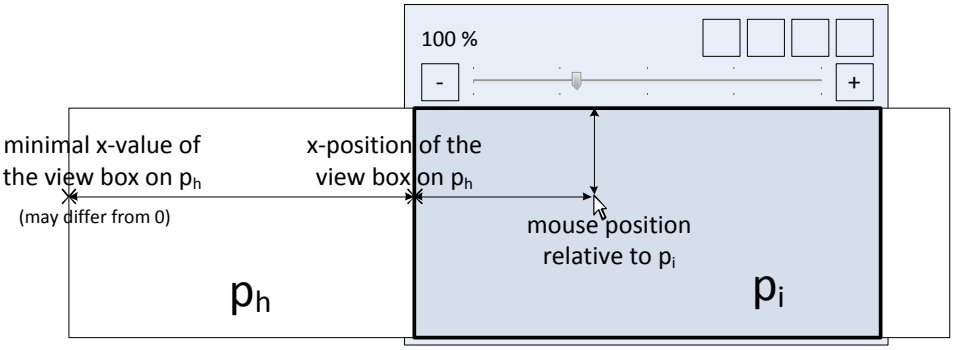


Figure 5.6: Illustration of the relation between mouse position on p_i and p_h

5.3.2 Spanning a new View Frame

If a mouse down event inside the Helicopter View but outside of the View Frame is caught, stretching mode is activated. Until a mouse up event is fired, all mouse movements now alter the size of the View Frame. If only stretching rightwards and downwards was allowed, the View Frame's x and y-coordinate could be set once to the start coordinates of the mouse movement and then left unchanged. Width and height would then be the current position of the mouse minus the start x and y-coordinates.

But allowing to stretch arbitrarily into every direction feels more intuitive and minimises wrong user interaction. Because negative width and height of a SVG element are prohibited, the calculation method explained above does not work anymore. Instead, the algorithm defines two points P_{fix} and P_{drag}. P_{fix} is initialised with the start position of the mouse movement while P_{drag} always

resembles the current position of the mouse pointer. As the name indicates, P_{fix} is the only corner of the View Frame whose position remains constant. Both coordinates of P_{drag} change continuously along with the mouse position. The two interjacent corners P_2 and P_3 each keep one dimension, whereas the other coordinate follows the mouse movement. Figure 5.7 demonstrates an exemplary situation. Initially, all points are identical. The dashed line shows the View Frame at the moment t=1, the solid line at the moment t=2. It can be seen that P_{fix} remains fix during the whole movement. P_2 steadily updates its x-coordinate and P_3 its y-coordinate. As P_{drag} follows the mouse pointer, both of its coordinates are changed.

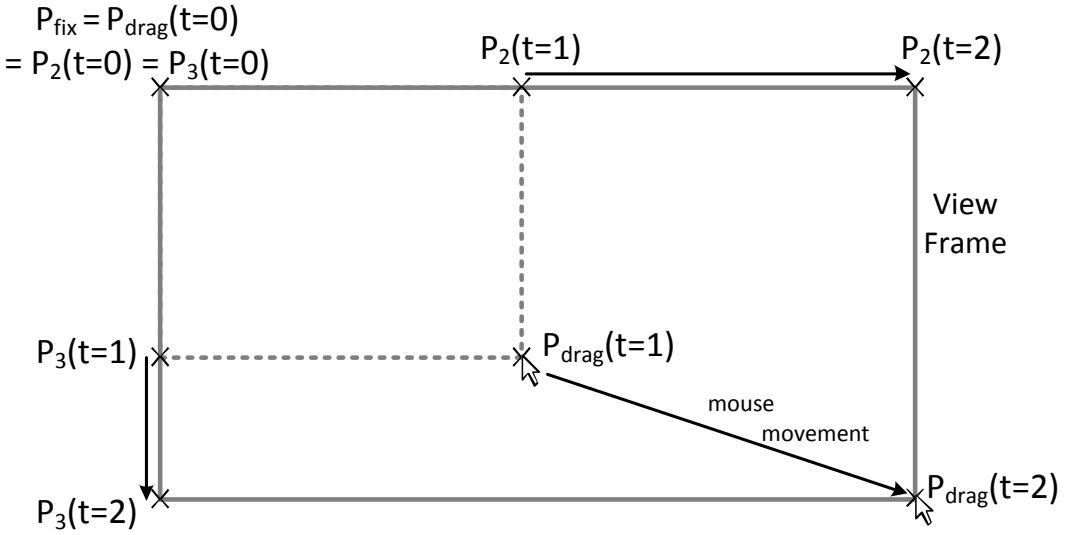


Figure 5.7: Spanning a View Frame

The View Frame's x and y-coordinates are defined to be the minimum of P_{fix} and P_{drag}. The View Frame size is set to the coordinates of the bigger of the two points minus those of the smaller one. Since width and height thus always have positive values, this algorithm works under all circumstances. Including scenarios with negative coordinates or stretching the View Frame across axes of the coordinate system.

In a last step, the new View Frame position and size get translated to dimensions of the visualisation's view box.

5.3.3 Resizing the View Frame

Resizing the View Frame gets enabled whenever users drag one of the small rectangles which represent the corners of the View Frame. P_{drag} has to be set to the corner the mouse has clicked on and which gets moved. P_{fix} is exactly the opposite corner. With these values, the algorithm for spanning the View Frame (described above in section 5.3.2) can compute a new View Frame and set the view box on the SVG.

5.3.4 Restoring defaults

Whenever a user clicks onto the Helicopter without moving the mouse between pushing the mouse button and releasing it, all pan and zoom states are reset. Zoom level is restored to 100% and the view box is placed back to the top left corner of p_h.

5.4 Navigating on a mobile device

Since no Helicopter View is shown on mobile devices, different navigation methods have to be implemented. Besides only one button for the fullscreen mode (shown in screenshot 5.8), basic touch gestures have to be supported. The latter is vital for easy and intuitive operation on touch screens like those found on all tablet computers and most modern smartphones. The most common gestures on touch screens are **swipe** for panning and **pinch** for zooming.

Figure 5.8: Button invoking the fullscreen mode on mobile devices

5.4.1 Pinch gesture

For a **pinch**, two fingers have to touch the screen. Whenever those two fingers move towards one another, content is zoomed out. If the distance between the two touches increases, zoom in is invoked.

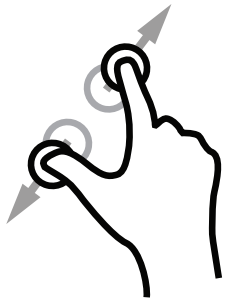


Figure 5.9: Pinch gesture on touch displays [ges12]

Browsers on touch devices support this gesture. And since all visualisations of this work are rendered as scalable vector graphics, the browser can zoom in without the graphics losing quality. But zoom levels under 100% are not possible. Thus, using the default functionality, users cannot zoom out. Still, this might be essential to take an overview of large visualisations.

In order to allow zoom levels smaller than 100%, the browser's default behaviour for the pinch gesture has to be suppressed and own event handlers have to be implemented.

5.4.2 Swipe gesture

Panning and scrolling is usually done via a stroke with one finger into one direction. As long as the finger touches the screen, the object to be panned sticks to the finger and follows all of its movements. This is called a **swipe**.

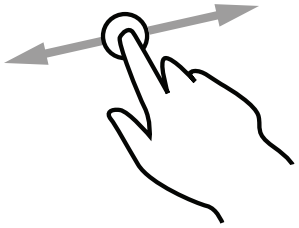


Figure 5.10: Swipe gesture on touch displays [ges12]

When a user has zoomed in as described in the previous section, the SVG container will no longer show the entire visualisation but only the zoomed

part. Then the browser's built-in swipe functionality can no longer navigate to the parts of the graphics outside the SVG container boundaries. To allow the scenario of panning to every part of a visualisation, default swipe behaviour of the browser has to be prevented. Custom code will detect swipe gestures and fire pan events.

5.4.3 Detecting touch events and gestures

A general problem is that there are still few conventions about how browsers should notify about touch events. Opera Mini (Version 7.0.1 for iOS), for example, does not forward any touch events. Devices using Safari on an iOS operating system provide event listeners which detect and forward a pinch event and its scale factor. But as most other browsers do not, gestures have to be detected on a lower abstraction level. W3C compliant JavaScript fires the events *touchstart*, *touchmove* and *touchend* [Zak09]. In one of their parameters a list of all touches on the screen along with their exact position is transferred. An algorithm detecting pinch gestures can listen to the *touchmove* event and check if exactly two fingers are touching the screen. If this is the case, on every further *touchmove* event it calculates whether the distance between those two touches has increased or decreased. Internal events to zoom by setting the SVG's view box are fired accordingly.

Gesture support in third-party libraries

Apart from writing JavaScript on the lowest abstraction level, one could of course use third-party libraries for detecting touch gestures. JQuery offers a JavaScript library especially for mobile devices, called jQuery Mobile. But this library does not detect pinch gestures. Swipes are recognised, but only horizontally. Their philosophy is to use scroll events instead of vertical swipes [Bai11]. But Apple's iOS browser forbids manipulation of DOM (document object model) elements before the scroll is finished. This impedes the usage of jQuery Mobile.

Sencha Touch 2, another JavaScript library specialised on touch devices, does support both pinch and swipe gestures. But it turned out to be too difficult to integrate into the existing code.

5.4.4 Computer response to touch interaction

A first approach wanted to achieve immediate response to user interactions and tried to set the View Box already during the move event of the finger. But browsers are quite slow at calculating a new view box of a SVG. This effort to enhance the user experience resulted in a noticeable stutter of view box updates.

An attempt to smooth the zooming process removed all shadows and fill effects of the visualisation at the beginning of a pinch or a swipe. During the gesture only the skeleton of the visualisation remained. After the gesture the original fill and shadows were restored. This decrease of visible elements led to an acceleration of setting the View Box while interacting with the device. But it still did not work fluently enough to satisfy user expectations.

Consequently, the current implementation waits for the touch move to end and sets the view box only afterwards. To give users feedback about how far they are zooming, the future zoom level will be displayed while interacting with the touch screen. A further idea to improve the visual response is to show a Helicopter View during pinch gestures. The constantly updated View Frame would then give an accurate preview of what part of the visualisation will be visible after ending the zoom interaction. Such visual response follows Nielsen's design guideline of providing feedback when an action might take longer than one second [Nie93].

5.5 Fullscreen mode

The requirements of this project demand a fullscreen mode to give a visualisation as much space on the screen as possible. This chapter describes two different functionalities implemented to meet this need.

Restrictions

Of course, all modern browsers already have a real fullscreen mode, which shows web pages on the entire screen size, overlapping even operating system controls. But due to security restrictions, this built-in fullscreen mode cannot be invoked via JavaScript.

Hence, as long as the user does not manually open this built in mode, visualisations are restricted to the inner boundaries of a browser window. Because of operating system and browser toolbars, this size is significantly smaller than the screen (illustrated in figure 5.1).

5.5.1 Fullscreen via a new browser window

One way to realize the desired fullscreen mode is to open a new browser window which shows solely the visualisation and tools needed for navigating on it. All other content of the original page is not shown on the new page. Thus, the whole browser window can be efficiently used exclusively for the visualisation. A little more space can even be gained by hiding all browser toolbars of the new window.

5.5.1.1 Implementation alternatives

Technically, there are three alternatives of how to move the visualisation into the new window:

- A **server callback** could be triggered. The newly created browser window would request a URL which indicates that only the visualisation and no other content shall be transferred. Yet consulting the server is superfluous, since all required information already exists on the client and no further server processing is necessary. Furthermore, doing a server callback would take a perceptible amount of time. Test with Sugiyama graphs containing 5x5x10 elements showed an average response time of 2220 milliseconds. As in the test setup client and server ran on the same

computer, network latency has still to be added to resemble realistic conditions. Literature [Dah06] quotes that reaction times over one second are not accepted by users.

A further disadvantage is that the navigation status (current zoom factor and pan position) cannot be saved and transferred back to the server. It will be lost when doing a server callback.

- The second alternative is to move all necessary JavaScript code to the new browser window and then **render the visualisation anew** there. The script which initialises all objects and renders the visualisation as a SVG element is stored as a node in the DOM tree. This DOM element has to be moved to the child window and executed there. With an average execution time of 1414 milliseconds (on a Sugiyama map with 5x5x10 nodes) this procedure is way faster than any server response. But it still exceeds the maximum response time defined in the non-functional requirements in chapter 3.2.3.

  Like in the server callback alternative, all information about the current navigation status will be lost here, too, because re-rendering resets everything and there is yet no way to transfer information about navigation. Moreover, all libraries the visualisation relies on have to be transferred manually to the child window.

- The last approach is to search for the elements containing the visualisation and navigation tools and simply **move these DOM elements** into the child window. Shifting the visualisation's SVG, the navigation's container element, as well as all of their deeply nested children to the new window is only a move operation in the computer memory. It does not require expensive processor computations and is hence faster than re-rendering the visualisation. The tested Sugiyama graph showed up after an average time of 649 milliseconds. According to Dahm [Dah06] this is still perceived as spontaneous by the end user.

  Another benefit is that all changes manually done to the visualisation, for example zoom and pan status, remain unaffected, because mere moving operations do not manipulate the DOM elements. But, like in the previous alternative, libraries have to be moved to the child window.

5.5.1.2 Moving DOM elements to the child window

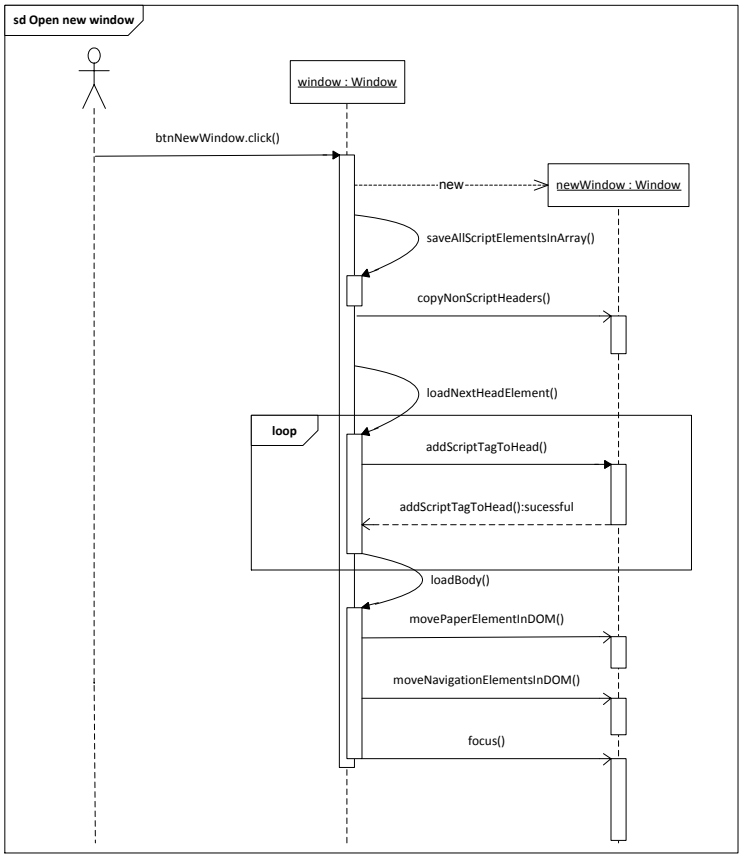


Figure 5.11: Sequence diagram about opening a new window

Because of its advantages, the latter of the described options was implemented in this work. An exact description of the approach of moving DOM elements to the new window works follows below. Sequence diagram 5.11 shows an overview of all important steps.

When a user clicks the button to open a new window, the click event first creates an instance of a child window.

A few preconditions have to be fulfilled. The visualisation depends on various JavaScript libraries including third party frameworks like jQuery and Raphaël. When requesting a site from the server, these libraries are transferred and executed before any content [Zak09]. Other important elements loaded prior to the content are cascading style sheets (CSS). They define style and appearance of HTML elements. Both libraries and CSS files are included in the Head tag

of the HTML document. So at first the complete Head element of the parent window needs to be copied to the child in order to provide the foundation the visualisation is built on.

JavaScript libraries usually have complex dependencies among themselves. Central libraries like Raphaël and the jQuery core have to be loaded first. Others, for example jQuery plugins, rely on the existence of those core scripts. These plugins might be the basis of yet other libraries.

After a normal HTTP request, the server automatically transmits all scripts in the right order. So the order in which entries appear in the HTML head resembles their dependencies. It must be ensured that this order remains the same in the child window.

In practice, sticking to the right order exposes subtle problems. A naive approach would simply iterate through the parent's head entries and create a new DOM element in the child's head every time. But the actual loading of scripts in the child window is done asynchronously, causing shorter libraries to be interpreted before longer ones they require are loaded completely. Every time a script calls functions which do not exist because they are part of a library not completely loaded yet, JavaScript throws errors. After an error, script execution cannot be continued. Violating script dependencies consequently results in a lot of errors and the visualisation not working.

The solution for this problem is an implementation where every script element is provided with a callback function [Fla98]. This callback is done after the library has been loaded completely. It is a function pointer to a method of the parent window which triggers the next copying step. Thereby the right order of script entries in the head is assured.

The sequence diagram 5.11 shows how the new window gets initialised. When iterating through the window's head, script sources are temporarily stored in an array. As CSS files do not have dependencies to one another nor to script files, they are transferred immediately. Then, the described script loading algorithm is triggered. Communication in the loop should symbolise that the next copying step can only be started after an answer of the script copied before.

Table 5.1: Measured response time in a large Sugiyama graph (5 nodes in the first level; each having 5 second level nodes times 10 nodes in the third level) in milliseconds

	move in DOM tree	re-rendering JavaScript	server callback
Mean	649	1414	2220
Median	645	1395	2178
Std. Deviation	75	74	162

After the head of the child window is complete, content of the body can be loaded. In this step the visualisation and the container of all navigation tools have to be moved to the child. These two elements are detached (i.e. cut out of the DOM tree) and appended under the body tag of the child HTML document. As the DOM tree follows the principle of nested XML, moving the container element always includes all children in the tree hierarchy.

The last step is to pass the focus to the child window. It is then active and visible on top of all other application windows.

5.5.1.3 Performance of the discussed alternatives

To get reliable results of how fast each of the three implementation alternatives described in the previous chapter works, extensive performance tests were run. An automated routine triggered 1000 runs for every test and logged the total response time for each run. A detailed report about the testing procedure can be read in section 5.5.1.4.

Two different visualisation types were tested:

- A big **Sugiyama graph** consisting of three levels. Every one of the five first level nodes is connected to five nodes of the second level. Each of the latter has 10 more third level child nodes.

  Table 5.1 lists average time, median and standard deviation of all implementation options. Figure 5.12 shows the corresponding boxplot. Medians are painted as horizontal lines for better legibility.

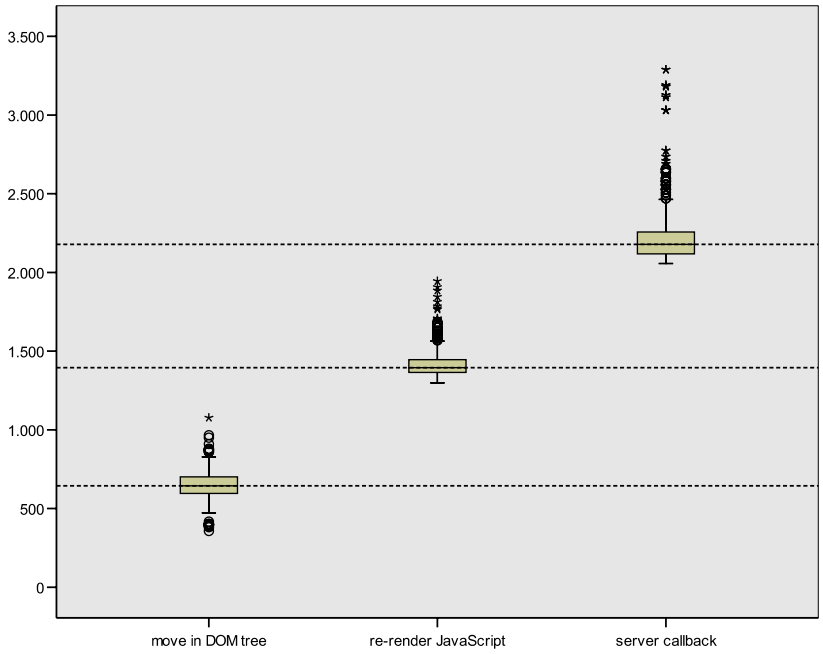


Figure 5.12: Boxplot illustrating the performance of the implementation alternatives on a Sugiyama map

- A medium-sized **cluster map** with eight outer elements, each of which contains eight inner elements. Table 5.2 and diagram 5.13 visualise the performance test results of this map.

As already mentioned, re-rendering the visualisation is obviously nearly always faster than doing a server callback. But moving the visualisation's container element in the DOM is even considerably faster than the re-rendering option.

When comparing the server callback results to the other two, the fact that network latency would be necessary for realistic circumstances has to be kept in mind.

On both maps a relatively high number of outliers can be noticed. For the alternatives of re-rendering and consulting the server those outliers are always higher than all other values. One explanation is that sometimes the processing time scheduler had to grant threads with higher priority (e.g. operating system threads) more processing time. Another factor which explains those outliers is the allocation of RAM memory to the browser application. It turned out that

Table 5.2: Measured response time in a medium-sized cluster map (8 outer elements, each containing 8 inner elements) in milliseconds

	move in DOM tree	re-rendering JavaScript	server callback
Mean	433	575	712
Median	433	535	702
Std. Deviation	54	131	47

with each further test run the browser demands more memory than it frees. Thus, the browser gets steadily slower.

Extreme outliers in the re-rendering alternative on the cluster map and the server callback alternative on the Sugiyama also lead to a high standard deviation. The standard deviation of all other tests ranges from 47 to 75 milliseconds, which is a rather good value, since this time interval difference is hardly noticeable.

A further indicator for a good quality of the test values is the small difference between mean and median in all tested cases.

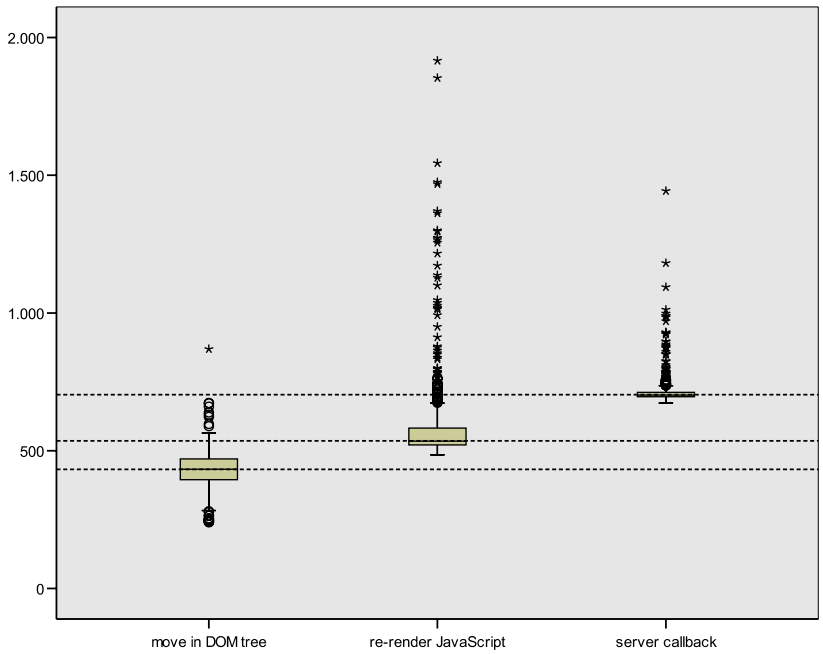


Figure 5.13: Boxplot illustrating the performance of the implementation alternatives on a cluster map

5.5.1.4 Measuring method of the performance tests

All performance tests were executed on a laptop with a 2,66 GHz dual core processor and 4 GB RAM running Mac OS X 10.6.8. Both server application and client (Safari 5.1.2 browser) simultaneously ran on this machine.

Tests were triggered automatically by the routine shown in listing 5.1.

Listing 5.1: Methods for performance tests

```
1  testNewWindow = function() {
2      if(testRun <= 100) {
3        startTime = new Date().getTime();
4        openChildWindow();
5        testRun++;
6      }
7  };
8  finishTestNewWindow = function(end) {
9      console.log((end - startTime));
10     closeChildWindow();
11 };
```

The method *testNewWindow* saves the system time before each test. Then a child window is opened. After rendering is completed, system time is fetched again and passed back to the function *finishTestNewWindow* of the parent window. This callback function logs the time difference to the console and closes the child window. After closing the window, a new test is triggered.

It turned out that after about 150 runs, with each further test the browser demanded more memory and thus steadily became considerably slower. But since users will hardly open a new window over a hundred times, it is justified to eliminate this effect in order to get more realistic performance measures. So, after one hundred runs, the routine was stopped and the browser restarted. This procedure was repeated ten times to get a total number of 1000 runs.

5.5.2 Fullscreen in a dialog window

The second way to implement a fullscreen mode is to show the visualisation in a dialog box. This dialog fills the whole browser window and is laid on top of all other content, showing only the visualisation and navigation tools.

This concept is implemented by using a standard dialog of the jQuery user interface library. Visualisation and navigation DOM elements are moved into the container element of the jQuery dialog. When closing the dialog, they are moved back to their original parent element.

5.5.3 Comparison between dialog and new window

Showing the visualisation in a new window is much more complex than in a dialog. Opening a new browser window takes about 35ms[1]. Then all header files need to be copied to the child window (approximately 320ms[1]). Only the last step is identical in both approaches. By moving elements in the DOM tree, the same fast concept is applied in the dialog as well as in the "new window" functionality.

Summing up, opening the visualisation in a new window is much slower than creating a dialog. Regardless of the map type, tests showed a total time difference of about 350ms[1].

However, opening a new window also has an advantage. All browser toolbars can be suppressed, whereby further display space is gained.

Since none of the two described approaches is clearly superior to the other, users may decide which one is more to their liking. A dialog is faster, whereas opening a new window can use more space on the screen for a visualisation.

[1] on the test computer described in section 5.5.1.4

6 Testing

All code of this work is written in JavaScript and executed on the client. This makes testing rather difficult.

Most of the functionality is invoked by user interactions like clicking on a button. Simulating mouse clicks and movements is often complicated.

The result of an operation is also difficult to determine. Most interactions here manipulate graphics on the web page. But the computer does not see the outcome and cannot distinguish between correct and incorrect visualisations.

Despite those obstacles, a try to test basic functionality via unit tests was made. Units are functions or code modules at a very basic level [MK07]. Unit testing should ensure that each of these individual units "is functioning according to its specification" [Bur03].

For test automation, the framework QUnit was chosen. It was originally developed to test code of the jQuery project, but is capable of testing any generic JavaScript code [jQu12].

It turned out that even complex actions of the mouse could be performed automatically with the help of a further jQuery library called *"jquery.simulate"*. It provides mechanisms to create artificial JavaScript event objects and to interact with user controls. Listing 6.1 shows a method which tests the functioning of the slider inside the navigation component.

Listing 6.1: Unit test of the slider inside the navigation component

```
1  test("Changing the slider fires eve event", function() {
2      expect(1);
3      var testZoomSlider = function(event, delta) {
4          notEqual(delta,0);
5      }
6      paper.events.registerEvent("*.zoom", testZoomSlider);
7
8      var slider = $("#zoomSlider"+paper.id).css("height", "5px");
9
10     var e1 = jQuery.Event("mousedown", {
11         pageX : slider.offset().left + 1,
12         pageY : slider.offset().top + 1
13     });
14     var e2 = jQuery.Event("mousemove", {
15         pageX : slider.offset().left + 10,
16         pageY : slider.offset().top + 1
17     });
18
19     slider.simulate("mousedown", e1);
20     slider.simulate("mousemove", e2);
21
22     paper.events.unregisterEvent("*.zoom", testZoomSlider);
23 });
```

QUnit tests are declared by calling the method *test* with a description and the testing routine as arguments. Line 2 tells the framework that exactly one assertion is expected to be reached. In line 6, the method *testZoomSlider* is registered to listen for all zoom events. If this function gets called, the *notEqual* statement asserts that the zoom delta is not zero. In this case the test case is considered to be successful.

The following lines simulate a click on the left side of the slider and, with the mouse button still pressed, a mouse move towards the centre. This manipulates the slider handle and should therefore trigger a zoom event. If it does not, the method *testZoomSlider* never gets called, the assertion is not reached, and QUnit will mark the test as failed.

This example tests the process from user interaction with controls on the web site until a zoom event is thrown. However, it does not test whether the zoom

event gets interpreted correctly by all components, nor if the SVG is zoomed correctly.

A few test cases were implemented in that way. They allowed to apply the methodology of regression testing. Thereby, all tests can be run after every modification of the program in order to verify that these changes "have not caused unintended effects" [IEE90, IEE86]. It is checked if all functionality which had already worked still operates soundly.

Bug in Firefox

Further testing revealed a bug concerning SVG in the Firefox web browser. Whenever a SVG element is moved in the DOM tree, all colour gradients turn black. This bug is reported [Goo12] and will hopefully be fixed in one of the next updates of Firefox.

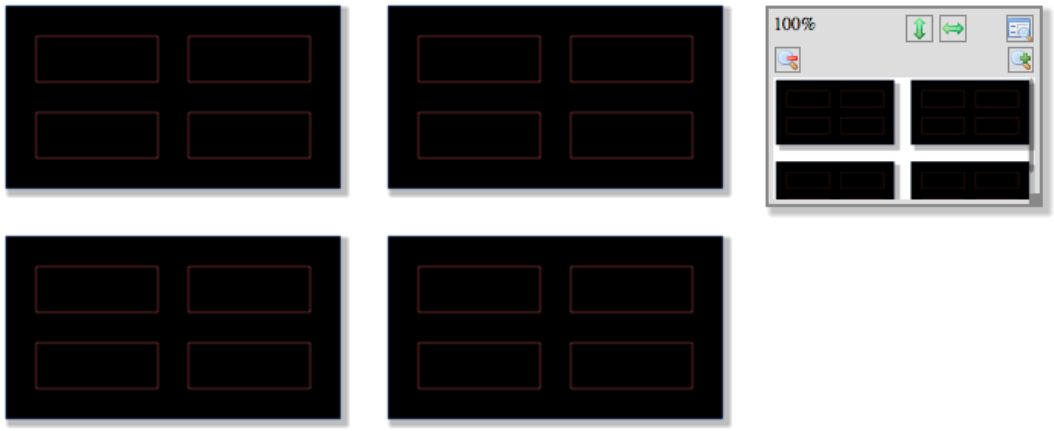


Figure 6.1: Visualisation in Firefox after moving the SVG container in the DOM tree

7 Related work

The following chapter describes a few existing products providing rich graphical interaction interfaces, ordered by popularity. It concludes with a summary of important scientific research related to this topic.

7.1 Google Maps

Figure 7.1: Navigation controls on Google Maps

Google Maps is a web application which basically shows an extremely large and detailed map of the whole world. To keep it usable, good navigation possibilities are of paramount importance. Google therefore shows buttons to pan the map and a slider to change the zoom factor (shown in figure 7.1). Apart from those visible components it offers the alternative to zoom via the mouse wheel. Panning can also be done using the arrow keys or by dragging the map with the mouse.

One clever detail is that the slider for zooming is being scaled down when reducing the size of the browser window significantly.

All interaction feels intuitive. However, users might lose track of where on the map they are when zooming in very far. A Helicopter View would solve that problem.

7.2 Microsoft Visio

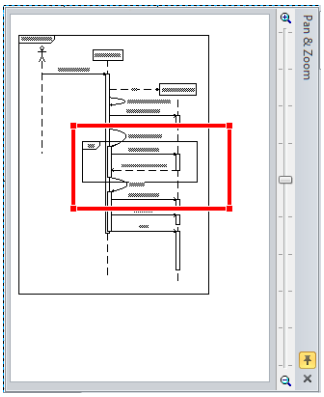


Figure 7.2: Microsoft Visio's Pan & Zoom Window

Microsoft Visio, as a program for painting diagrams, has also need for easily usable navigation functionality on visualisations. Visio, like Google Maps, provides various ways of zooming and panning. A big difference to Google and most graphic processing applications is, that turning the mouse wheel scrolls the document instead of zooming it.

Visio goes one step further than Google Maps and offers a "Pan & Zoom Window", following the same concept as the Helicopter View in this work. Visio's Helicopter shares many features and interaction mechanisms with the one introduced here.

One difference is the behaviour after a click on the Helicopter View but outside the View Frame. Visio does not reset the zoom level to 100%, but instead pans the View Frame until the mouse pointer becomes its centre. As Visio diagrams usually do not reach the vast dimensions a software map might have, zooming

is not of such importance as in the GraphicalVisualizations plugin. So this difference in interaction behaviour is basically a result of distinct requirements.

Another difference shows up when spanning or resizing the View Frame. Visio forces the View Frame to exactly match the aspect ratio of the document. Users might feel confused about that when the corner they are dragging does not always follow the mouse pointer.

Eventually, Visio has a big performance advantage compared to the GraphicalVisualizations plugin. Since it is a full sized client application it can use much more processing power of the computer than a JavaScript web-application inside a browser.

7.3 Adobe Photoshop

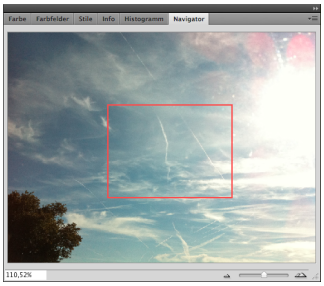


Figure 7.3: Navigator in Adobe Photoshop

Adobe Photoshop gets shipped with a Helicopter View, too. Its aim is to ease the handling of large pixel graphics. Like Visio it is a client application and does not show any performance weaknesses. Adobe calls its Helicopter View "Navigator" (shown in figure 7.3).

One notable characteristic is that Photoshop makes switching between a custom zoom level and 100% extremely easy. A simple click on the View Frame saves the current zoom level and then transfers the graphics to a level of 100%. A further click anywhere on the Helicopter restores the saved zoom level. This is particularly useful to make a quick overview of the entire picture and then resume working on the excerpt zoomed in before. All this only takes two clicks.

But a View Frame in Photoshop cannot be resized nor can users span a new View Frame.

Unlike the current implementation of the Tricia GraphicalVisualizations Helicopter (explained in chapter 5.1), Photoshop forces the whole picture to fit into their "Navigator". An extreme aspect ratio as shown in screenshot 7.4 might lead to reduced usability.

Figure 7.4: Navigator in Adobe Photoshop on a picture with an extreme aspect ratio

7.4 SyCaTool

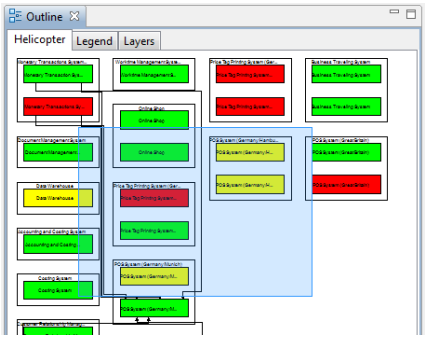


Figure 7.5: Helicopter View in the SyCaTool

A Helicopter View is also part of the SyCaTool, a fat client application for software cartography developed at the SEBIS chair. The SyCaTool was the predecessor of Tricia's GraphicalVisualizations plugin, on which this work is based.

The Helicopter there supports basic functionality similar to Microsoft Visio. Differences exist in the way the arrow keys work. Instead of panning they trigger zoom events.

7.5 EMT Valencia

Figure 7.6: Button controls and Helicopter View on the EMT Valencia web site

One of the very few Helicopter Views found on the internet is part of a map which shows Valencia's metro and bus stations. The web site of the Valencian public transport company EMT offers the same interaction controls as Google Maps plus a Helicopter View (shown in figure 7.6). Special about this Helicopter View is the fact that it does not show the whole map of the city but only parts of it when zooming in. The part of the map shown in the Helicopter is always only slightly bigger than the View Frame. If users zoom in further and the View Frame would get too small to handle it properly, EMT simply zooms in the part of the map shown inside the Helicopter until the View Frame becomes big enough to be usable again. With this mechanism developers solved the problem of a very small View Frame on a very large map.

7.6 Comparison of selected navigation components

Table 7.1 compares the Helicopter View of this work with its predecessor and the two most widespread proprietary applications which ship with similar navigation components.

Table 7.1: Comparison of selected navigation components

	Microsoft Visio	**Adobe Photoshop**	**SyCaTool**	**Tricia Graphical Visualizations**
Architecture	Fat client	Fat client	Fat client	Web application
Name of the zoom window	Pan & Zoom window	Navigator	Helicopter View	Helicopter View
Helicopter is resizable	✓	✓	✓	✓
View frame	✓	✓	✓	✓
Pan by dragging the View Frame	✓	✓	✓	✓
Pan using arrow keys	✗	✓	✗	✓
Slider	✓	✓	✗	✓
Buttons for zooming	✓	✓	✗	✓
Mouse wheel functionality	Pan	Pan (Zoom when Alt is pressed)	none	Zoom
Mouse click functionality	Pan until that point is centred	Switch between custom zoom level and 100%	Pan until that point is centred	Reset zoom to 100%
View Frame is resizable	✓	✗	✗	✓
Span new View Frame	✓	✗	✗	✓
View Frame independent of aspect ratio	✗	✗	✗	✓

7.7 Related scientific research

Jerding and Stasko called their Helicopter View *Information Mural* [JS95]. It is used as a global view of more detailed information displays. For the generation of the Information Mural, anti-aliasing and grayscale shading techniques were applied. Compared to most other Helicopter Views the Information Mural is not restricted to classic visualisations. Instead, as shown in figure 7.7, it can also give an overview of text documents.

Figure 7.7: Information Mural for a LaTeX document [JS95]

Tominski et al. [TAS09] developed navigation mechanisms especially for graph layouts. They introduced *pan-wheel interaction* and *edge-based traveling*. The former allows users to pan via clicking and dragging the mouse into one direction at the desired speed. Edge-based traveling facilitates jumping from one node of a graph to an adjacent node.

Blanch and Lecolinet proposed the concept of *zoomable treemaps* for navigation in large hierarchical data sets [BL07]. Their work expands the possibilities of browsing through hierarchies visualised via treemaps.

Research has also been done about the question how navigation can be done smoothly [WN04, CSW05]. Van Wijk et al. came to the conclusion that for panning a long distance the virtual camera should first zoom out and do the panning on a high level before zooming back in to the original level.

Cecconi and Galanda criticise the cartographic quality of current web mapping applications [CG02]. They postulate adaptive zooming. Currently, maps on

the web have few defined levels of detail. But really intuitive adaptive zooming would “calculate every arbitrary desired scale” [CG02].

The InfoVis Toolkit, a framework for visualising data structures, uses *fisheye lenses* to magnify parts of graphics [Fek04]. It enlarges the area of interest while other parts remain less detailed. As shown in figure 7.8, this results in distorted pictures. But viewers “can become disoriented by nonlinear distortions” [HS12, NBM+06]. Despite that, the fisheye technique in general is common both in literature (inter alia [RMG07, STST98]) and in applications such as the Mac OS X dock.

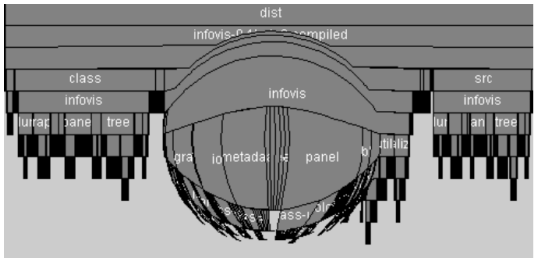


Figure 7.8: Fisheye lens in InfoVis [Fek04]

The University of Rostock developed yet a further navigation functionality on graphs. Their approach “maps a hierarchy graph onto the surface of a hemisphere” [KS99]. A projection then helps to focus the desired area of the graph. This *Magic Eye View* is shown in illustration 7.9.

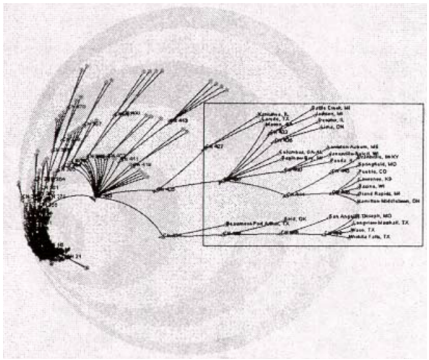


Figure 7.9: Magic Eye View [KS99]

8 Summary, conclusion and outlook

The rising demand for Enterprise Architecture documentation [BEG+12] is gradually being addressed by tools like the plugin for System Cartography of the SEBIS chair at the TU München. However, until now generated visualisations were primarily static and did not provide means for interaction. This work introduced components for interactive navigation on large visualisations.

It pursued the goal of enabling efficient navigation on various different devices. Stationary computers profit from a Helicopter View, which gives an overview of a visualisation and allows users to perform navigation tasks quickly and precisely. Right now, only few applications provide such mighty navigation interaction possibilities. The novelty also lies in integrating this rich user interface into a web application. Users do not have to install a new application, but can access all functionality in any ordinary internet browser.

Mobile devices like tablet computers demand slightly different interaction mechanisms. Small screens make sophisticated controls like the Helicopter View unmanageable. But finger gestures on touch screens compensate for this disadvantage and offer many new ways of user interaction. One example of future technology within this field of research is a patent Google filed recently [Pat12]. It suggests to invoke a Google search when users write a "g" with a finger onto the screen.

Apart from such advanced technology the trend will hopefully lead towards standardisation of the recognition of basic touch gestures. This would facilitate development significantly. A research team around F. Echtler took a first step into that direction and promoted a "Unified Gesture Description Language" [EKB10].

A leap in standardisation is HTML 5, which has yet to become widely adopted. Apart from the one main user interface thread, HTML 5 introduced a concept called *web workers*. Web workers allow to execute code outside the UI thread and thus offer the possibility to run JavaScript applications on multiple threads simultaneously [Zak10]. Clever employment of this concept might alleviate some of the unsolved performance problems mentioned in previous chapters.

This work is a first little step within the field of graphical interaction. Apart from navigation, many further interaction mechanisms can be useful. Updating views, changing existing items and relationships or even creating new ones is highly relevant as technology advances.

The developed prototype is currently evaluated within the Wiki4EAM community [SEB12b] and industry projects. First end-user user response has been positive. Further research based on this feedback is in progress.

List of Abbreviations

ANSI	American National Standards Institute
CSS	Cascading Style Sheet
DOM	Document Object Model
EA	Enterprise Architecture
EAM	Enterprise Architecture Management
HTML	HyperText Markup Language
ID	Identificator
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Organization for Standardization
IT	Information Technology
JS	JavaScript
RAM	Random-Access Memory
SEBIS	Chair for Software Engineering for Business Information Systems at TUM
SVG	Scalable Vector Graphics
TUM	Technische Universität München
UI	User Interface
URL	Uniform Resource Locator
VML	Vector Markup Language
W3C	World Wide Web Consortium
XML	Extensible Markup Language

Bibliography

[App12] Apple: *iOS Human Interface Guidelines*. `http://developer.apple.com/library/ios/DOCUMENTATION/UserExperience/Conceptual/MobileHIG/Characteristics/Characteristics.html`. Version: April 2012, Last checked: 5.4.2012 14:50

[Bai11] Bai, Giulio: *jQuery Mobile - First Look*. Packt Publishing, 2011

[BBF+12] Berneaud, Marcel ; Buckl, Sabine ; Fuentes, A. ; Matthes, Florian ; Monahov, Ivan ; Nowobilska, Aneta ; Roth, Sascha ; Schweda, Christian M. ; Weber, U. ; Zeiner, Monika: Trends for Enterprise Architecture Management and Tools. 2012. – Technical report

[BEG+12] Buschle, Markus ; Ekstedt, Mathias ; Grunow, Sebastian ; Hauder, Matheus ; Roth, Sascha ; Matthes, Florian: Automating Enterprise Architecture Documentation using Models of an Enterprise Service Bus. In: *Americas Conference on Information Systems (AMCIS)*, 2012

[BEL+07] Buckl, Sabine ; Ernst, Alexander M. ; Lankes, Josef ; Schneider, Kathrin ; Schweda, Christian M.: A Pattern based Approach for constructing Enterprise Architecture Management Information Models. In: *8. Internationale Tagung Wirtschaftsinformatik* (2007), S. 145–162

[BGS10] Buckl, Sabine ; Gulden, Jens ; Schweda, Christian M.: Supporting ad hoc Analyses on Enterprise Models. In: *5th International Workshop on Enterprise Modeling and Information Systems Architectures*, 2010

[BL07] Blanch, Renaud ; Lecolinet, Éric: Browsing zoomable treemaps: structure-aware multi-scale navigation techniques. In: *IEEE Transactions on Visualization and Computer Graphics* 13 (2007), No. 6, S. 1248–1253

[BMM+11] Buckl, Sabine ; Matthes, Florian ; Monahov, Ivan ; Roth, Sascha ; Schulz, Christopher ; Schweda, Christian M.: Enterprise Architecture Management Patterns for Enterprise-wide Access Views on Business Objects. In: *European Conference on Pattern Languages of Programs (EuroPLoP)*, 2011

[BMR+10a] Buckl, Sabine ; Matthes, Florian ; Roth, Sascha ; Schulz, Christopher ; Schweda, Christian M.: A Conceptual Framework for Enterprise Architecture Design. In: *Trends in Enterprise Architecture Research - 5th International Workshop*, 2010, S. 44–56

[BMR+10b] Buckl, Sabine ; Matthes, Florian ; Roth, Sascha ; Schulz, Christopher ; Schweda, Christian M.: A method for constructing enterprise-wide access views on business objects. In: *Informatik 2010: IT-Governance in verteilten Systemen*, 2010

[Bro87] Brown, C. M.: *Human-Computer Interface Design Guidelines.* Ablex Publishing Corporation, 1987

[bro93] *Brockhaus - Enzyklopädie.* F. A. Brockhaus, 1993 (21)

[Bur03] Burnstein, Ilene: *Practical Software Testing.* Springer Science+Business Media, 2003

[CG02] Cecconi, A ; Galanda, M: Adaptive Zooming in Web Cartography. In: *Computer Graphics Forum* 21 (2002), No. 4, S. 787–799

[CSW05] Cockburn, Andy ; Savage, Joshua ; Wallace, Andrew: Tuning and testing scrolling interfaces that automatically zoom. In: *Proceedings of the SIGCHI conference on Human factors in computing systems CHI 05* (2005), S. 71

[Dah06] Dahm, Markus: *Grundlagen der Mensch-Computer Interaktion.* Pearson Studium, 2006

[EKB10] Echtler, Florian ; Klinker, Gudrun ; Butz, Andreas: Towards a Unified Gesture Description Language. In: *13th International Conference on Humans and Computers*, 2010

[Fek04] *Chapter* The InfoVis Toolkit. In: Fekete, J. D.: *IEEE Symposium on Information Visualization*. IEEE, 2004, 167–174

[Fla98] Flanagan, David: *JavaScript*. O'Reilley, 1998. – 222–225 S.

[Fri08] Friendly, Michael: Milestones in the history of thematic cartography, statistical graphics, and data visualization. In: *Seeing Science: Today* American Association for the Advancement of Science, 2008

[ges12] *GestureWorks*. `http://gestureworks.com/features/open-source-gestures/`. Version: 2012, Last checked: 14.4.2012 21:35

[GHJV95] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns*. Addison-Wesley, 1995

[Goo12] GoogleGroups: *Raphaël JS*. `http://groups.google.com/group/raphaeljs`. Version: 2012, Last checked: 15.4.2012 19:50

[HMM00] Herman, I ; Melancon, G ; Marshall, M S.: Graph visualization and navigation in information visualization: A survey. In: *IEEE Transactions on Visualization and Computer Graphics* 6 (2000), No. 1, S. 24–43

[HMRS12] Hauder, Matheus ; Matthes, Florian ; Roth, Sascha ; Schulz, Christopher: Generating dynamic cross-organizational process visualizations through abstract view model pattern matching. In: *Architecture Modeling for Future Internet enabled Enterprise*, 2012

[HS12] Heer, Jeffrey ; Shneiderman, Ben: Interactive Dynamics for Visual Analysis A taxonomy of tools that support the fluent and flexible use of visualizations. In: *ACM Queue* 10 (2012), No. 2, S. 30:30 – 30:55

[IEE86] IEEE: *Standard 1008-1987 for Software Unit Testing*. IEEE Computer Society, 1986

[IEE90] IEEE: *Standard Glossary of Software Engineering Terminology*. IEEE Computer Society, 1990

[IEE00] IEEE: *Standard 1471-2000: Recommended Practice for Architectural Description of Software-Intensive Systems*. IEEE Computer Society, 2000

[IEE07] IEEE: *ISO/IEC Standard 42010: Systems and software engineering — Recommended practice for architectural description of software-intensive systems*. IEEE Computer Society, 2007

[IEE11] IEEE: *ISO/IEC/IEEE Standard 42010: Systems and software engineering - Architecture description*. IEEE Computer Society, 2011

[inf11] INFOASSET: *Tricia Hybrid Wikis*. `http://infoasset.de/wikis/infoasset/tricia-hybrid-wikis`. Version: April 2011, Last checked: 8.4.2012 18:50

[jQu12] JQUERY: *QUnit*. `http://docs.jquery.com/QUnit`. Version: March 2012, Last checked: 27.3.2012 14:30

[JS95] JERDING, D F. ; STASKO, J T.: The information mural: a technique for displaying and navigating large information spaces. In: *Proceedings of Visualization 1995 Conference* (1995), S. 43–50,

[Kel07] KELLER, Wolfgang: *IT-Unternehmensarchitektur*. dpunkt.verlag, 2007

[KS99] In: KREUSELER, Matthias ; SCHUMANN, Heidrun: *Information visualization using a new focus+context technique in combination with dynamic clustering of information space*. ACM Press, 1999, 1–5

[Lan05] LANKHORST, Marc: *Enterprise Architecture at Work*. Springer-Verlag, 2005

[Lee99] Lee, Y T.: Information Modeling: From Design to Implementation. In: *Proceedings of the Second World Manufacturing Congress*, Citeseer, 1999, S. 315–321

[LMW05a] *Chapter* Architekturbeschreibung von Anwendungslandschaften: Softwarekartographie und IEEE Std 1471-2000. In: Lankes, Josef ; Matthes, Florian ; Wittenburg, André: *Software Engineering 2005*. Köllen Druck+Verlag, 2005

[LMW05b] *Chapter* Softwarekartographie als Beitrag zum Architekturmanagement. In: Lankes, Josef ; Matthes, Florian ; Wittenburg, André: *Unternehmensarchitekturen und Systemintegration*. Vol. 3. Gito, 2005

[LMW07] *Chapter* Exkurs Softwarekartographie. In: Lankes, Josef ; Matthes, Florian ; Wittenburg, André: *IT-Unternehmensarchitektur*. dpunkt.verlag, 2007

[Mas05] Masak, Dieter: *Moderne Enterprise Architekturen*. Springer-Verlag, 2005

[Mat08] *Chapter* Softwarekartographie. In: Matthes, Florian: *Informatik-Spektrum*. Springer-Verlag, 2008, S. 527–536

[Mil68] Miller, Robert B.: Response Time in Man-Computer Conversational Transactions. In: *Proceedings of the AFIPS Joint Computer Conference*, 1968

[MK07] Mustafa, K. ; Khan, R.A.: *Software Testing - Concepts and Practices*. Alpha Science, 2007

[NBM+06] Nebrasovski, Dmitry ; Bodnar, Adam ; McGrenere, Joanna ; Guimbretière, Francois ; Munzer, Tamara: An Evaluation of Pan&Zoom and Rubber Sheet Navigation with and without an Overview. In: *Proceedings of the SIGCHI conference on Human Factors in computing systems CHI 06* (2006), S. 11

[Nie93] Nielsen, Jakob: *Usability Engineering*. Academic Press, 1993

[Pat12] PatentlyApple: *Google search gesture*. `http://www.patentlyapple.com/patently-apple/2012/02/google-invents-an-original-search-gesture-for-future.-devices.html`. Version: February 2012, Last checked: 2.4.2012 20:20

[PCS95] Plaisant, Catherine ; Carr, David ; Shneiderman, Ben: Image-Browser Taxonomy and Guidelines for Designers. In: *IEEE Software* 12 (1995), S. 21–32

[RMG07] Reinhard, Tobias ; Meier, Silvio ; Glinz, Martin: An Improved Fisheye Zoom Algorithm for Visualizing and Editing Hierarchical Models. In: *Requirements Engineering* (2007), No. Rev, S. 9

[RPK+11] *Chapter* Optimized GUI Generation for Small Screens. In: Raneburger, David ; Popp, Roman ; Kavaldjian, Sevan ; Kaindl, Hermann ; Falb, Jürgen: *Model-Driven Development of Advanced User Interfaces*. Springer-Verlag, 2011, S. 107–122

[Sch06] Schweda, Christian M.: *Architektur eines Visualisierungswerkzeugs für Anwendungslandschaften - Anforderung und Realisierung von Kernkompetenzen*, TU München, Diplomarbeit, 2006

[SEB12a] SEBIS chair (Ed.): TU Munich: SEBIS chair, 2012. `http://wwwmatthes.in.tum.de/wikis/sycastore/cluster-map`, Last checked: 14.4.2012 14:35

[SEB12b] SEBIS chair (Ed.): TU Munich: SEBIS chair, 2012. `http://wwwmatthes.in.tum.de/wikis/sebis/wiki4eam`, Last checked: 14.4.2012 23:25

[SMR12] Schaub, Michael ; Matthes, Florian ; Roth, Sascha: Towards a Conceptual Framework for Interactive Enterprise Architecture Management Visualizations. In: *Modellierung*, 2012

[STST98] Shizuki, B ; Toyoda, M ; Shibayama, E ; Takahashi, S: Visual patterns+multi-Focus fisheye view: an automatic scalable visualization technique of data-Flow visual program execution.

In: *Proceedings 1998 IEEE Symposium on Visual Languages Cat No98TB100254* (1998), S. 270–277

[STT81] Sugiyama, Kozo ; Tagawa, Shojiro ; Toda, Mitsuhiko: Methods for Visual Understanding of Hierarchical System Structures. In: *IEEE Transactions on Systems, Man and Cybernetics* (1981)

[Sug02] Sugiyama, Kozo: *Graph Drawing and Applications for Software and Knowledge Engineers*. World Scientific Publishing, 2002

[TAS09] Tominski, Christian ; Abello, James ; Schumann, Heidrun: Two Novel Techniques for Interactive Navigation of Graph Layouts. In: *proc EuroVis EurographicsIEEE Symposium on Visualization* (2009), S. 1

[Tid09] Tidwell, Jenifer: *Designing Interfaces*. O'Reilley, 2009

[W3C11] W3C: *Scalable Vector Graphics (SVG) 1.1, http://www.w3.org/TR/SVG11*. `http://www.w3.org/TR/SVG11/`. Version: 2011, Last checked: 9.3.2012 15:30

[W3C12] W3C: *http://www.w3.org/TR/SVG/coords.html*. `http://www.w3.org/TR/SVG/coords.html`. Version: 2012, Last checked: 2012-03-26 11:52

[Wit07] Wittenburg, André: *Softwarekartographie: Modelle und Methoden zur systematischen Visualisierung von Anwendungslandschaften*, TU München, Diss., 2007

[WN04] Wijk, Jarke J. ; Nuij, Wim A. A.: A Model for Smooth Viewing and Navigation of Large 2D Information Spaces. In: *IEEE Transactions on Visualization and Computer Graphics*, 2004

[Zak09] Zakas, Nicholas C.: *JavaScript for Web Developers*. Wiley Publishing, 2009

[Zak10] Zakas, Nicholas C.: *High Performance JavaScript*. O'Reilley, 2010