Visualization Space Exploration

Theoretical and Practical Viewpoints

Fabian Bolte

Thesis for the degree of Philosophiae Doctor (PhD)
University of Bergen, Norway
2020

UNIVERSITY OF BERGEN

Visualization Space Exploration

Theoretical and Practical Viewpoints

Fabian Bolte

Thesis for the degree of Philosophiae Doctor (PhD)
at the University of Bergen

Date of defense: 20.11.2020

Year: 2020

Title: Visualization Space Exploration

Name: Fabian Bolte

Print: Skipnes Kommunikasjon / University of Bergen

Scientific Environment

The work presented in this thesis was conducted as part of my PhD studies in the Visualization Group at the Department of Informatics, University of Bergen. I have further been enrolled in the ICT Research School at the Department of Informatics, University of Bergen. My research was supported by the MetaVis project (#250133) funded by the Research Council of Norway. Furthermore, parts of the research have been carried out in collaboration with the Indie Lab at the Department of Computer & Information Science & Engineering (CISE), University of Florida.

UNIVERSITY OF BERGEN

ICT Research School In
Information and Communication Technology

The Research Council
of Norway

Acknowledgements

First of all, I want to thank my main supervisor Stefan Bruckner, who provided help and advice even in the most time-critical scenarios. His guidance and mentoring influenced my work more than anything else and I am grateful for his support and belief in me when things did not seem to work out as expected. His skepticism towards common knowledge, his specific knowledge about unknown phenomena, and his regular lame jokes (or in his words: "brilliant and hilarious") probably marked me for life.

I also want to thank my co-supervisor Noeska Smit, who always made time to listen to problems and to provide generally good advice. Friday beers, coffee breaks, and social get-togethers would not have been the same (or even existed) without her.

Special thanks go to Mahsan Nourani and Eric Ragan for providing the expertise to evaluate my visualization method. Mahsan proved to be a constant source of motivation, listened to my complaints, and helped me get through the worst. She is the most supportive and caring person I know and I am most grateful to have her by my side.

I want to thank Helwig Hauser for contributing his wide-ranging knowledge during discussion sessions and for looking displeased when I arrived late for work or meetings. I thank my office mate Yngve Kristiansen for arriving even later to work and meetings, making me feel less bad about it. His coding experience improved my knowledge in web development and code quality immensely. I want to thank Juraj Pálenik for introducing me to the fine art of beer brewing, for taking me sailing, for a countless number of interesting discussions, and for teaching me the "basics" of math. I want to thank Thomas Trautner for his shared interest in beer and his expertise in writing shader code. Without him, I might still be working on that. I thank Sergej Stoppel for the best pizza ever, for organizing countless social events, and for being a role model in coming early to work and handing in papers ahead of time (both of which I successfully managed to avoid). I thank Laura Garrison for her motivation to go out and dance, for bringing a living cuddly toy to work, and for her explanations of weird American habits and language oddities. I want to thank all further and former VisGroup members: Eric Mörth, Charoan Fan, Fourough Gharbalchi, Sherin Sugathan, Veronika Šoltészová, Jan Byška, Katarína Furmanová, Åsmund Birkeland, Julius Parulek, Ivan Kolesar, Erlend Hodneland, Andreas Lind, and Eduard Gröller, for making the time in Bergen one of the best in my life.

Lastly, I want to thank my family for supporting me throughout this time and for believing that I can reach whatever goal I set myself. My grandparents provided all the mental and financial support for my unburdened education. I know they would be proud of me. My family gave me, as my mum — and Goethe — would say, roots for staying grounded and wings to fly and follow my dreams. I could not wish for more.

Abstract

Visualizations are graphical representations of data that have been used in a wide-ranging field of applications to provide a quick overview over data-inherent information. By taking advantage of human perceptual capabilities, visualizations help users understand features and phenomena in data, gather meaningful insights, and drive decision making processes. One of the main motivations in visualization research is to find the best visual representation for a given dataset, user, and task. This challenge is often solved in a subjective manner, where a visualization designer chooses graphical representations, visual channels, and encodings that they believe are best suited for the task at hand. Therefore, the effectiveness and reliability of the result largely varies with the designer's expertise. To make an objectively good design decision, the designer needs to consider all possible visualization methods, or in our words: explore the visualization space. For that purpose, the advantages and disadvantages of individual techniques can be highlighted through comparison methods based on quality metrics, user studies, or theoretical models. Each of these methods can additionally target the visual perception of representations, task-oriented and application-specific measures, structure-oriented matters, or meta-perceptual processes.

In this thesis, we aim to establish a greater understanding of the interconnections between independently studied approaches for visualization evaluation by exploring the visualization space from several different viewpoints. First, we take a theoretical approach to identify and classify previous work on the evaluation of visualization methods. We analyze theoretical models, user studies, and quality metrics, and combine them in a unified structure to distinguish classes of task-oriented, perceptual, meta-perceptual, and structure-oriented measures. We then describe the individual class strengths and shortcomings and propose a direction to combine the separate efforts into a bigger picture to advance the field of visualization research as a whole.

One instance where visualization exploration takes place in practice is during the development of visualization algorithms. By writing code, adding features, and changing parameters, visualization developers expose a large number of representations in visualization space. We developed a system that explicitly displays these individual states to the user and allows for their exploration and comparison. Parameter changes and their effect on all developed visualization states can be inspected to investigate their impact on visual features. The system not only encourages visualization developers to consider multiple representations when creating a visualization, but further allows for comparisons on a more general level. The simultaneous display of source code changes and visual changes in a meta visualization opens up a large branch of possible future research. We made a first step towards a practical development environment that encourages visualization comparison during the development process and reasoning about correlations of source code changes and their impact on the visual result.

In our implementation, we display source code states via node-link diagrams of their abstract syntax trees. Although this representation provides a clear outline of individual hierarchical structures, its juxtaposed nature impairs the comparison of many states. To overcome this issue, we analyzed existing methods for the visualization of dynamic hierarchies and combined the benefits of treemaps and stream-based approaches to display both the individual hierarchies and their evolution over time. We conducted a user study to evaluate the differences in effectiveness on low-level tasks and captured perceptual characteristics in hierarchical visualizations over time. The results suggest that our visualization can be applied as a general-purpose method to replace previous representations for static hierarchies and hierarchical changes over time. All compared visualization types and the effects of mutual parameters can be explored through our open-source implementation.

Finally, we explored aesthetic characteristics of artistic diagrammatic paintings and aimed to apply their visual appeal to storyline visualizations. We developed an interactive application that utilizes techniques for automatic layouting and image processing to create visual results similar to hand-drawn diagrams. Our application can further help artists create an initial layout by interactively adding data to the representation and focus their efforts on artistic aspects that are difficult for machines to imitate.

In the combination of our work, we explore the visualization field from several different viewpoints, move from visualization theory to practice, and show how individual components of visualization comparison can be combined for greater knowledge gain. We hope to encourage visualization researchers to merge their efforts into a larger theory and understanding of how visualizations work and to create objectively effective visualization solutions.

List of Papers

This thesis is based on the following papers:

(A) **Fabian Bolte** and Stefan Bruckner. Measures in Visualization Space. In *Foundations of Data Visualization* (2020), Springer. ISBN: 978-3-030-34443-6

(B) **Fabian Bolte** and Stefan Bruckner. Vis-a-Vis: Visual Exploration of Visualization Source Code Evolution. In *IEEE Transactions on Visualization and Computer Graphics* (2020). doi: 10.1109/TVCG.2019.2963651

(C) **Fabian Bolte**, Mahsan Nourani, Eric D. Ragan, and Stefan Bruckner. SplitStreams: A Visual Metaphor for Evolving Hierarchies. In *IEEE Transactions on Visualization and Computer Graphics* (2020). doi: 10.1109/TVCG.2020.2973564

(D) **Fabian Bolte** and Stefan Bruckner. Organic Narrative Charts. In *Eurographics 2020 - Short Papers* (2020). doi: 10.2312/egs.20201026

The following publication is also related to this thesis:

(1) Thomas Trautner, **Fabian Bolte**, Sergej Stoppel, and Stefan Bruckner, Sunspot Plots: Model-based Structure Enhancement for Dense Scatter Plots, *Computer Graphics Forum* (2020). doi: 10.1111/cgf.14001

The manuscripts presented in this thesis were written during the PhD studies of the main author and in collaboration with Stefan Bruckner, the main supervisor of the main author. Stefan Bruckner significantly contributed with advice and guidance to the realization and publication of the scientific work. Paper C was coauthored by Mahsan Nourani and Eric D. Ragan, who contributed their evaluation knowledge by designing a user study and analyzing the results.

Contents

Part I

Overview

«I am ready! I am ready! I am ready!»
Spongebob Squarepants

Chapter 1

Introduction

Data is one of the most sought after resources to analyze, improve, and personalize products, but it is only as valuable as the insights gained from it. Visualization is a tool to discover information in data and to form a mental model of underlying features and phenomena by taking advantage of human perceptual capabilities. In a standard visualization pipeline, data is transformed, mapped to a visual representation, and displayed to the user. The question of which visual mapping to apply for a given task poses one of the main pillars of visualization research. The vast amount of visualization types and layouts, visual channels, and encodings for individual data items describes a large space of possible visualization solutions that needs to be explored to discover the best fitting representation.

In the remainder of this thesis, we use the term **visualization space** to refer to the collection of all possible visual representations. We can imagine existing visualizations to be scattered in this multi-dimensional space and the distance between two specific visualizations to indicate their similarity. In this case, less similar visualizations are farther apart from each other. Additionally, every visualization method has a number of parameters that influence the mapping from data to a visual result. We describe the **parameter space** as a collection of visualizations that result from applying the same visualization method with different parameter settings. We see the parameter space of a visualization method as a subset of the visualization space. We can imagine it as a bubble that contains multiple visualizations which can be morphed into another through the adjustment of parameters. When two visualization methods with certain parameter sets create the same visualization, their parameter spaces overlap.

When a visualization designer is asked to create the best visualization for a given dataset, user, and task, the procedure can be described as an exploration of the visualization space. The designer can consider every possible visualization, compare their individual benefits and drawbacks, and estimate their suitability for the problem at hand. But the visualization space is vast, in that it requires a lot of time to inspect every possible visualization, and it is sparse, meaning that only a number of different visualization types have been realized while the space between distinct visualizations is often unexplored. Most importantly though, it is not clear how the visualization space can be explored and navigated through. For this reason, a designer might only consider a few visualization solutions and, based on their experience, create a new visualization when considered representations are not sufficient for the task at hand.

Considerable research efforts have been made to explore the visualization space from different perspectives and to guide designers through the process of creating a

visualization. **Theoretical approaches** typically attempt to describe the visualization space as a whole by defining models or pipelines that characterize a visualization based on standardized terms, or detail the visualization creation process. Such models can, amongst other things, provide guidelines towards best practices (e.g., "overview first, details on demand" [179]) and rankings of visual encodings. Over time, these guidelines can become principles [48] in a generalized system to, e.g., analyze if a visualization uses the best visual channels for the given data and task, or if data relationships are also related in the visual representation. While such a general analysis allows for the comparison of multiple visualizations, it is difficult to propose a theory that covers all visualization types without major abstraction from their actual application.

Quality metrics, on the other hand, take a rather practical approach at improving a given visual data representation. The idea is that a measurable component of the visualization is analyzed and optimized to improve its visual quality. Typical examples are the minimization of wiggle in streamgraphs [43] and the reduction of edge crossings in graph representations [157]. The former is specific to stream-based visualizations, but the latter can be applied to all visualizations that use edges, such as parallel coordinate plots, storylines, and Sankey diagrams. This shows that quality metrics are mostly specific to a small number of representations based on the individually utilized visual encodings. They provide useful means for the improvement of a given visualization, but are limited in their capability of comparing multiple visualizations with each other.

User studies can help evaluate a single representation, compare multiple visualizations, or assess general visual properties in terms of their effectiveness and efficiency for the user. For example, studies of human visual perception aim to understand how visual information is transformed from the visualization to its representation in the human mind. This approach can help build a theoretical understanding of how visualizations work and provide practical means to optimize visualizations for a better information mapping. So far, we have only understood low-level perceptual properties of the human visual system, which allows us, for example, to choose a good color scheme for a visualization and to steer the viewer's focus. To understand higher-level perception and cognition, such as sense-making, more research will be required.

Even with human perception fully understood, there are other cognitive processes in the human mind that affect our experience when viewing visualizations. We refer to such characteristics as **meta-perceptual properties**, which are hard to measure and often neglected in common visualization design. Such features include how aesthetically pleasing a representation is, how well it can be memorized, or how profitable it might be. By analyzing such high-level properties, we can integrate knowledge from other fields, such as art and economy, and broaden the field of visualization research beyond the general goal of generating insights.

In this thesis, we aim to assess the visualization space from a variety of different viewpoints. First, we provide a theoretical analysis of measures for the assessment of visualization techniques. We then reflect on our findings and present a novel system for the exploration of the visualization space for visualization developers. Our approach includes innovative ideas for the comparison of visualization techniques and their visual results across parameter spaces. Based on our system's requirement to display changes in hierarchical data structures, we develop a static visualization method for dynamic hierarchies that combines the advantages of treemaps and stream-based approaches. To assess the effectiveness of our visualization methods, we incorporate both qualitative

and quantitative user evaluations and thereby contribute to the understanding of human visual perception. Finally, we assess how the visual appeal of artistic paintings can be transferred to stream-based visualizations to improve their aesthetics and possibly their memorability. Our approach benefits both fields by transferring knowledge from arts to visualization and by developing a visualization tool for artists.

1.1 Problem Statement

The visualization space describes all possible visual representations that designers can explore to choose an optimal visualization for a given task, user, and dataset. Due to the sheer amount of possible representations, an exhaustive exploration of this space is not practically feasible and needs to be limited and steered to at least find a well fitting representation for the data.

Theoretical assessments of visualizations provide models and guidelines for the creation of effective visualizations. They often stem from empirical knowledge and long-term experience in the field, and aim to abstract from specific approaches towards general statements and best practices. In order to strengthen the field of visualization research, continuous efforts need to be made to integrate new insights, abstract gathered knowledge, and merge theories into a general understanding of the research field.

While guidance towards good visualization practices is often of theoretical nature or integrated in recommendation systems for novice users, the field lacks support systems for experts who create visualizations from source code. In particular, the graphical result of visualization algorithms is often neglected in common software visualizations and analysis tools. The visualization community would greatly benefit from an expert tool that integrates means for the comparison of multiple visualization algorithms, supports a large number of different visualization types, and allows for the inspection of parameter settings during the development process.

Several approaches already enable the exploration of the parameter space for a single visualization with varying datasets. But during the development process, an expert might consider multiple visualization techniques or visual features that expose different parameters or react differently to similar parameter changes. So far, little effort has been made to explore the common parameter space between multiple visualization algorithms. For example, changing the same parameter in two different algorithms might result in different magnitudes of impact on the visual result. Similarly, some parameters might in general have a larger impact on the visual result than others. The analysis of how a single parameter acts in different visualization algorithms and how its change influences the respective outcome can yield further insight in the importance of this parameter and towards the comparison of diverse visualization approaches.

While efficiency and insight gain are often the main considerations when selecting a visualization method, other properties can be considered to steer the exploration of the visualization space. Aesthetics, for example, are known to influence the user's interest in a visualization [80] and their ability to memorize it [33]. A visualization designer might therefore prefer an aesthetically pleasing representation over an efficient one for users in the education domain. Since art continuously manages to fascinate viewers and to create aesthetically pleasing visuals, we could benefit from an integration of common art practices to create more engaging and memorable visualizations.

1.2 Scope and Contributions

The research presented in this thesis explores the visualization space from multiple perspectives. We take both theoretical and practical approaches into account, contribute to the comparability of scientific visualizations, evaluate task-specific metrics, and consider aesthetics as a meta-perceptual characteristic for information visualization. The contributions can be summarized as follows:

1. **Theory**. We analyze previous work on visualization space exploration and distinguish classes of task-oriented, perceptual, meta-perceptual, and structure-oriented measures. We describe their individual strengths and shortcomings and propose a direction to combine the separate efforts into a bigger picture to advance the field of visualization research as a whole.

2. **Visual comparison**. We develop one of the first applications to explore the evolution of visualization algorithms during their development. We then compare visualizations of dynamic hierarchies and allow for their exploration through the provision of an open-source library. In this process, we utilize visual comparison, visual parameter space exploration, task-specific measures, and user evaluations for the assessment of specific visual properties.

3. **Timelines**. We evaluate and develop visualizations for time-dependent data including provenance in visualization source code development, evolution of hierarchical data structures, and storyline visualizations. Our applications target a variety of user groups including visualization experts, students, and novices, as well as expert users from other fields (e.g., art).

4. **Parameters**. We present an initial attempt to ease parameter space exploration across visualization techniques by analyzing the impact of parameter changes on visual results of varying algorithms. We discover a parameter for the interpolation of two visualization techniques for the representation of dynamic tree structures and analyze the effects of selected parameters on stream-based visualizations.

5. **Aesthetics**. We present an approach to integrate aesthetics into storyline visualizations by analyzing visual elements in existing artistic paintings and incorporating similar stylistic patterns into the visualization algorithm.

1.3 Thesis Structure

This thesis consists of two parts: I) an overview of the carried out research and II) a presentation of individual publications. The format of the published work was adjusted to fit the thesis layout and the bibliographies of individual papers were merged into a unified bibliography. Chapter 1 of the overview poses an introduction to the topic, states the general problems to solve, and provides a scope of the achieved contributions. We discuss related work on visualization exploration in Chapter 2, including practical approaches, systematizations, interpolations, and parameter explorations. Based on the related work, Chapter 3 outlines the contributions of this thesis. We present examples

and demonstrations of individual achievements in Chapter 4 and conclude the thesis in Chapter 5 by providing a direction for future work.

The second part of this thesis includes four publications that describe the individual contributions in detail. Paper A expands on the first contribution. Papers B and C provide details for contribution 2. Contributions 3 and 4 are shared by papers B, C, and D. Paper D further illustrates contribution 5.

1

«Oh yes, the past can hurt. But you can either run from it or learn from it.»
Rafiki

Chapter 2

State of the Art

In this thesis, we present novel methods for the exploration of the visualization space and parameter space, provide a new example for visualization interpolation, and integrate aesthetic considerations from art into visualization design. Our work contributes to the fields of scientific and information visualization.

In this chapter, we discuss the state of the art for visualization space exploration. We begin our literature review with methods that enable the visual exploration of the parameter space for a given visualization. We continue with approaches that extend the exploration to multiple visualizations and allow for their comparison. We specifically examine methods that can recommend existing visualizations to the user or even create new designs through visualization interpolation. Finally, we look into work where arts and aesthetics were considered and incorporated into the visualization design. We provide a more detailed discussion of the relations between previous work and our own contributions in the individual related work sections of our papers in Part II.

2.1 Parameter Space Exploration

Many visualization algorithms contain parameters that considerably influence the visualization result and provide the flexibility to adjust the visualization to the desired task. While simple interactive interfaces let the user adjust parameter values through sliders and buttons, they only display the result for the selected parameter set and require the user to build a mental model of the parameter space. When the number of parameters increases, it can become difficult for users to select a good set of parameters to achieve the desired visual result, especially when the change of one parameter affects the visual impact of other parameters. In these cases, approaches that ease the exploration of complex parameter spaces can support the user in understanding the relations between parameters and selecting a set that results in a visualization suitable for the given task. We specifically focus on methods that utilize result images for the exploration of parameters and their effects on the visual output.

The GRASPARC [36] system displays parameter settings and the visual result in a tree representation to record the history of user interactions and to allow for their exploration. Design Galleries [143] automatically compute the result images for different parameter sets in a given visualization and display them side by side to enable the visual comparison of results. They utilize an image distance function to group and display a broad range of visually differing results and their parameter sets. The user

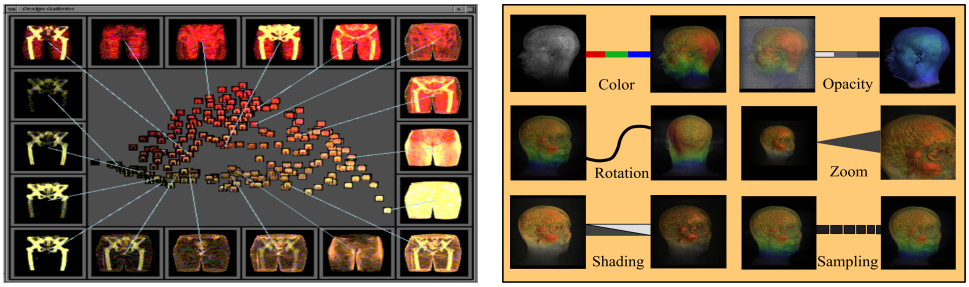


Figure 2.1: Left: Design Galleries [143] display example images for parameter sets that create visually distant results. Right: Image Graphs [138] additionally display the difference in rendering parameters via edges between result images.

is given a simple overview and can choose a parameter set based on the visual quality of the results. Image Graphs [138] go a step further by linking each result image of a parameter set to another result that it can be derived from. They display the parameter change that is required to lead to another result, like a change in color, shading, or camera position, which results in a more detailed picture of the parameter space. Both approaches, shown in Figure 2.1, inspired us to display result images of varying visualization algorithms in a side-by-side overview.

Jankun-Kelly et al. [101] extend 2D spreadsheets of image results into an interactive exploration of the multi-dimensional parameter space. They describe a formal model [102] to capture the process of a visualization exploration session in a graph where each visual result is displayed in relation to the results it derived from. This concept matured in the formalization of the P-Set model [103], which describes parameter space exploration for visualizations as a selection of new parameters based on the result of a previously chosen parameter set. When the user applies a new parameter set to the algorithm, the visualization is transformed. The exploration process can be continued by investigating the visualization and applying further parameter adjustments. The tracking and display of the interaction history and parameter sets can benefit the exploration process by reducing repetitive selections and providing suggestions for where to go next. We apply a similar concept in our own work, where users change source code and the history of changes is displayed for exploration purposes and to reduce repetition.

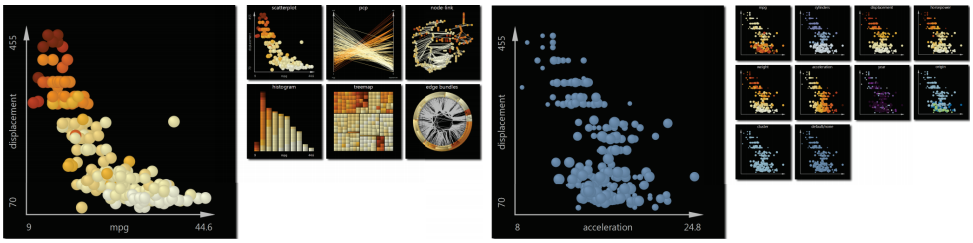


Figure 2.2: Small multiples provide an overview over different visualization types (left) or different parameter sets (right), whereas large singles allow for a detailed comparison of selected approaches. This figure was adapted from *Small Multiples Large Singles* [206].

Tory and Möller [201] utilize a parallel coordinate like view and map result images to their underlying parameter settings for an interactive parameter exploration. In an attempt to structure the navigation process, Small Multiples Large Singles [206] support the user in a continuous exploration of the visualization space. The authors show many parameter options as small multiples and selected states in larger views for a detailed comparison. The user can, in an iterative manner, alternate between large selections and detailed comparisons to find a visual result that best fits their needs. We display an example for this navigation method in Figure 2.2 and utilize the display of small multiples and large singles for visualization exploration in our own application.

Sedlmair et al. [177] describe tasks for parameter space analysis — like optimization, partitioning, filtering, finding outliers, exploring uncertainty, and inspecting parameter sensitivity. They further describe problems and navigation strategies for parameter space exploration and combine their findings in a conceptual framework.

2.2 Visualization Space Exploration

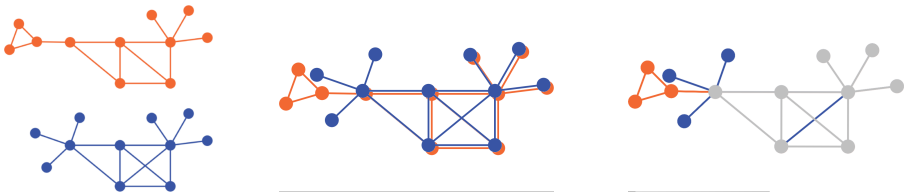
While the exploration of the parameter space is essential to derive a good parameter set for a specific visualization, it does not provide answers for which visualization type to apply to a given dataset. As such, the parameter space for visualization algorithms can be seen as a subspace of the visualization space, which in turn includes all possible visual data representations. In this section, we discuss approaches that enable the comparison of visualization results, the navigation through the visualization space, recommendations for visualization techniques, and interpolations to combine advantages of multiple representations.

Comparative Visualization

Comparative visualizations enable the comparison of multiple data sets through their visual representations [152]. They help users in finding differences in 3D Meshes [173], diffusion tensor fields [231], genome sequencing data [18], and many others. Woodring and Shen [227] display computed differences in volume data and Lampe et al. [125] suggest to reform volumes along a curve for easier comparison. The high importance of the comparison task led researchers to survey, analyze, and discuss comparison strategies for individual application domains, as seen by Verma and Pang [210] for flow visualization and Graham and Kennedy [76] for tree representations.

Given that most visual comparison tasks require users to find differences in result images, Zhou et al. [233] aim to automate this process and evaluate strengths and weaknesses of eleven image quality metrics for their ability to separate similar and different images. Malik et al. [141] take a different approach and present multi-image views to compare multiple datasets in a single image. Similarly, VAICo [172] displays sets of images in a single image and integrates interaction to reveal their individual differences.

Gleicher et al. [73] analyze existing work on information visualization comparison and demonstrate that all examined approaches can be summarized in a taxonomy of juxtaposition, superposition, and explicit encoding. In other words, the authors show that, comparative visualizations display objects next to each other, on top of each other, or display computed object differences. We show an example of each category in Figure 2.3. Some approaches further combine these methods, most commonly by showing



(a) Juxtaposition (b) Superposition (offset) (c) Explicit Encoding

Figure 2.3: Visual designs for comparative visualization (adapted from Gleicher et al. [73]). (a) Juxtaposition places objects side by side, whereas (b) superposition places objects on top of each other. Offset and blending can address overplotting issues. If differences can be computed, (c) explicit encodings can display the differences directly. For example, we display common properties in grey and differences in the color of the object they stem from.

multiple computed differences in a side-by-side view. Javed and Elmquist [105] further suggest overloading, nesting, and integration as additional design principles, which provide a more detailed partitioning of juxtaposition and superposition designs. Similarly, Chen et al. [50] establish the term *visual multiplexing* to refer to visually overlaid information and provide a framework for superposition designs. While none of these authors argue for one design being better than another, Ondov and colleagues [104, 151] evaluate basic visualization designs with common comparison tasks. In an attempt to automate the design of comparative visualizations for spatial data ensembles, Kolesar et al. [119] compute an optimal abstraction of spatial information while preserving user-defined characteristics. This way, more ensemble members can be displayed and compared to each other, without losing essential information.

Finally, Gleicher [72] discusses visual comparison in terms of the elements that are being compared, what challenges occur, which strategies help address these challenges, and which designs are considered for effective comparison. As main challenges, he names the number, size, and complexity of compared objects, as well as the size and complexity of their relationships. To address these scalability issues, common strategies limit the number of examined items, let the user experience them in sequence, or display an abstraction of objects. These challenges often lead to tradeoffs in the application design, so that either the number of addressed challenges or the usability of the application is reduced.

While we apply many of the discussed comparison concepts, the goal of our work is not the comparison of datasets, but rather the comparison of visualization techniques.

Exploration of Visualization Pipelines

The visualization process is often described as a transformation from data space into visualization space into image space [77]. Several visualization frameworks, like VTK [174], ParaView [17], and MeVisLab [118], allow for the construction of a visualization pipeline, which consists of several connected algorithmic modules that realize the implementation of the scientific visualization process. Such systems allow for rapid prototyping of visualization algorithms and provide insight into the created pipeline. The comparison of visualization pipelines can be utilized to compare visualization techniques and to explore their parameter settings. For example, Chi et al. [51] use visualization pipelines to implement information visualization spreadsheets that

display multiple visualization techniques at the same time. They then apply varying parameter sets and display the resulting visual outcomes. By displaying results for varying parameter sets and visualization techniques at the same time, this approach allows for the simultaneous exploration of the visualization space and parameter space.

Provenance is a term widely used by the visualization community (and beyond) to describe the history of data transformations, visualization states, user interactions, analytical findings, and reasoning [159]. Giving users access to such histories enables them to inspect previous results, undo mistakes, and explore multiple paths to solve the problem at hand. Furthermore, provenance can serve as a tool to increase reproducibility of results, communicate cognitive processes in a collaborative environment, and teach visualization creation to students. VisTrails [26] displays the history of visual results and user interactions in the creation of a VTK pipeline. The user can inspect previous states and compare them to the current pipeline. The results of multiple pipelines can further be shown next to each other to create large visualization ensembles and spreadsheets (see Figure 2.4). By inspecting the visual results and pipelines of multiple visualization techniques, users can compare the methods and discover their individual differences.

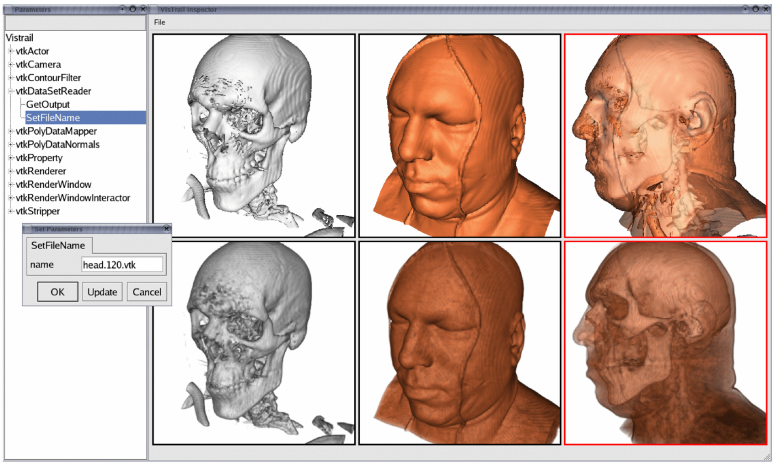


Figure 2.4: VisTrails [26] allows for the creation of spreadsheets to compare results from multiple visualization pipelines (top: isovalues, bottom: volume rendering). Parameters can be changed and synchronized across pipelines for their exploration.

In contrast to VisTrails, which is built on top of VTK, Callahan et al. [44] developed a stand-alone application that receives data through an API, so that it can extend any existing visualization system. They demonstrate how their application can extend ParaView [17] to inspect the history of user interactions. Silva et al. [181] further demonstrate how the exploration and comparison of visualization pipelines can be utilized to teach university students how visualizations are created, which common mistakes happen during their creation, and how varying techniques differ in detail.

Our work focusses on the rendering process of the visualization pipeline and, specifically, at the development of rendering algorithms. We aim to integrate the provenance of user interactions and visualization states into the development process and display the concurrent evolution of source code and visual result. This history can then be used

to explore the exposed visualization space and to teach visualization algorithm development to university students.

Visualization Interpolation and Recommender Systems

Instead of users interactively exploring the visualization space, automatic systems can analyze existing visualization techniques to present methods that fit the requirements or to suggest novel designs. A strong theoretical understanding of different visualization types, their shortcomings and strengths, as well as their perceptual properties, provides the basis for automatic recommendation of good visualizations. Such tools can in particular benefit novice users to select a visualization that fits their data and task at hand. While many recommendation systems suggest well defined presets of visualization designs, others are capable of combining diverse representations. We discuss different approaches of visualization recommendation systems, include combinations of designs, and specifically highlight methods that interpolate between multiple visualizations.

In one of the first automatic approaches for visualization design, Mackinlay [139] utilizes rankings of visual embeddings from perceptual studies to automatically choose a visualization with preferable perceptual properties. SAGE [163] presents previously created visualizations that match the user's intent, allows for the assembly of visualizations, and for their customization. Similarly, IMPROVISE* [234] holds a database of existing visual designs and recommends sketches that are most similar to a user request. Furthermore, it allows for visual refinements via sketch synthesis based on user feedback. Focusing on the presentation of small multiple views, Show Me [140] integrates an automatic selection of visual marks with a selection of visualization techniques ranked by their fit to the data. Gotz and Wen [74] present an alternative approach where they analyze user behavior and detect patterns in the interaction with the visualization to recommend a visualization that better facilitates the task. Other approaches integrate knowledge ontologies for the semantic web (SemViz [71], Viso [214]), statistical data properties and user votes (VizDeck [110]), and deep neural networks [62, 93] into the recommendation process. Finally, CompassQL [225] is a query language which aims to generalize the inquiry of visualization recommendations and demonstrates how it can be integrated with existing recommender systems. This general-purpose approach yields the potential to query the visualization space and fosters its exploration.

Based on visualization pipelines, Scheidegger et al. [171] demonstrate how an automatic comparison can not only query for varying visualization algorithms, but also create new visualizations by adjusting pipelines based on computed analogies. As shown in Figure 2.5, features can be extracted from one visualization technique and added to another pipeline. VisMashup [169] particularly targets novice users and provides simple interaction widgets for the creation of custom visualizations. VisComplete [120] suggests partial completions of visualization pipelines based on a collection of existing pipelines. This approach helps both novice users by suggesting full pipelines and experts by suggesting minor improvements. Based on our application for visualization algorithm development, we aim to integrate similar functionalities for visualization recommendation and automatic source code completion in the future.

Liu et al. [134] present an approach to interpolate results of volume rendering techniques by finding features in data and merging the transfer functions that created individual results. They provide a simple interface that enables novice users to create a transfer function by combining the results of provided example images. A similar ap-

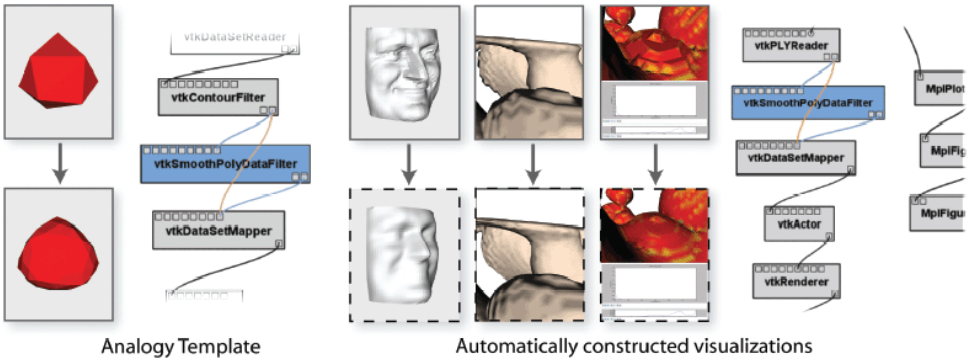


Figure 2.5: Existing visualization pipelines (left) can be analyzed to extract features and automatically apply them to a new pipeline (right) [171]. This way, implemented features can be reused and integrated into recommender systems.

proach is presented by Wu et al. [228], where volume renderings from different transfer functions are aligned on a palette similar to a color wheel. Claessen and Van Wijk [52] show how visualizations for multivariate data, like scatter plots and parallel coordinate plots, can be unified in a description of multiple linked axes. Their flexible approach further allows for the creation and exploration of new visualization techniques based on the same original concept.

Multiple visualization techniques can be combined through embedding, like treemaps embedded in bar charts [94] or data tables [196], and adjacency matrices embedded in node-link graph representations [88]. Specifically the latter could implement a parameter to linearly interpolate between an adjacency matrix and a node-link graph to define the percentage of nodes that are represented by each individual approach. Schulz and Hadlak [176] formalize this idea and describe the visualization space as an interpolation of visualization presets. They demonstrate how the blending between different visualization techniques through numerical parameter configurations can yield new interpolated visualization designs (see Figure 2.6). We extend this approach by presenting an interpolation of time-dependent visualization presets. In our work, we focus on representations for dynamic hierarchies and define a parameter for the interpolation of one-dimensional treemaps and nested streamgraphs.

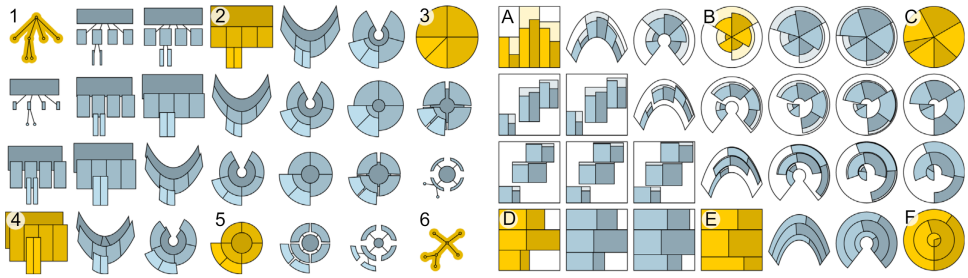


Figure 2.6: Yellow charts mark predefined visualization designs, whereas grey charts are interpolated between presets through parameter adjustments [176].

2.3 Arts and Aesthetics in Visualization

While the typical goal of visualization designers is to create representations that are effective and efficient in conveying the underlying data features, some designs turn out more visually appealing than others. It is therefore to no surprise that researchers became interested in studying the aesthetic qualities of visualizations. For example, Tractinsky et al. [202] found a strong correlation between aesthetics and the usability of a visualization. Harrison et al. [80] found that users judged the aesthetic appeal of a visualization on a mere 500ms exposure mostly by its colorfulness and its complexity. These findings varied based on age and gender, but generally suggest more colorful and less complex representations, which can in turn improve their memorability [33].

Several research efforts were made to apply artistic styles to 3D graphic renderings. Winkenbach and Salesin [222] demonstrate a rendering technique that looks similar to pen and ink illustrations. Kyprianidis et al. [124] summarize many of the approaches in non-photorealistic rendering and create a taxonomy of techniques for the transfer of artistic styles to images and videos. Later approaches are mostly example-based techniques or built on top of convolutional neural networks. StylIt [67] uses hand-drawn spheres as examples to integrate illumination effects and to improve the fidelity of stylized images. Some of the examples and result images can be seen in Figure 2.7. Frigo et al. [68] inspect the differences between local and global effects of color and texture to preserve image structure during style transfer. Such example-based style transfers are one application of patch-based image synthesis techniques, surveyed by Barnes and Zhang [23]. Convolutional neural networks were for example used by Gatys et al. [69] to transfer styles from artworks to photographs, as well as by Selim et al. [178] to apply artistic styles from drawn to photographed portraits. In our work, we follow similar goals and aim to apply an artistic style to information visualization.

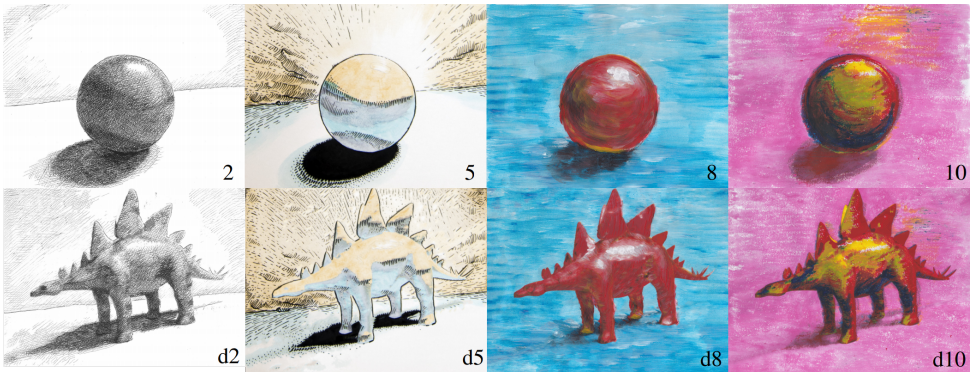


Figure 2.7: Hand-drawn spheres (top) are used as examples to apply similar artistic styles to 3D renderings (bottom). This figure was adapted from Fišer et al. [67].

Lau and Moere [129] proposed a model for aesthetics in information visualization, seeing aesthetics as the degree of artistic influence on the data mapping, rather than as a measure of appeal. In this sense, it should be discussed how art and visualization differ from, or influence each other. Kosara [122] compares information visualization and arts, saying that visual efficiency (like the ability to read data) is not a major concern

in arts, but rather to convey the basic matter. He further alludes to visualization and art lying at opposite ends of a sublimity scale and thereby refers to their contrary goals. Ramirez [160] on the other hand argues that aesthetic information visualization aims to present a subjective impression rather than to convey a message.

Instead of applying artistic approaches to visualization, Informative Art [91] considers the opposite direction where information is integrated into art displays by, for example, changing the color in the image based on the weather or time. Viégas and Wattenberg [213] survey similar artworks which integrate methods from information visualization. They provide a common vocabulary in this field and, for example, define artistic visualization (or visualization art) as a data-based visualization that was created with the intent to make art. This definition also clarifies the distinction between artistic visualization and art, since the latter is typically not based on data. At the same time, they argue that beautiful visualizations cannot be considered art since they lack the artistic intent. This definition draws a clear line between aesthetically pleasing visualizations and visualization art. An extended discussion of literature on aesthetics, art, and visualization is given by Lang [126] and Gough [75].

In an attempt to integrate artistic appeal into visualizations, several approaches employ digital brush strokes to imitate paintings [83, 115, 116, 194]. Samsel et al. [168] extract color palettes from artworks that communicate different emotional moods to create engaging scientific visualizations. Finally, Johnson et al. [109] introduce Artifact-Based Rendering, a technique that starts with physical artistic elements like paintings, sculptures, and photographs, and integrates them with 3D scientific visualization methods. Their approach yields benefits for artists to easier create visualization art, as well as for visualization experts, by introducing new and engaging elements for the visual mapping of data. This method provides an expressive way of synergizing art and visualization and to create aesthetically pleasing results. We demonstrate an example for stroke-based and artifact-based rendering in Figure 2.8. Our own efforts follow this direction by combining arts and information visualization to support artists in the creation of visualization art, and to enhance the aesthetic appeal of storyline visualizations.

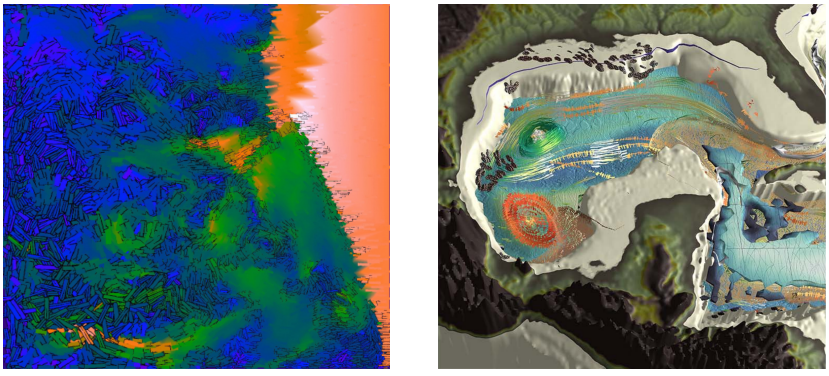


Figure 2.8: Left: Visualization of a supernova data set using brush strokes [194]. Right: Artifact-Based Rendering [109] of the current, flow curvature, temperature, salinity, and nitrate concentrations in the Golf of Mexico.

2

«Answers were always important, but they were seldom easy.»
Patrick Rothfuss

Chapter 3

Contributions

In this thesis we explore the visualization space from several viewpoints, moving from theory to practical approaches and present specific applications of visualization design. After an extensive theoretical assessment of existing approaches for visualization space exploration in Paper A, we present an application for visualization experts and students in Paper B to compare multiple visualization algorithms on a general level. We choose source code as the underlying basis that all visualization algorithms have in common, track the complete algorithm development process, and allow for the exploration of all development states on a visual and source code level. The representation of source code changes over time helps users to compare states and see trends in the algorithm development, but it also poses a major challenge due to the complexity of the acquired information. We therefore examine existing approaches that display the evolution of hierarchical data structures over time, compare and evaluate their distinct strengths and shortcomings, and present a novel visualization method in Paper C that merges two representations to take advantage of their individual benefits. Finally, based on our gathered knowledge in stream-based visualization approaches, we explore the field of meta-perceptual properties by analyzing aesthetics in artistic diagrammatic paintings and by applying a similar visual style to storyline visualizations.

3.1 Theory of Visualization Assessment

One of the primary tasks of a visualization creator is to choose a visualization method that represents the data well and displays it in such a way that a particular user can extract the information required for a specified task. The effectiveness of the chosen visualization often depends on the creator's expertise and can suffer from issues in quality when, e.g., common guidelines for good visualization practices are not followed. It is therefore one of the goals of visualization research to provide the necessary knowledge to distinguish good from bad visualizations and to equip visualization creators with the tools and know-how to build effective visualizations. In the remainder of this section, we discuss common approaches to assess visualization methods and discuss how they can be utilized to compare visualizations.

Different visual encodings, such as color, size, and shape, used in a visualization, steer the user's focus and affect the way comparisons between individual entities are made. Users build a mental model of the underlying data based on the visual input they receive and it is a visualization expert's goal to find a visual mapping that provides

the user with the most accurate description of the data. If we could faithfully simulate human visual perception, we could potentially synthesize visualizations that create desired impressions in a user's mind. Having an at least partial understanding of the perceptual system enables us to choose a visualization that is likely to better represent the data. We summarize these efforts of comparing visual representations on the basis of human visual perception as *measures of perceptual characteristics*. Typical examples for such measures are user studies to understand low-level perceptual properties, mathematical, physiologically-based models of neural behavior, and models of higher-level phenomena like saliency. Cleveland and McGill [53] measure user accuracy in value comparison tasks based on different visual encodings. This study provides visualization experts with the guidance to choose certain visual encodings over others, like length rather than angle, for the task of value comparison. Pineo and Ware [154] utilize a model of retinal and V1 Gabor response to predict the users' perception of speed in a flow visualization and thereby steer an automatic parameter optimization for an improved visual representation. Jänicke and Chen [98] measure the mismatch between features in data and visual features, which allows for a direct comparison of visualizations and the choice of one that best represents the data. Based on these examples, we can see that the results of such perceptual approaches can provide guidelines to best practices, allow for the comparison of different approaches, and steer optimizations of existing visualization algorithms.

Task-oriented quality measures quantify specific visual properties of a representation and, when optimized, mean to improve the quality of the visualization. For example, edge crossings increase the cognitive load on users reading graph representations [156]. Given this knowledge, the visualization can be improved by choosing a graph layout with a minimal number of edge crossings. Similarly, wiggle describes the change in the slope of streamgraph layers and the larger the wiggle, the harder it is to read the heights of individual layers at any point [43]. In both cases, optimizing the defined quality measure not only improves the readability of the visualization, but further improves its aesthetic appeal. Other applications for quality measures are optimizations of orderings in parallel coordinate plots [106], orderings of rows and columns in matrix views [27], and comparisons of treemap layouts for time-varying hierarchical data [211]. While task-oriented quality measures are practically oriented and often lead to automatic approaches to optimize a given visualization, they are mostly specific to a certain visualization type and therefore rarely generalize to a broader visualization comparison.

Some properties of visualizations can be measured for all visualization types and provide a general basis for comparison. Tufte [204] defined a data-to-ink ratio, which measures the percentage of data-representing pixels in an image, and a lie factor, which compares data values to the size of their visual representation. But more general properties are often more theoretically defined so that it can be unclear how to exactly measure them. One example is Mackinlay's expressiveness [139], which describes if a visualization encodes all facts in the data without introducing additional facts. This definition requires an assessment of what "facts" are and a thorough understanding of what can be found in the data. We therefore distinguish these measurements from previously discussed quality metrics and rather consider them as definitions of desirable characteristics of visualizations. Similar efforts to find relationships between data and their visual representation have been made by Demiralp et al. [61], who compare the

symmetry and distance in data and visualization space, and Kindlmann and Scheidegger [114], who investigate relations of manipulations between these spaces. Since all these approaches share a common goal of describing structural characteristics of the visualization process, we summarize such higher-level measures, theoretical models, and pipelines for visualization creation as *structure-oriented measures*. While some research is purely aimed at building a theoretical foundation for visualization [136, 158], it is often possible to draw practical implications and measures from such models. For example, Xu et al. [229] apply information theory to visualization, measuring the amount of information transferred through visual channels, and thereby optimize the placement of streamlines in flow visualizations. Bruckner et al. [38] describe a model to analyze the directness of user interactions, which builds a basis for the comparison of different interaction techniques in a visualization context. Both of these approaches show how theoretical models not only provide principles for good visualization design, but can be employed to create specific measures for visualization comparison.

Finally, *meta-perceptual process measures* describe desired properties of visualizations beyond the common goals of gaining insights and improving user task performance. Examples for such measures are memorability, aesthetics, preattentive processing, engagement, and enjoyment. They are often difficult to quantify, not purely explainable through an understanding of human visual perception, and highly user dependent. For instance, Harrison et al. [80] found that colorful and complex visual representations are perceived as more aesthetically pleasing, but depend on age, gender, and education of the user. Visual embellishments improve a user's ability to recall a visualization [24] and increase enjoyment [166], but also increase the error rate when reading charts [183], which stands in direct contrast to common visualization goals. While these meta-perceptual properties might not be the main focus for most visualization creators, they can find application in more specialized fields like education, where memorability and enjoyment can be of higher importance than the perception of exact values. They therefore play an important role for the general goal of finding a visualization that is best fit for the target user and task at hand.

In conclusion, we summarized and categorized existing approaches for visualization comparison in the context of their measurability. While some of these attempts, like task-related measures, have a higher practical impact at present, others may provide a fuller picture of the visualization space and contribute practical means in the future. While the individual research fields should be followed for short and long term insight gain, we believe that a combination of distinct research efforts can lead to a unified visualization theory and strengthen the field of visualization even further.

3.2 Visualization for Visualization Developers

While many visualization applications are intended for scientists, analysts, and lay users [86], visualization developers themselves are rarely the target user group of a visualization system. In the context of visualization comparison, existing applications are catered to different user groups. For novice users, visualization recommender systems compare multiple predefined visualizations to automatically choose a good visualization for a given dataset. Applications for visualization experts are more specialized, e.g., to explore the parameter space for one specific visualization type or concrete vi-

sualization algorithm. By examining existing methods for visualization space exploration, we have identified a gap in visualization research which we aim to address by developing a practical approach for general visualization comparison targeted at visualization experts. We view source code as the common basis of all visualization algorithms and present an environment for visualization developers to compare different visualization methods during their development on a visual and algorithmic level. We provide tools specifically designed to support the development process of visualization algorithms, the exploration of the visualization space and parameter space, and common comparison tasks that would otherwise be hard to execute.

3

3.2.1 Automatic Compilation and Version Control

We design the interface of our system similar to common integrated development environments (IDEs) by providing a text editor with syntax highlighting and auto completion, and an output window for compilation errors. We utilize a modular window layout where individual views can be added, moved, and deleted based on the user's preferences. The remainder of the available space displays a meta-visualization to provide visual support for visualization exploration tasks during the algorithm development. Figure 3.1 shows the application interface during the development of a flow visualization. We develop the system as a web application with a client-server architecture to make it widely available, to facilitate access to higher processing power via server-side backends, and to provide an easy entry point for students so they do not need to install compilers or libraries.

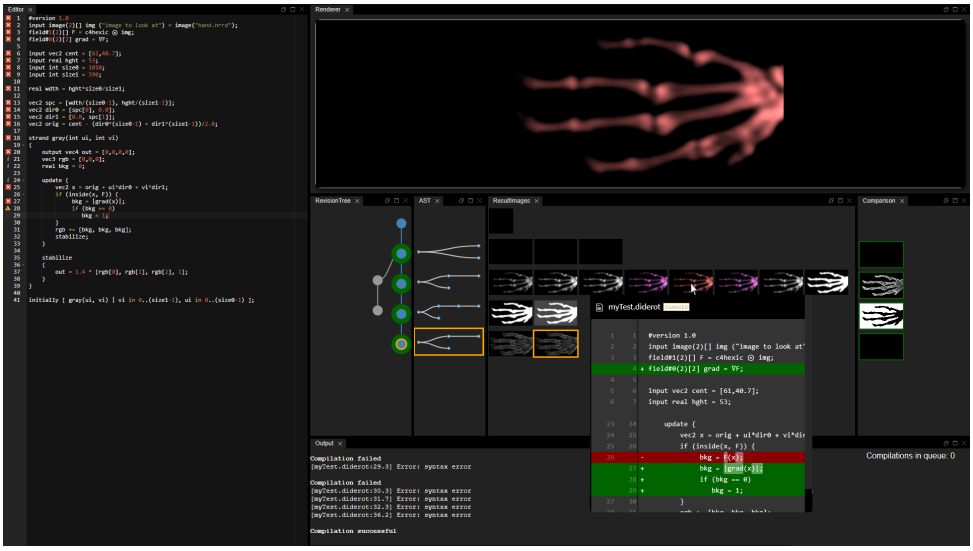


Figure 3.1: The interface layout can be adjusted by scaling and moving the individual modules. It consists of a code editor (left), an interactive view of the visual result (top right), our meta visualization (center right), and a textual output (bottom right).

Compared to other algorithms in computer science, visualization algorithms provide a visual result that, as the easiest form of a sanity check, can be inspected for er-

rors or artifacts. We therefore provide a result window that displays the current visual output of the algorithm and allows for direct interactions with the visualization. We follow previous approaches that utilize on-the-fly live previews, like ShaderToy [13] and Vega-Lite [170], by automatically compiling and updating the result view whenever the user changes the source code. Parameters of the algorithm can be defined as input parameters, which not only enables the computation of a new result image without recompilation of the source code when a parameter is changed, but also allows for the integration of interactions for parameter manipulation. Since the user can define the algorithm and parameters of interest, this setup provides a good basis for future integrations of extensive parameter space explorations. Similar to Design Galleries [143], the system could automatically compute images of varying parameter settings, cluster them, and present possibly interesting sets of parameter values to the user. In our approach, we aim to investigate the unexplored area of fixed parameter effects on varying visualization algorithms to reveal, for example, parameter sensitivities that are highly dependent on algorithmic properties.

When a user changes the source code, it is sent to a server which initiates a compilation process and provides feedback either in the form of a visual result, if it succeeded, or as a descriptive error message. We keep track of all source code changes in a version control system (Git), but only include successfully compiling states as a measure of clutter reduction. In order to enable the exploration of the complete development process, we visualize the Git tree in the form of a node-link diagram and integrate intuitive interactions for users to communicate with the version control system without having comprehensive background knowledge. The individual visual cues of our Git tree visualization are explained in Figure 3.2. We automatically create a repository for the user and commit a new revision whenever the source code compiles. Each revision will be shown to the user as a colored circle that is connected to the circle of the previous revision through a colored line. The visualization follows a top-down timeline, where newer revisions are shown below their predecessors. The circle of the currently viewed revision is marked in orange compared to other revisions that are blue. When the user clicks on any of the previous revisions, the system will execute a Git checkout, load the source code from the selected revision into the text editor, show its visual result in the result view, and mark the now active revision in the Git tree visualization. This way, the user can easily jump back and forth between different visualization states through a single mouse click. When a revision with a successor is selected and the user changes the source code to a new compiling state, the system creates a new Git branch from the current revision and commits the new source code to the newly created branch. This change is represented by a visual branch in the Git tree visualization, showing two circles that are connected to the same predecessor (see Figure 3.2f). We highlight the currently active branch by displaying all its revisions in blue and other nodes in plain grey. To reduce the vertical space of the timeline, consecutive revisions can be grouped into a *merge node*, which is emphasized by a green background.

So far, these features could have been achieved in a similar fashion through the integration of Git and hot-reloading extensions or scripts to common development environments. We relieve the user from setting up such an environment themselves and move computationally expensive procedures to a server so that the application can run on low-level clients. The described architectural functionality builds the basis for the now following features that drive the exploration of the visualization space.

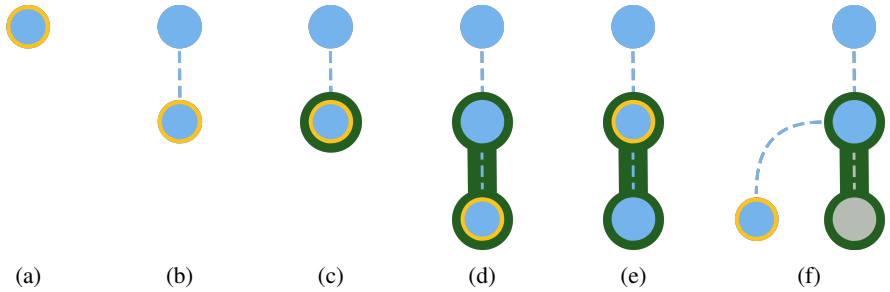


Figure 3.2: Git tree visualization. a) Initially, only one node is visible. b) When a new revision is added, its node is connected to the previous revision's node and it is marked as the current state by an orange outline. c) When a new revision with the same source code structure is added, the nodes are automatically grouped, as indicated by a green background. d) Groups can be expanded to reveal all grouped states. e) When clicking on a previous revision, its source code and result image are loaded and it is marked as the currently inspected state. f) A change of the source code then leads to the creation of a new branch, which is visually indicated by one node having two child nodes.

3.2.2 Visual Exploration of Visualization Algorithms

Given the data for every compiling state of the algorithm development, we can start to integrate functionalities for the support of visualization developers that help them compare different algorithmic approaches. To present differences in visual features and artifacts of the rendering process, we aim to provide the user with a visual comparison between different states of their algorithm. Existing result-driven approaches for parameter space exploration [39, 143] inspired us to show images of results in a juxtapositional view to allow for their visual comparison. We display the result image of every revision of the current branch and horizontally align it with the corresponding node in the Git tree timeline (see Figure 3.3). This thumbnail serves as a visual representation of implemented features in a state and can lead to a faster recognition of a revision of interest than common commit messages can offer. This visualization of visualizations provides a fast overview over all developed states and highlights errors in the algorithm through visually erroneous images. Since this representation of small multiples limits the size of individual images, we display a larger version of each image on mouseover. By alternately hovering over images of interest, the user can identify differences of higher detail.

While this approach showcases major visual differences between states, other features might only lead to subtle changes. In order to highlight even minor visual differences and counteract cases of change blindness, we display the per-pixel image variance between result images (Figure 3.3D). The presented variance image not only reveals areas where the compared images differ, but can also be seen as a kind of task-related quality metric that quantifies the amount of visual change over time. A completely white output shows that all pixels of the compared images differ, whereas a black output represents identical images. Each variance image can be hovered to overlay the currently displayed result image with the visual variance and to identify exact areas of change. We tackle visual comparison on multiple scales by providing a visual

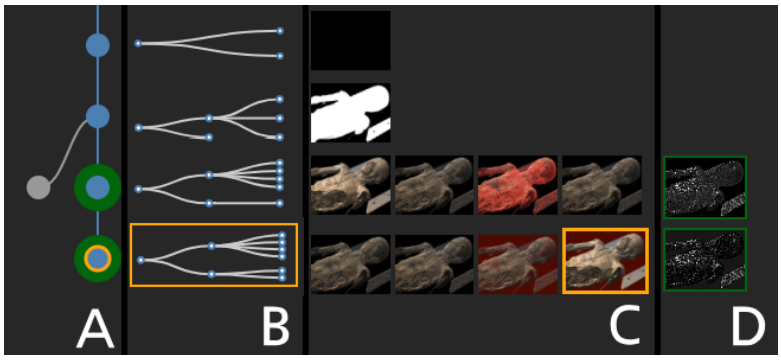


Figure 3.3: Our meta visualization follows a top-down timeline of source code revisions. A) Git tree visualization. B) Representation of the source code structure in each revision. C) Visual result image for each revision. D) Visual difference between grouped results.

overview first and details on demand.

In contrast to approaches of parameter space exploration that show images of visual results for a single visualization with different parameter sets, we not only display the result for the current parameter set, but for all states of the developed visualization algorithm. We take advantage of our client-server architecture to compute the effect of parameter changes not only on the current algorithmic state, but on all revisions that feature the changed parameter. To be precise, we store the executable of every compiled revision, execute every revision of the current branch with the updated input parameter, and update the result images in an asynchronous fashion on the client side. In Figure 3.4, we demonstrate how a change of camera parameters is applied to multiple algorithms to ease their comparison. If a user wanted to achieve the same effect in a common development environment, they would need to change the parameter of interest, store the visual result, find a revision of interest based on the stored commit messages, checkout, compile, and execute the selected revision. Only then could a user compare the result image to the stored one. This process would need to be repeated for every other revision that the user wants to include in the comparison process. Our simplified approach of comparing parameters across visualization algorithms provides a new perspective on parameter space exploration and combines it with approaches for visualization space exploration. It can, for example, reveal parameters that the underlying algorithm is sensitive to and facilitate the identification of parameter values that provide reasonable results independent of individual feature implementations. We further enable the mapping of selected input parameters to interactive interface elements to ease user access to frequently changed parameters. We lay the foundation for future research to apply common knowledge from parameter space exploration to multiple visualization algorithms and explore the effects of parameter changes across the visualization space.

In addition to the evolution of the visual result, we aim to investigate the evolution of the source code and its relation to implemented visualization features. We therefore display a high-level abstraction of the source code that represents the principal code structure for each state (Figure 3.3B). Since the abstract syntax tree of an algorithm is often too large to reasonably display in its entirety, we trim it to a representation of

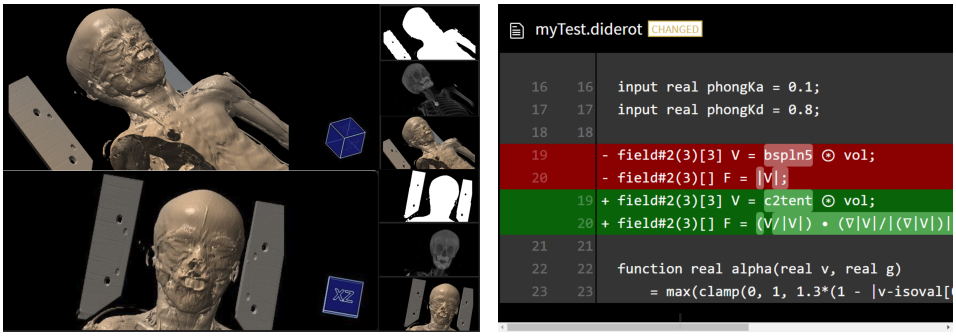


Figure 3.4: Left: An integrated interaction technique manipulates the camera parameters. The new parameter set is applied to all previous revisions to ease the comparison task. Right: When hovering one of the result images, a tooltip reveals the source code differences between the hovered and the current state.

nested block scopes. In C-like languages, a code block is enclosed in curly brackets and each block is either nested inside another block or defined on the document level. We display the static scope tree as a node-link diagram where the root node represents the complete source code, the first level includes one node for each source code file, and each file is a subtree of nested block scopes. The tree displays the number of block scopes, number of files, blocks per file, and tree depth, which can all provide directions to describe the complexity of the algorithm. By comparing trees of different states, we can perceive when files, functions, loops, and conditional statements are added, removed, or moved during the development. In case the user is interested in a more detailed breakdown of code changes, we integrate a tooltip that shows the exact code difference between two states when hovering over a result image of interest (see Figure 3.4). Similar to the representation of visual changes, we address multiple scales by visualizing source code changes in an abstract overview and in low-level detail.

The combination of source code changes and representation of result images can provide insight in the relation between algorithmic modifications and visual impacts. It can, for example, highlight when many source code changes did not lead to significant changes in the visual output, or when very few changes majorly impacted the result. As a future direction, we may integrate quality metrics to measure the visual impact of every single line of code and visualize it to identify lines that are most sensitive to adjustments. We believe that the mapping between source code and its visual result has great potential to showcase the correlation between parts of the algorithm and the visual features they are responsible for. If this correspondence can be sufficiently encapsulated, we might provide tools for visualization developers to define features of interest and provide recommendations for source code that adds this feature to the existing code base. Until then, our application provides visualization developers with the visual support to identify differences between varying algorithmic approaches on both the visual and the source code level, and to compare implemented features towards their suitability for the intended task.

3.3 Stream-based Visualization and Aesthetics

After our discussion of general visualization comparison in theory and practice, we will now limit our exploration to time-dependent, stream-based visualization techniques. We will specifically investigate two visualizations of time-varying hierarchies and define a parameter for their interpolation to benefit from their individual advantages. We then explore the field of meta-perceptual measures by inspecting the aesthetics of stream-based approaches and mimic the artistic appeal of diagrammatic paintings into storyline visualizations.

3.3.1 Visualization Interpolation for Dynamic Hierarchies

While exploring the source code evolution for visualization algorithms, we realized that tree representations can sufficiently visualize each source code state, but come with certain deficiencies for the visualization of code changes over time. In particular, when the trees become larger, the mental matching of nodes in adjacent trees and the identification of individual changes can be challenging tasks. Even if changing nodes would be highlighted in the visualization, the fact that one step in the timeline can include multiple source code changes leads to ambiguities in their interpretation. For example, the user could not distinguish between a code block moving from one position to another, and a block being deleted and another block being added in another location. We therefore investigated methods for the visualization of hierarchical data structures and discovered Chronicler [223] and Nested Tracking Graphs [137] that both utilize streams to display the evolution of each node over time. We generalize these visualizations under the term *nested streamgraph* and display their creation process in Figure 3.5.

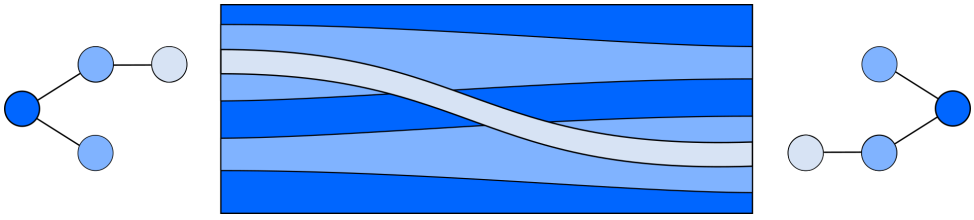


Figure 3.5: Stream-based visualization of changes between two trees from left to right. The nesting of tree nodes is transferred to the stream representation through containment; the dark blue root node of the tree contains the complete hierarchy and the two medium blue nodes are nested inside the root node. The light blue node is nested inside one of the medium blue nodes in one tree, but moves to the other medium blue node in the next hierarchy.

Instead of displaying the hierarchies at individual points in time, nested sreamgraphs visualize the hierarchical changes directly and thereby provide a more intuitive representation that resolves ambiguities in the perception of changes. For example, the nested streamgraph in Figure 3.5 clearly shows that the light blue node is not deleted and added, but moving from one parent to another. However, these visualizations seem to lack the clear outline of hierarchies at individual points in time that juxtaposed tree visualizations excel at. We therefore conducted a user study to test this phenomenon

and demonstrate that users require a significantly larger number of attempts to answer questions regarding the understanding of hierarchies in nested streamgraphs compared to one-dimensional treemaps. We believe that these findings can be explained by the fact that stream-based visualizations only devote one dimension to the containment of nodes, whereas every node in a treemap is completely contained by its parent, which enhances the perception of shapes.

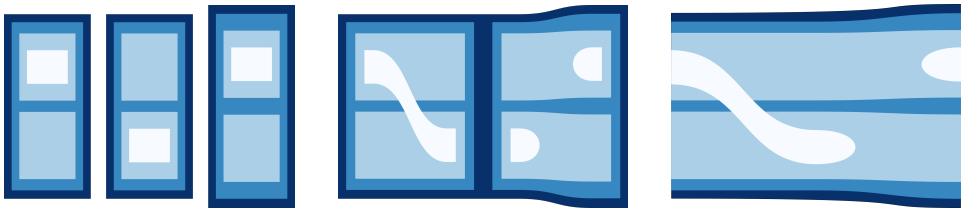


Figure 3.6: Nodes in one-dimensional treemaps (**left**) are fully contained by their parent elements, which results in a clear representation of the hierarchy in all three time points. Nested streamgraphs (**right**) clarify what kind of change happened to individual nodes, but their continuous visualization hinders the perception of hierarchies. Our approach (**center**) combines both approaches by showing changes via streams, but using full node containment for displaying hierarchies.

We designed a new visualization method called SplitStreams that aims to combine the advantages of treemaps and stream-based approaches to facilitate the perception of hierarchies and their changes over time. We take the nested streams as a starting point, split them at every point in time, and, similar to one-dimensional treemaps, utilize the gained space to represent the hierarchy through containment in both dimensions. To demonstrate these differences, Figure 3.6 showcases all mentioned visualization types displaying the same dataset. We define a parameter to control the proportion of space used for hierarchies and for displaying changes, which can be seen as an interpolation between one-dimensional treemaps and nested streamgraphs. We explore further parameters of our visualization that aim to improve the visual quality of all inspected methods through the adjustment of spacings between elements. The results of our user study demonstrate that SplitStreams can be employed as a general method for the visualization of dynamic hierarchical structures, because we were unable to find significant differences between our method and the combined methods in their individual strengths.

3.3.2 Aesthetics in Stream-Based Visualizations

During the investigation of stream-based methods for the visualization of hierarchies over time, we discovered Temporal Treemaps [121] that utilize cushions, similar to cushions for treemaps [209], to emphasize the perception of hierarchies. We applied both cushions and drop shadows to our SplitStream approach and, additionally to possibly improving the hierarchical perception, found them aesthetically appealing. We demonstrate a comparison of SplitStreams with and without shadows in Figure 3.7.

Ward Shelley's [16] diagrammatic paintings are hand-drawn artworks that show the evolution of art-related topics in a stream-based timeline. In addition to using inner

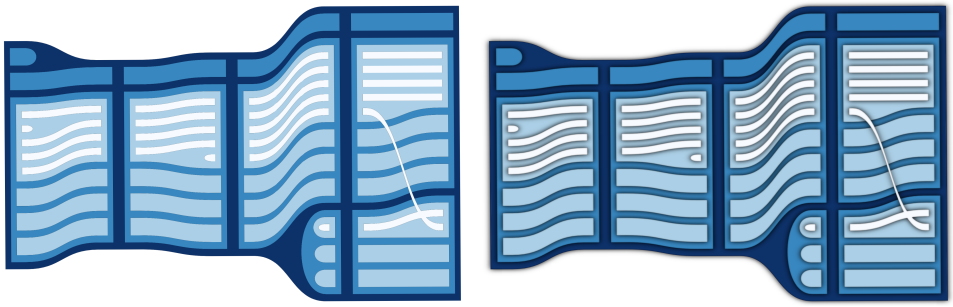


Figure 3.7: SplitStreams with (right) and without (left) applied shadows.

shadows and drop shadows, they possess a remarkable aesthetic style that stems from their unique combination of color and shapes. We aim to learn from the artist by analyzing the aesthetics of his artworks and their individual elements. The goal of our work is two-fold: First, we aim to generate a similar visual appeal as Ward Shelley's hand-drawn paintings in digitally created storyline visualizations. Secondly, by developing an application for the automatic generation of storylines from data, we can provide artists with a tool that simplifies the planning process for their complex arrangements through automatic layout algorithms and integrated interactions for data generation.

Compared to existing layouts for storyline visualizations [135, 150, 192, 193] that aim for a clean presentation, straight streams, and equal spacings, Ward Shelley utilizes rather wavy forms for his streams, while still embracing the space – or absence of space – between individual topics to represent their relation. We encounter a trade-off between a visualization creator's aim to present data in a clear fashion, and an artist's goal to provide an aesthetically pleasing representation. We aim to find a middle ground between both approaches and provide the ability to simulate both behaviors with the aid of a particle-based force layout.

Every stream is represented by multiple connected particles equally spread over the timeline and individual attractive and repelling forces push the stream into its final form. An example for the graph representation and its stream-based result can be seen in Figure 3.8. The computation aims to optimize individual properties of the layout and is highly dependent on the set of parameter values. Such parameters include the power of repelling forces between particles, attractive forces between connected particles, a general gravity keeping the visualization in the center of the selected space, collision forces, and the definition of a cooldown parameter that reduces all forces over time. In Figure 3.9 we demonstrate how a different set of parameters can either result in very straight, or rather wavy streams. The optimization process becomes even more complicated, because the same force type can be defined with different strengths on varying properties. For example, we define a stronger attractive force between nodes of the same stream, compared to connected nodes of different streams, to keep them in a rather straight line. Also, we apply a different force strength on nodes of connected streams than on nodes of labels that are connected to a stream node. This way the distance of labels to their connected streams can be controlled.

The definition of multiple parameters for the optimization process represents a great opportunity for the application of parameter space exploration approaches. In the con-

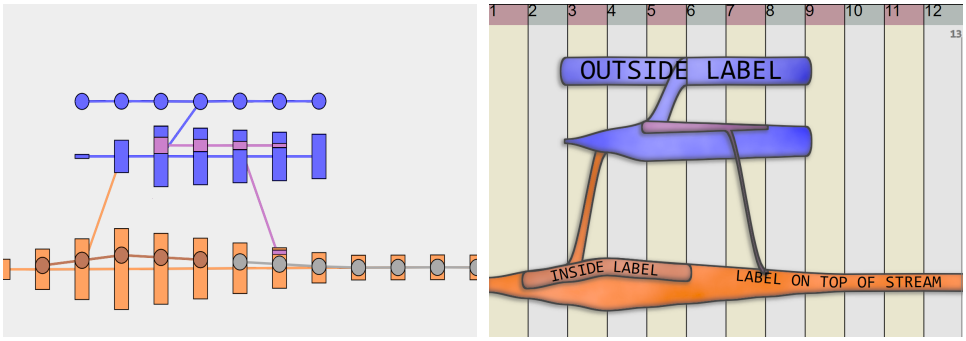


Figure 3.8: We interpret the data as a directed acyclic graph (left), where each stream is represented by connected rectangular graph nodes at every time step. A stream can be nested (pink inside blue), move into another parent (pink moves into orange), merge with another stream (orange and blue), and split into multiple streams (pink). Textual nodes (labels) are represented by a number of circular nodes based on the text length and font size. We differ between labels outside (blue circles), inside (orange circles), and on top of streams (grey circles). The resulting stream representation is shown on the right.

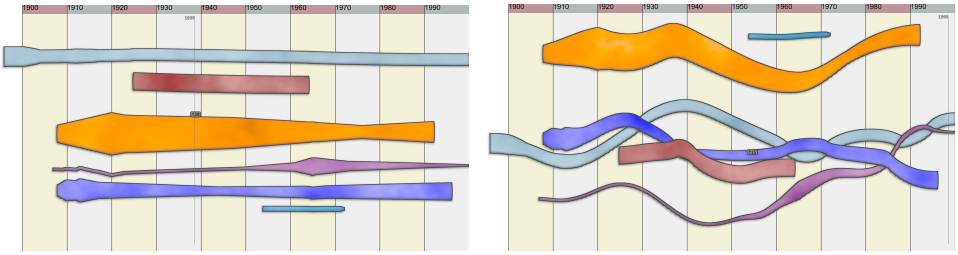


Figure 3.9: The application of different parameter settings allows for vastly varying layouts. In this case, lowering the attractive forces between connected stream nodes leads to a wavier representation.

text of stream-based visualizations, many task-related quality metrics like edge crossings and empty space can be integrated into the process [192]. Given the inclusion of aesthetics as a meta-perceptual measure and the combination with arts as another research field, our Organic Narrative Charts portray an example of an interdisciplinary and wide-ranging application in the field of visualization.

3

«Many people, probably most, must, to find something,
know in advance that it's there.»
Georg Christoph Lichtenberg

3

Chapter 4

Demonstration Cases

With our summary of existing work on measures in visualization space and our application for the development of visualization algorithms, we have contributed to the field of visualization space exploration. Our work on stream-based visualizations gives an example for parameter-based visualization interpolation and demonstrates how aesthetic considerations can be applied to an existing visualization technique. We will now showcase the implemented support features for visualization development, the advantages of our stream-based representation for dynamic hierarchies, and the aesthetic appeal of organic narrative charts.

4.1 Exploration of Visualization Source Code

Given a 3D vector field as input, we aim to visualize the flow patterns in the data and render the output into an image. We decide to implement the visualization as a ray casting algorithm, where a ray is shot through every pixel of the result image into the three-dimensional space to determine its pixel color. The domain-specific programming language Diderot [113] provides simple features that ease the implementation of visualization algorithms. In Figure 4.1 we demonstrate how a ray casting algorithm can be set up in just a few lines of code. While line 1 defines the version of the Diderot language, lines 2 and 3 define input parameters to the algorithm, which in this case set the resolution of the output image. The *initially* keyword on line 19–21 describes the entry point to the algorithm. It calls the *raycast* function for every pixel of the output image by looping through its width and height. The *strand* keyword defines a self-contained function, which means that it is independent of the rest of the algorithm. Such functions can be executed in

```
1  | #version 1.0
2  | input int iresU = 800;
3  | input int iresV = 300;
4  |
5  | strand raycast(int ui, int vi) {
6  |   output vec4 out = [0,0,0,0];
7  |   vec3 rgb = [0,0,0];
8  |
9  |   update {
10 |     rgb = [0,0,0];
11 |     stabilize;
12 |   }
13 |
14 |   stabilize {
15 |     out = [rgb[0],rgb[1],rgb[2],1];
16 |   }
17 | }
18 |
19 | initially [ raycast(ui, vi) |
20 |             vi in 0..iresV-1,
21 |             ui in 0..iresU-1 ];
```

Figure 4.1: Simple ray casting algorithm in the Diderot [113] language.

parallel on multiple threads without the individual executions affecting each other. In our case, the *raycast* function (lines 5–17) takes the x and y position of a pixel as input and returns the pixel color as an RGBA vector in line 15. Diderot repeatedly executes the *update* function (line 9) until the *stabilize* function is called (line 11). For now, our algorithm only produces a black image as shown in Figure 4.2A.

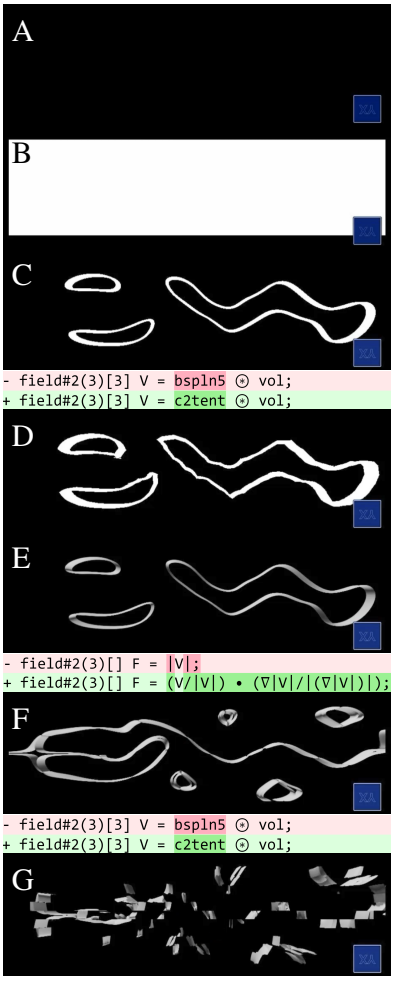


Figure 4.2: Evolution of a flow visualization on a three-dimensional vector field. When source code differences are displayed between two images then they are complete.

We adjust the update function to step through the dataset, which means that every ray is evaluated at every step starting from a near plane until it either encounters a value in the dataset or hits the far plane. In both cases, the stabilize function is called to return a pixel color, which we set to white if a value is encountered or black otherwise. This leads to the result shown in Figure 4.2B, which displays the bounding box of the dataset in 3D space.

Diderot comes with integrated data types that ease certain visualization-specific tasks. As such, our vector field is stored as type *field#2(3)[]*, which means that it is defined in 3-dimensional space and it can be continuously derived twice. While the original input data is discrete, its representation in a continuous field means that we can evaluate the dataset at any given point in space through interpolation. For this, several predefined kernels can be applied to the data to interpolate from a discrete to a continuous vector field. Initially, we select a B-Spline interpolation kernel. For Figure 4.2C we calculate the magnitude of the flow and display it if it is above a set threshold. Figure 4.2D displays the exact same algorithm with trilinear interpolation instead of using B-Splines. When the user hovers over the previous image in our development support system, a unique identifier for both states is sent to the server, where their source code difference is retrieved from the Git repository and sent back to the client. We can see that the only difference between both versions is indeed the interpolation kernel. In addition to the source code, the user can inspect the visual differences and notice that the B-Spline interpolation creates a much smoother representation than trilinear interpolation. By clicking on the B-Spline image, its ID is sent to the server, the revision's source code is retrieved and updated on the client side. This way, the user revoked all source code changes with just a single click and can continue to work on their preferred version of the algorithm.

For Figure 4.2E, we add a light source and perform a gradient-based shading. With

the basic visualization in place, we can display other flow features by, e.g., computing extremum lines instead of the magnitude in the vector field (see Figure 4.2F). While previously changing the reconstruction kernel from B-Splines to trilinear interpolation resulted in a less smooth representation, it now completely breaks the flow visualization (Figure 4.2G). By showing the visual result for every revision, this faulty state automatically stands out from other implementations. Having the source code differences directly available together with their visual impact allows for a structured analysis of the source code and supports the user in narrowing down the problem. Since the shading works as intended and the kernel could previously be changed without an issue, the problem must be due to the new feature computations on top of the interpolated vector field. Basically, the gradient computations (∇) calculate derivatives of the vector field which are not continuous if the field was linearly interpolated. Once again, the user can click on a previously working visualization and continue improving it from there. In this case, the system will automatically create a new branch and thereby remove the faulty result from the currently inspected history. All states can still be retrieved by interacting with the Git representation. Having the history available for the comparison of different states does not only support the visualization expert during the development, but is also valuable to students of visualization courses to investigate common mistakes and build a better understanding of the mathematical and algorithmical basis of visualization techniques.

Evaluation

To evaluate our system, we performed an expert review [200] in two rounds. First, we recruited 4 visualization experts and showed them the development history of the previously discussed flow visualization (Figure 4.2). We provided participants with an introduction to the system's functionalities and asked them to complete several tasks that were designed to explore and use all system features. Participants were asked to follow a think-aloud approach while we recorded their voice and the screen for later analysis. Finally, subjects answered a 28-statement post-study questionnaire on a 5-point Likert scale. Out of the 28 questions, 10 were focused on assessing the overall system usability through the System Usability Scale (SUS [37]), whereas the remaining 18 questions targeted specific support features of our application.

With an SUS score of 78.75, our system was rated "Good" [21] in terms of general usability. Participants praised its superior functionality over existing development environments for creating visualization prototypes, first and foremost the ability to see all visualization results and to investigate source code differences through a simple mouse hover. The two main concerns that were raised during our interviews were whether the system can support other programming languages and its scalability towards larger software projects. Larger projects may contain multiple source files with many more lines of code, which lead to larger static scope trees, larger tooltips of source code differences, and more visual results due to longer development sessions.

In order to address some of these issues, we extended our system to support multiple source code files, C++ and GLSL programming languages, and algorithm caching for faster execution. We showcased these additions in a new example algorithm with about 10 times more lines of code. We recruited 4 new visualization experts and followed the same protocol as for our previous participants by adjusting the tasks towards the new algorithm. The SUS score went up to 88.75, which refers to a rating of "Ex-

cellent" [21]. Working with a familiar programming language, participants considered the tool more beneficial for their own work and asked for the integration of additional features. Answers projected that subjects were confident in the system's ability to handle the development of visualization prototypes, but the development of fully-fledged software solutions would require a larger set of features from common integrated development environments. For a more detailed summary of our evaluation results, please refer to *Paper B*.

4.2 Visualization of Dynamic Hierarchies

While the detailed inspection of source code changes and visual changes can reveal critical lines of code, this task becomes harder as the number of source code changes grows. For this reason, we display a higher level overview of structural changes in the source code for the user to find correlations between visual and structural changes. In our meta visualization, we displayed the code structure as a node-link diagram of nested code blocks. While the hierarchy was clearly visualized for every state, the detection of changes between consecutive revisions was left to the user. Without the necessary visual support, this task can be be ambiguous, as demonstrated in Figure 3.6, and becomes harder as the source code structure increases.

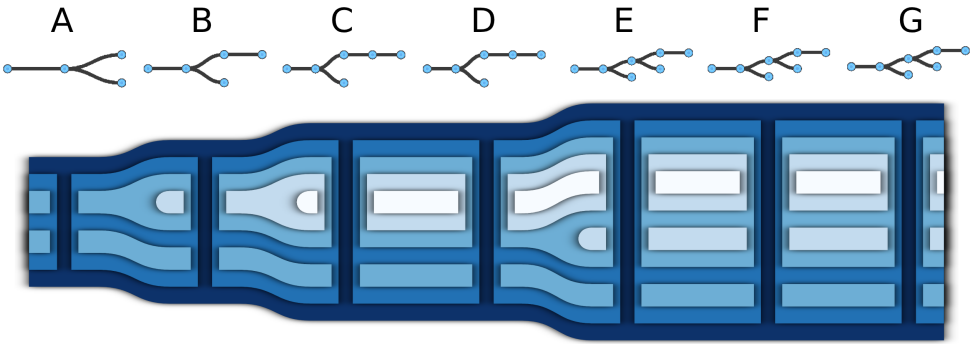


Figure 4.3: Visualization of the source code structures in revisions A-G from the flow algorithm in Figure 4.2. When displayed as node-link diagrams, changes can be hard to detect or even be ambiguous. SplitStreams display changes between consecutive structures directly while showing hierarchies at individual time points in a treemap-like fashion.

We developed SplitStreams as a visualization method that displays the hierarchical changes over time while still allowing for the inspection of hierarchies at individual time points. In Figure 4.3 we compare the node-link diagrams from our meta visualization to our SplitStream technique. The displayed hierarchies represent the evolution of code structures in the algorithm development from Figure 4.2. The stream-based visualization provides a clear overview of the evolution, showing that the code structure monotonically increased and that new code blocks were added in states B, C, and E. We can also perceive that structural changes were only made to the update function from Figure 4.1, while the stabilize function further consisted of one code block only.

While we demonstrated in multiple examples that our method is effective for small datasets, we also want to explore the limits of this visualization technique. For this

reason, we display 34 revisions of the Leaflet [8] repository — a JavaScript library for interactive geographical maps — as a nested streamgraph and as a SplitStream in Figure 4.4. The dataset is borrowed from an evaluation of time-dependent treemaps by Vernier et al. [211]. The nested streamgraph reveals the general evolution of the data, as well as a recurrent pattern of a large number of similarly structured files being added and deleted in the exact same position. The introduction of splits yields a clearer representation of hierarchies, showing that the dark blue color represents the root folder with, at most times, 4 subfolders. It also reveals that the recurring pattern happens on the root level, where multiple files and folders are added to the root directory, before and after the third folder, and deleted in the next revision.

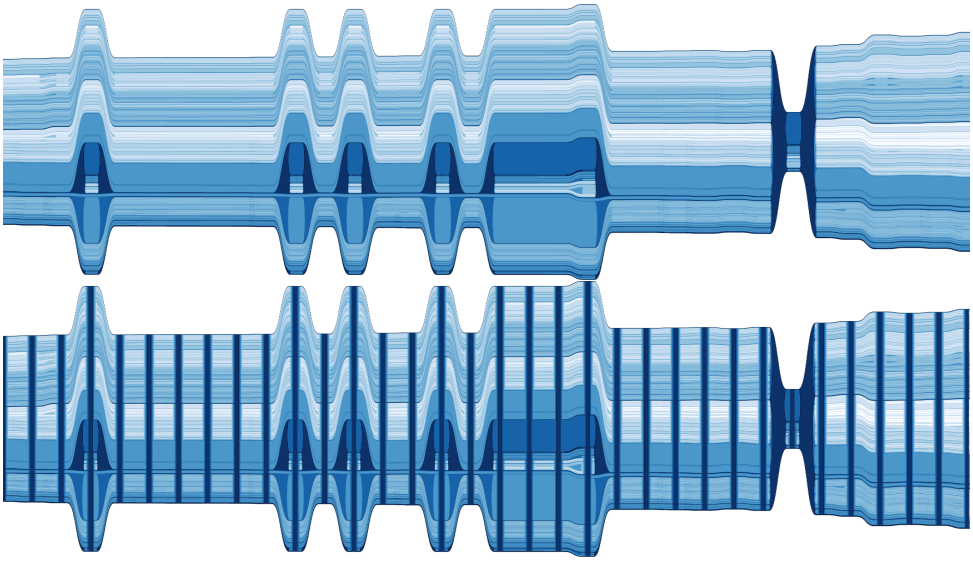


Figure 4.4: Visualization of the Leaflet [8] repository as a nested streamgraph (top) and a SplitStream (bottom).

While this additional information is a useful improvement over nested streamgraphs, we can also identify clear downsides of the SplitStreams approach. Given that streams are split at every point in time, the visualization and perception of flow becomes more disrupted as the number of timepoints increases. It becomes apparent that not all of these splits are required, especially at times where the hierarchy stays relatively constant. Furthermore, although structures become clearer, only a limited number of streams can be displayed at a time. As the number grows, streams are drawn in smaller height and their nested structure is less perceivable. As such, SplitStreams suffer from similar limitations as nested streamgraphs in the number of streams that can be adequately shown.

Although our visualization was inspired by evolutions of source code structures, it is genuinely a representation of dynamic hierarchies. As such, it can visualize evolutions of ancestries, taxonomies, topologies, company staff, file systems, text articles, and population data, just to name a few. In the following, we demonstrate the application of SplitStreams to represent the development of a mouse brain over time.

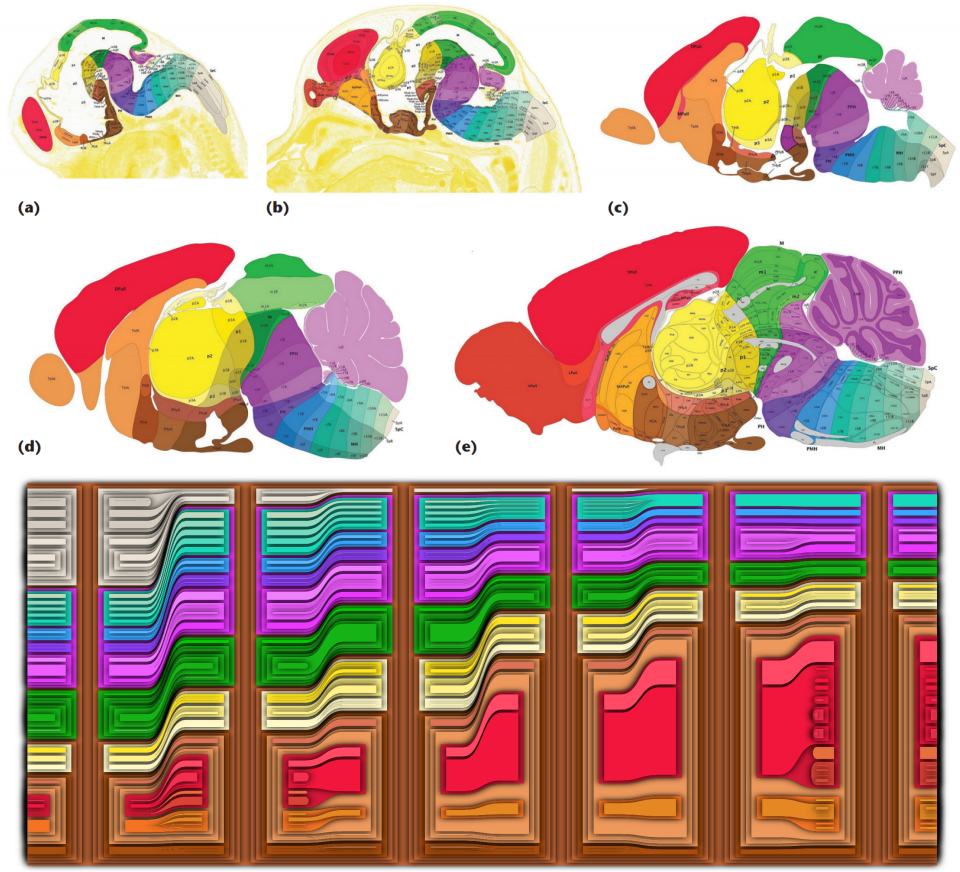


Figure 4.5: Allen Developing Mouse Brain Atlas [131]. In addition to displaying juxtaposed 2D spatial maps of the mouse brain areas (top) [132], we can display their development through a stream-based representation. We can inspect the spatial closeness of individual areas in the maps and gain a better impression of time-dependent growing and shrinking of areas. The introduction of splits in the stream representation allows for the perception of the complete ontology integrated into the visualization. We can see which areas are considered subgroups of other areas due to their functionality (the two displayed representations show slightly different stages of the mouse brain development).

The Allen Developing Mouse Brain Atlas [2, 131] includes over 20000 genes and lists them in an ontology based on their spatial position and function. In Figure 4.5, we display the evolution of brain areas at multiple development stages as 2D spatial maps [132] and as SplitStreams. Since the size of the brain radically increases in the early stages, and to ease comparison, we display area heights as percentages of the complete brain area at each stage. We can see that the spinal cord (light grey) takes up about 25% of the brain in the first stage, but quickly declines. Although we lose some information about area shapes and their spatial closeness, we can investigate the complete mouse brain ontology in our SplitStreams representation. We can for example see that the brain is split into four main parts: the spinal cord (light grey), the hindbrain (pink),

the midbrain (green), and the forebrain (brown). This ontology can be investigated in further detail, showing two subareas for the mid- and forebrain, but four subareas for the hindbrain. Our stream-based representation eases the tracking of dynamic values for individual areas, provides a quick overview for the comparison of multiple stages, and integrates the hierarchical structure for in-depth investigations. Our open-source implementation is available under https://github.com/cadanox/SplitStreams.

Evaluation

We conducted a controlled user study to assess the perceptual differences between one-dimensional treemaps, nested streamgraphs, and SplitStreams. Since our method displays node containment similar to one-dimensional treemaps, we hypothesized that users would understand hierarchies at specific points in time better with SplitStreams than with nested streamgraphs. Similarly, hierarchical changes over time are displayed in the same manner in our method as in nested streamgraphs. That is why we hypothesized users to perform better with SplitStreams than with one-dimensional treemaps in tasks regarding time-related changes.

We recruited 120 participants from Amazon Mechanical Turk who completed 14 basic analysis tasks. We assessed their performance in understanding hierarchical structures at a given point in time, understanding changes in hierarchies, and comparing node value changes. The study followed a between-subject design, where participants were assigned to one of the three different groups based on the tested conditions: 1) SplitStreams, 2) nested streamgraphs, and 3) one-dimensional treemaps. In the tutorial stage, we provided participants with an introduction to their assigned visualization type, 3 example trials, and 3 practice trials with explanations on how the correct answer can be achieved. Then, for each task, we showed an image of a dynamic hierarchy and asked users to provide a numerical answer to a simple question. The hierarchy included no more than 20 nodes and at most 5 timesteps. All the images within the study were displayed by the visualization technique assigned to the user's condition.

Examples of the task-specific questions include determining the number of children of a highlighted node (to assess understanding of hierarchies), the number of parental changes (to assess understanding of hierarchical changes), or the number of nodes declining in value (to assess understanding of value changes over time). Users were allowed to answer multiple times until they answered correctly or hit a 5-minute time limit. In the middle of the assignment, we added a simple task that served as an attention check (only one attempt to answer) based on which 18 participants were excluded from the analysis. We measured users' time and error of the first response, number of attempts, and total task time. For each user and each metric, we calculated the average of all trials within the same task type (hierarchy, change, value).

In our quantitative analysis, we used a Kruskal-Wallis non-parametric test for the main effect and a Wilcoxon post-hoc test for pairwise comparison. Regarding understanding hierarchies at a given point in time, our results show that using the same shape-based node containment as one-dimensional treemaps, users made less error in their first attempt, required less attempts, and performed such tasks faster with SplitStreams than with nested streamgraphs. For understanding hierarchical changes, users of SplitStreams and treemaps made less error in their first attempt and required less attempts than users of nested streamgraphs. Although we expected nested streamgraphs to perform better than treemaps in this task, the result could indicate that a better un-

derstanding of hierarchies also leads to a better understanding of their changes. On the other hand, both stream-based approaches showed users spending less time on the first attempt and on completing the task than users in the treemaps condition. This might indicate that stream-based approaches put less cognitive load on the user than one-dimensional treemaps.

In conclusion, we provide evidence that SplitStreams combine the advantages of one-dimensional treemaps and nested streamgraphs and make a good general-purpose technique for the visualization of dynamic hierarchies. For a more detailed summary of our evaluation results, please refer to *Paper C*.

4.3 Aesthetics in Storyline Visualization

When analyzing Ward Shelley's work, we identified and classified elements of his diagrammatic paintings and analyzed his application of shape, color, and shade. By multiplying stream colors with grey scale fractal noise, we imitate the color irregularities of water color paintings. Additionally, we apply a strong black stroke to stream outlines, as well as inner and outer shadows that are directed to the bottom left. The shape of streams is, as previously discussed, determined by the underlying force layout, whereas individual graph nodes of the streams are, identical to SplitStreams, connected with Bézier curves.

As a preliminary evaluation of the acquired results, we visually compare our automatically created visualization with the original diagrammatic painting. For this purpose, we manually extract the data elements from Ward Shelley's work by utilizing the interaction techniques integrated into our software implementation. By clicking and dragging along the timeline we create a new stream. When hovering over a stream, we can adjust its height at that point in time, whereas intermediate heights are calculated by interpolating between defined height values. By clicking on an existing stream, we can create labels inside, outside, and on top of streams. Finally, by dragging from one stream to another, we can create a link between both streams. While a fully-fledged software suite would provide a good layout out of the box, we need to adjust the provided parameter sliders until the layout fits our needs. We display the original work and our digital recreation of *Downtown Body* in Figure 4.6.

At a first glance, both representations appear rather similar given their choice of color, style of the timeline, overall shape, and complexity. But it also becomes clear that more work is required to create a more similar result in terms of overall clarity of the representation. Although the data is very complex with many links between streams, Ward Shelley's work clearly separates individual streams and shows connection clusters, whereas our force layout tends to create overlaps and more of a "hairball" of indistinguishable streams in dense areas. This part can most likely be solved by a better choice of parameter settings for the graph layout. Another point is the positioning of labels which are at times far from their connected streams and overlap each other. Particularly labels inside and on top of streams tend to be stuck to the top or bottom of the stream instead of filling the available space. Furthermore, some labels turn too sharply when following the streams curvature, making the text hard to read.

Given that most of our concerns regard the force graph layout and parameter settings, we consider this work a success in the application of an appealing visual style to

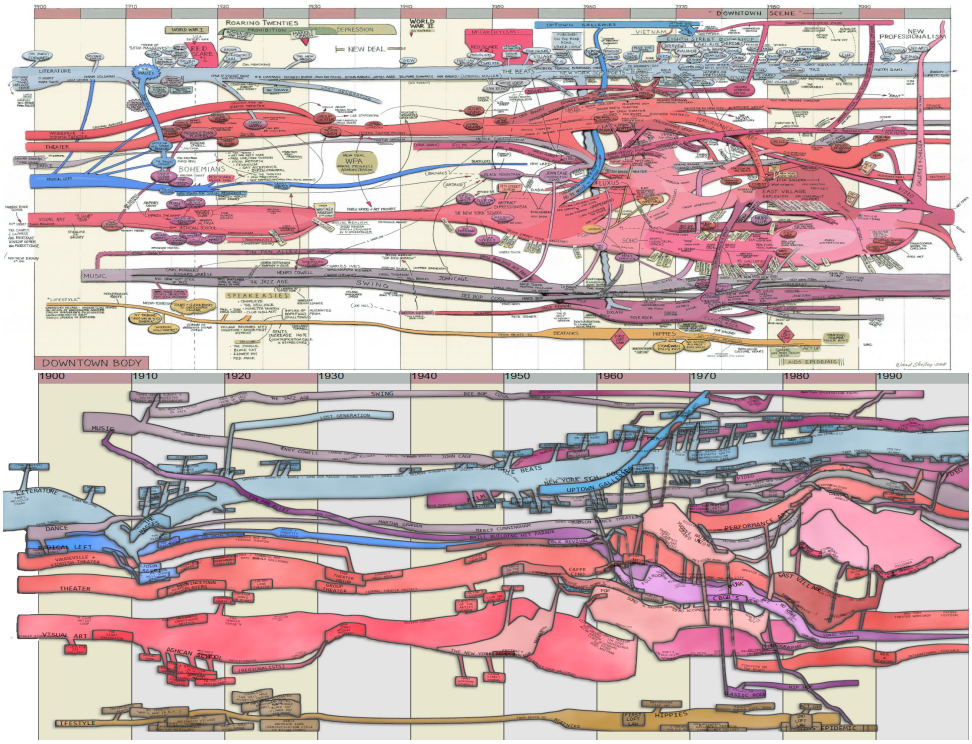


Figure 4.6: Top: Ward Shelley's [16] Downtown Body represents the evolution of art-related topics, movements, major events, contributions, and representatives of individual times. Bottom: Our attempt to digitally recreate the artistic appeal in Shelley's work and to apply it to storyline visualizations in general.

storyline visualizations. We built a foundation to study the effects of aesthetics on user interaction and perception specific to storyline visualizations and possibly inspire other researchers to consider similar aesthetic styles for broader visualization design. Our open-source implementation is available under `https://github.com/cadanox/OrCha`.

4

«Sometimes science is more art than science, Morty.»
Rick

Chapter 5

Conclusion and Future Work

We have contributed to visualization research through a theoretical analysis of measures and approaches for visualization assessment. The lack of practical approaches for visualization space exploration during the development of visualization algorithms led us to build our own development environment. Our tool is one of the first visualization applications that specifically targets visualization experts as intended users. We support them during the development process with simple interaction techniques for the comparison of visual results, source code changes, and parameter settings. Our meta visualization serves as a novel concept to provide practical means for general visualization comparison and to investigate correlations between source code and visual features. The display of a visual history of visualization algorithms follows previous research on provenance visualization and allows for the analysis of expert user behavior and the inspection of the complete algorithm life cycle. Its representation yields further benefits for teaching students common mistakes in visualization development and complex relations between mathematical concepts and visual features.

We developed a new method for the visualization of dynamic hierarchies that combines two existing visualization techniques and thereby presents a time-dependent example for parameter-based visualization interpolation. Our technique improves user perception of hierarchies in a stream-based visualization and contributes to the uncovering of the visualization space by proposing new interpolated visualization designs. Our quantitative user study evaluates low-level perceptual properties in multiple visualization techniques and adds to the existing understanding of perception knowledge in the community. By providing an open-source JavaScript library of our visualization technique, we join the community efforts to improve reproducibility and ease method integration and comparison for other researchers.

The consideration of parameter explorations and quality metrics is visible throughout all our work. Our integrated development environment supports the exploration of parameter changes across visualization algorithms. Assessments of quality can be made by investigating the high-level source code representation and the visualization of variance in result images. SplitStreams define parameters for visualization interpolation, semantic zoom, and general improvement of visual quality. Finally, organic narrative charts aim to mimic diagrammatic paintings through storyline visualization and require the exploration of parameter settings in the force-based layout. Quality metrics for the assessment of white space and edge crossings can steer the automatic layout process and yield results of higher visual appeal. The integration of aesthetics into our visualization is in line with previous research efforts on meta-perceptual

properties. By mimicking artistic appeal and providing an interactive visualization for artists, we bridge the gap between research fields and contribute to both.

In conclusion, we explored the visualization space from theoretical and practical viewpoints and integrated quality metrics, parameter space exploration, perceptual, and meta-perceptual properties into our assessment. We see many possibilities to further improve the comparability of visualizations and to uncover the visualization space. When thinking of our visual support system for visualization development as a basis, additional features could strengthen the understanding of correlations between source code and visual outcome. Individual lines of code could be identified and highlighted to display their impact on the visual result. By analyzing the performance of individual code revisions, performance might be employed as a quality metric to favor one visualization algorithm over the other. When a large number of algorithm histories would be stored and analyzed, algorithms or code snippets that produce the same or similar visual features could be recommended as replacements to currently used code bases. We could build towards a recommender system for visualization experts similar to how recommendations for visualization pipeline constructions were previously proposed. The analysis of these histories could further reveal common patterns in visualization algorithm development and lead to better support tools for experts. By analyzing parameter impacts across visualizations, the algorithms' sensitivity towards parameter changes might lead experts to prefer one technique over the other in particular for interactive systems. By automatically running tests for different parameter values on the server, users would be released from common trial-and-error routines during development and instead be presented with good default values out of the box. Finally, we can imagine further advances in the investigation of parameter-based interpolations for example-based sampling of the visualization space, as well as integrations of aesthetics with common visualization techniques to make them more approachable for public audiences.

«To infinity and beyond!»
Buzz Lightyear

Part II

Scientific Results

Paper B

Vis-a-Vis: Visual Exploration of Visualization Source Code Evolution

Fabian Bolte and Stefan Bruckner

University of Bergen, Norway

Abstract

Developing an algorithm for a visualization prototype often involves the direct comparison of different development stages and design decisions, and even minor modifications may dramatically affect the results. While existing development tools provide visualizations for gaining general insight into performance and structural aspects of the source code, they neglect the central importance of result images unique to graphical algorithms. In this paper, we present a novel approach that enables visualization programmers to simultaneously explore the evolution of their algorithm during the development phase together with its corresponding visual outcomes by providing an automatically updating meta visualization. Our interactive system allows for the direct comparison of all development states on both the visual and the source code level, by providing easy to use navigation and comparison tools. The on-the-fly construction of difference images, source code differences, and a visual representation of the source code structure further enhance the user's insight into the states' interconnected changes over time. Our solution is accessible via a web-based interface that provides GPU-accelerated live execution of C++ and GLSL code, as well as supporting a domain-specific programming language for scientific visualization.

B.1 Introduction

The process of developing a visualization algorithm typically involves a trial-and-error approach consisting of the repetitive task sequence of writing code, compiling the program, and comparing the visual result to previous outputs. This is common practice in research and industry for fixing bugs or developing new features. Existing development tools ease the software development on a general level by providing visual insight

This article was published in IEEE Transactions on Visualization and Computer Graphics (2020).

into the source code structure and the application's performance. Development tools specifically designed for visualization algorithms could further improve the user's experience by providing insight into the visual changes created by the source code [128]. An overview of the visual results created at different points in time can ease the comparison of features in the algorithm. Source code changes and changes in the visual result can be investigated as a unit, instead of being considered individually. When teaching visualization algorithms to students, free exploration of the source code's evolution and its resulting visual outcome can build a deeper understanding of the underlying technical details and problems that occur during the development of such algorithms. Some recent tools [13, 170] provide a live view of the application's result and thereby relieve the user from the common compile-and-run cycle. While several approaches present a visual history of the source code evolution [197, 215, 223], none of them connect the source code to its graphical output. The development of visualization techniques, in particular for prototyping and education purposes, could greatly benefit from a coupling of the source code to its visual result, making it easier to pinpoint when artifacts are introduced or whether a specific method is sensitive to noise in the data.

In this paper, we present a novel approach for visualizing an algorithm's evolution for a general purpose (C++, GLSL) and a domain-specific programming language targeted at scientific visualization algorithms. We provide tools for investigating all revisions at different levels of detail with side-by-side comparisons for visual results, a visual representation of the source code's structure, difference images, and source code differences. We further apply user-defined algorithmic parameters to all states of the evolution to ease the comparison task with respect to parameter changes. We present results in an automatically updating and interactive environment, that enables direct state comparison and navigation throughout the whole development process with instant state switching for free investigation. Our system provides visual support of the development process in an interactive visual analysis tool for visualization researchers and practitioners. The tool can further be utilized in the education domain to teach visualization algorithms and their detailed differences to students, building a fundamental understanding of the correlations between algorithmic and visual changes, and a strong basis for future visualization research.

Our main contributions can be summarized as follows:

- We introduce a novel approach for the concurrent live visualization of the evolution of scientific visualization source code and its visual output.
- We provide automatic revision management with interactive state switching and visual guidance.
- Algorithm parameters can be automatically mapped to user interface elements and their effects can be explored interactively on the entire revision history.
- The system is implemented as a web-based client-server environment enabling the development of GPU-based visualization algorithms on any client.

B.2 Related Work

Our system provides support for the development of scientific visualization prototypes and benefits from on-the-fly compilation, live previews, automatic revision management, and parameter management. While our general approach of combining these features into a single development environment is novel, many visualization techniques exist for individual features.

Terminology
A visualization developer (user) writes *visualization source code* to describe an algorithm. Instead of considering full-fledged visualization solutions, which integrate the whole visualization pipeline (reading and processing data, creating a render window, etc.) we mainly focus on the visual mapping and rendering part. We use the term *state* to refer to a revision of the source code in a version control system. Every state creates a *visual result* (*output*) based on a given parameter set. We create a *meta visualization*, used in the same sense as Bertini et al.[31] – a visualization of visualizations, to showcase the evolution of an algorithm.

Software Visualization
Software visualization has been a prominent topic over many years and many powerful methods have been developed. They visualize software projects at different granularities, such as the source code level, the software structure, or the runtime behavior of the program [20].

SeeSoft [65] represents a file as a column and each line of code (LOC) as a fixed size row. Marcus et al. [142] reduce the required space of this visualization by representing each LOC as a square and appending them to fixed length rows. They further highlight the syntactical structure of the source code by coloring each square depending on its nesting level or keywords of the language. The visualization of source code history is likewise a prominent research field in software visualization [145]. DeVis [232] summarizes the number of LOCs that have been added, deleted, or modified in a pie chart and provides this information over time in a spiral layout. CVSscan [215] highlights structures in source files and aligns the LOCs across time to create a history view. Telea and Auber [197] take this approach further by directly visualizing the evolution via a flow graph. Holten and van Wijk[92] show how this approach can be enhanced by edge bundling. Chronicler [223] builds a node-link diagram from the source code structure and connects correlating nodes in a flow graph to display the evolution of hierarchical structures. The entire structure of a software project can be visualized to gain an overview of all existing classes, their relationships [59] and their evolution [54, 111, 127, 219]. Additionally, the authors of developed classes and their collaborations can be displayed [70, 150]. The runtime behavior and performance of an application can be visualized to detect issues or bottlenecks within the code [95].

This kind of visual analysis can be generally applied to, but is not specifically designed for, source code of visualization algorithms. In our approach, we take advantage of the visual output that is specific to visualization algorithms and provide tools to support the intrinsic needs of visualization developers.

Visualization Pipeline
Existing frameworks like VTK [174], ParaView [17] and MeVisLab [118] enable the construction of and insight into the pipeline of a scientific visualization process. Each pipeline consists of several modules featuring algorithms with their respective inputs and outputs, including operations from data space, through visualization space, into image space. These kinds of frameworks provide visual support for the rapid prototyping and analysis of visualization pipelines. They simplify the task of comparing individual pipelines against each other and finding proper parameter sets to run them with. In contrast, our system provides visual support for the process of prototyping a visualization algorithm, which could then be utilized as a module within one of these frameworks.

Parameter Management
Many systems provide support for finding a suitable parameter set for a given visualization. Design Galleries [143] generate multiple output images from dispersed parameter sets to provide an overview of the parameter space and enable the easy identification of desirable results. Image graphs [138] display the parameter changes between different output images and spreadsheet-like interfaces [101] allow for the investigation of parameter effects and their interplay. Sedlmair et al. [177] evaluate existing work on visual parameter space analysis and divide problems, strategies, and tasks into a conceptual framework.

All these approaches analyze and compare the effect of different parameter sets for a given visualization with the goal of finding one suitable set. In contrast, we utilize one parameter set at a time to analyze its effect on different visualizations in order to inform the user about aspects such as sensitivity to parameter changes during different phases of development.

Visual Provenance
Visualizing the evolution of a workflow is closely related to the notions of provenance [89] and graphical histories [85]. They provide important information for improving the user's understanding, the product's quality, and ensuring reproducibility. Several tools, including Kepler [19], Triana [195], and work by Pimentel et al. [153], focus on the history of user interactions and modifications in data, meta data, information systems, and workflow. Such systems have shown that a version tree and thumbnails can be utilized to improve a user's exploration capabilities [45, 186]. VisTrails [26] is a system specifically designed for visualizing provenance in the visualization pipeline, allowing for the comparison of several visualizations for different data and parameter sets. It therefore addresses many of the issues found in the process of constructing a visualization pipeline for a given task and data set, that we face in the context of developing a visualization algorithm. However, our approach starts at an earlier level, even before the visualization methods are known, and visualizes the provenance of the algorithms themselves by allowing for the investigation of the development process. We thereby provide a support system for visualization developers instead of users of existing techniques.

Live Execution & Notebooks
An increasing number of powerful online tools provide live previews for the output of code in different languages. Notebooks like Jupyter [117] and Observable [11] al-

low for the definition and individual execution of code snippets, interlaced with text to provide explanations of the code. This leads to a tighter coupling of source code and its output for interpreted languages. For compiled languages, on-the-fly previews, as seen in Overleaf [12] for LaTeX, are often only provided for the output of the complete source code. ShaderToy [13] allows for the editing of GLSL code and combines a live view of the result and predefined inputs to create a powerful environment for prototyping shader effects. Vega-Lite [170] is a high-level grammar for interactive information visualization that provides an online editor with instant visual feedback and extensions for visual debugging techniques [90]. Literate visualization [226] builds an evolution of visualizations and explanations for their design choices on top of that. None of these solutions, however, provide means for exploring the evolution of the code and its results over time.

B.3 Overview

Visualization has been proven to be invaluable for many applications in vastly varying fields, by improving the user's insight into complex data [79]. It is therefore remarkable that we, as visualization researchers and practitioners, make limited use of advanced visualization tools in what is often a considerable part of our daily work – the development of visualization algorithms. We want to improve the current situation by providing an interactive tool for visualization developers to support the investigation of their work. Our first approach in this direction is aimed at the prototyping of such algorithms and we see great potential for our application being utilized in the education domain for teaching visualization algorithms to students.

B.3.1 User and Task Requirements

In order to improve the visualization researcher's development process, it is important to first understand their problems and needs. We therefore take a look at a programmer's standard workflow, visualized in Figure B.1. During the development process a user typically (partially) implements a visualization algorithm, compiles the source code, and tests if the developed program generates the expected visual result. If it does, the current state can be manually stored as a backup and extended until it supports all required features. If an intermediate result is incorrect or undesired in any way, the source code must be inspected to locate and fix the issue (debug), recompiled, and tested again. This time-consuming process must be repeated until the result appears free of issues. If the identified issue cannot be located or cannot be fixed, a previously stored state must be restored. Even if the source code is free of issues, the implemented feature might turn out to be unsuitable for the given task and should be replaced by another feature. This again requires the user to continue from a previous state.

Many visualization algorithms define several parameters which significantly influence the visual result [100]. Aspects such as the robustness against small perturbations of these parameters typically need to be continuously verified. While unit tests and other forms of testing are meant to fulfill a similar purpose, they are often only employed once a set of required features is clearly defined and already implemented in

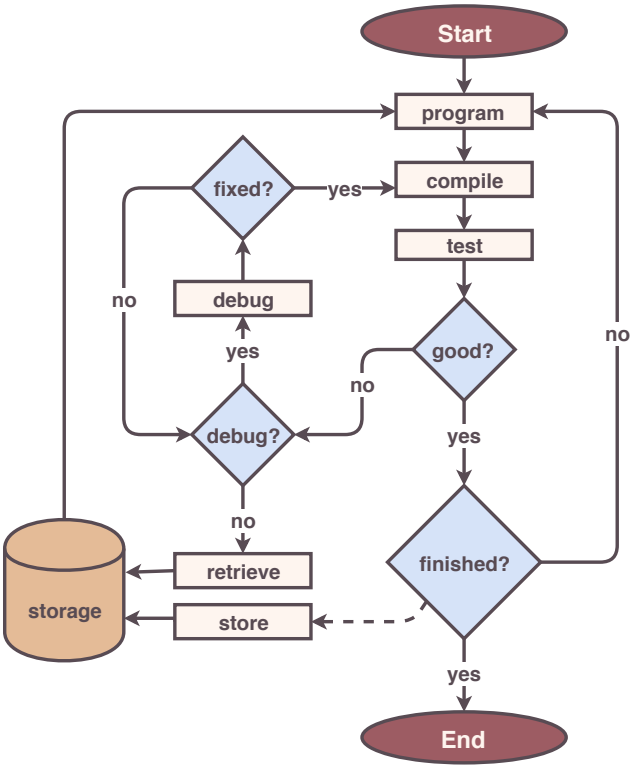


Figure B.1: Development workflow. During the development process, a user programs a visualization algorithm, compiles the source code, and tests the outcome. Bugs are typically fixed in an iterative manner. The whole process is normally performed separately for each feature of the implementation. Functioning features can be manually stored, and restored when the current state is not satisfactory.

B

form of a prototype. In research, in particular, we often lack the detailed specification needed to clearly specify such tests from the very beginning. Based on previous observations [128] and our own analysis of a typical visualization developer's workflow, we can identify a set of tasks that are regularly performed by the user and could benefit from additional support:

T1. Compiling the visualization source code to investigate the visual result regarding functionality
T2. Comparing visual results from different revisions
T3. Understanding the impact of several implemented features on the visual outcome
T4. Locating the parts of source code that are responsible for a visual feature or bug
T5. Switching between source code states
T6. Finding suitable input parameter values

These tasks are partially supported by common programming environments, but could in many cases be performed in a more efficient manner. *Compilation* of the source

code typically needs to be manually initiated, thus breaking the developer's focus on the implementation of the algorithm. Several visual results can only be compared by individually compiling and investigating the corresponding states, which have to be manually stored and restored.

Although the *source code* creates the *visual features* in the algorithm's result, these two aspects are never directly shown in *relation* to each other. The source code that is responsible for a given feature can thus only be located by reading and understanding the source code. This task is only accelerated if the user knows in which order the features were coded. The same issue occurs when trying to find a bug in the source code. By the time the user manually initiates the compilation process, a lot of code might have been written and is suspect to having introduced the problem.

In order to *compare* features and source code, the user must be able to switch between different states. This task is frequently performed to compare functioning to malfunctioning code, but not sufficiently supported by commonly used development tools. As bugs or other issues are often only discovered when inspecting the visual results, current revision management tools, which provide guidance by highlighting source code differences and listing manually-entered commit messages, are a suboptimal solution.

Finally, the analysis of *input parameters* and their effects on different stages of the development process is only poorly supported by current tools and typically requires a cumbersome trial-and-error process.

B.3.2 System Design

Looking at the outlined shortcomings of commonly used systems with respect to the discussed user tasks, we are able to identify several key aspects that would greatly assist visualization developers:

- Visualization for investigating the visual results of several different states at the same time (T2, T3)
- Visualization of the states' source codes in relation to their visual features (T3, T4)
- On-the-spot switching between several states, visually guided by the states' visual results (T5)
- Automatic compilation of source code in the background (T1)
- Automatic and transparent storage of source code states (T2, T5)
- Interactive elements for on-the-fly value definition of input parameters (T6)

The desire to provide an integrated solution that addresses these points lies at the heart of our system's design, which is illustrated in Figure B.2. We aim to assist the programmer in focusing on the implementation of the algorithm's features, understanding and comparing the impact of different choices on the visual result, and allowing for their comparison. We provide further support for the localization of issues and the automation of tedious tasks.

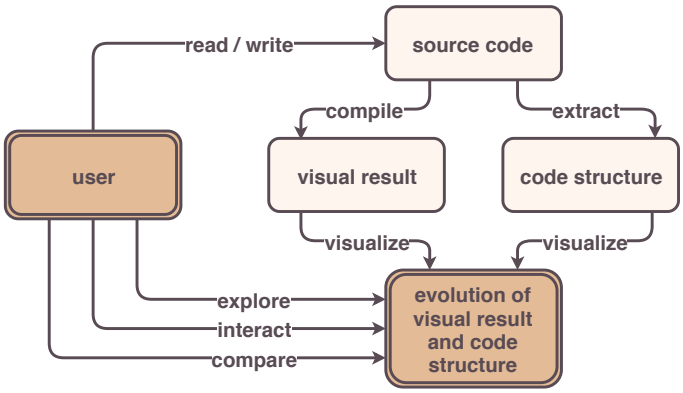


Figure B.2: System overview. When a user writes visualization source code, its structure can be extracted and its execution provides a visual result. Following this procedure over time and combining both components into an interactive meta visualization, provides insight into the evolution of the developed algorithm.

Our system was designed to flexibly support multiple back ends. It supports C++ and GLSL code as general-purpose programming languages, as well as Diderot [113], a domain-specific language (DSL) specifically designed for the development of scientific visualization algorithms. While C++ and GLSL have been utilized in many visualization applications for possible performance gains, DSLs provide benefits in usability and expressiveness, while reducing the complexity of the algorithm's source code with efficient syntax. Diderot specifically provides support for visualization-specific data and features, leading to compact and readable code. It further allows for parallel execution of the developed algorithm and thereby provides faster visual feedback. All examples and results in the remainder of this paper are either generated using Diderot, or a combination of C++ and GLSL code.

When analyzing the planned support functionalities to build a meta visualization which enables the exploration of a visualization's evolution, it becomes clear that our system requires access to the algorithm's source code and its visual result. Many visualization algorithms further depend on the definition of a proper parameter set to produce a meaningful output. We want to apply the same parameter changes to several different states of the source code evolution to compare their impact on the visual result. Therefore, input parameters need to be extracted from each algorithm to check which parameters can be applied. The toolchain of a programming language needs to meet two requirements to be supported by our visual analysis system:

- Facilities for extracting the visual outcome from the algorithm's execution
- Definition of input parameters to steer the execution and influence the runtime environment of the algorithm via external tools

When these conditions are fulfilled, the programming language can be integrated into our system and benefit from all the additional functionality.

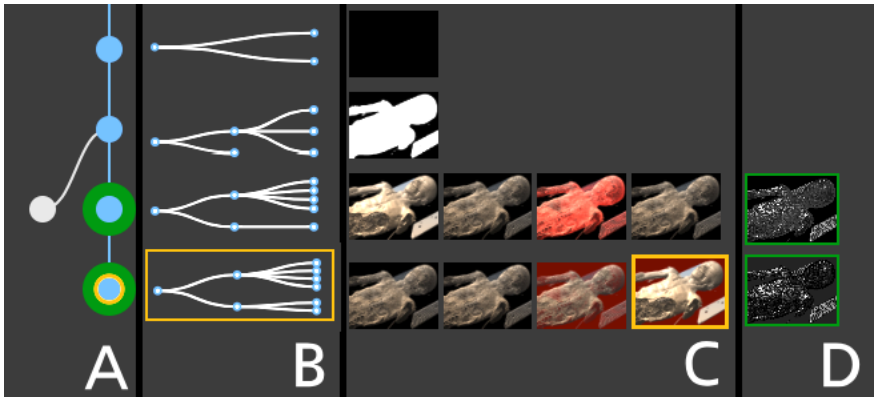


Figure B.3: Meta visualization. (A) The revision tree of all stored states is shown as a node-link diagram, where every revision is represented as a node and a link between two nodes illustrates the evolution from the upper to the lower node. The number of links at the bottom of each node represent the number of branches evolved from this state. States and their links are colored in blue for the current branch and grey for other branches. The current state is highlighted in orange and collapsed nodes are highlighted in green. (B) The static scope tree outlines the source code's structure and is shown for each state of the current branch. Each node represents one structural block of the source code. A link between two nodes represents that the scope to the right is nested inside the scope to the left. (C) The comparison of visual changes is presented in a juxtapositional view. For each state of the current branch, the visual result created by the source code is displayed as an image from a user-defined viewpoint. For states which are collapsed in the revision tree representation, result images are positioned next to each other. (D) A comparison image shows the visual variance between all result images inside such a collapsed node.

B.4 Exploring Visualization Source Code

In order to transcribe our main concept into a usable interface, we provide a meta visualization displaying the information of all visualizations created during the development process. All methods, their main ideas and comparison to alternative approaches, will be covered in the following. The resulting meta visualization, produced from an exemplary development process, is depicted in Figure B.3.

B.4.1 Automatic Revision Management

A central goal of our approach is to enable and support the comparison of several states during the development of a visualization algorithm. Many of our design choices draw inspiration from the VisTrails [26] approach for visualization pipelines. In order to access a given revision and all its information at a later point in time, it needs to be stored. Many applications make the user responsible for storing their progress by enabling them to manually save the current state. This approach only provides access to the last development state and is prone to system failure, whereas smarter systems create automatic backups. In practical software development, version control systems like Git [6] or SVN [3] are typically employed as they store the entire revision his-

tory. However, the process of storing an individual revision still needs to be triggered manually.

We internally utilize a version control system (Git) and draw inspiration from modern tools such as Google Docs that automatically track the version history of a document without user intervention. Instead of constantly interrupting their workflow, users can simply focus on the development process. We automatically track and store every code edit and create a new revision whenever the corresponding source code compiles successfully. All automatically stored states are displayed to the user in a node-link diagram to keep transparency over the process and visualize the development progress (Figure B.3A). Each node represents a development state and a link between two nodes describes the evolution from one state to the other, or, in other words, that the successor node (child) was created by modifying the preceding node (parent). Interacting with the visualization allows for intuitive switching between development states and further eases the comparison task. If the user switches to a previous state, which by definition already has a child, and continues coding, a new branch is created and the compiled state is represented by a new node.

B.4.2 Visualization of Algorithm Evolution

After having described our visualization of the revision tree, we will in the following describe the other visual components which compound our meta visualization: the visualization of source code structure, the visual result, and the difference image.

Result Image
Writing source code is an error-prone task and mistakes in the program can result in failure or incorrect visual results. The visual output can serve as a compact descriptor of the functionality and features integrated into the given source code state. It can be utilized to easily identify different feature sets and to add, remove and exchange them. Furthermore, having an overview over existing features can reduce redundant code and effort in reimplementing features. For each compilable revision, we execute the program and display the result image (Figure B.3C) aligned with the corresponding node of the revision tree and a representation of the generating source code (Figure B.3B).

Source Code Structure
While there are many different techniques for visualizing source code structures, as discussed in section B.2, we focus on its outline in the form of a static scope tree (SST). This representation highlights the nesting scopes (block structures) of the source code and is sufficiently compact while still conveying the main structural aspects of the code. In both programming languages featured in this paper, nesting scopes are defined by opening and closing curly brackets. The SST is visualized by a node-link diagram (Figure B.4B), where every node represents a code block and a link exists between two nodes, if one block is nested within the other. The tree depth represents the nesting level. If several source code files exist, the SST is computed for each of them, and their root nodes are linked as children to a new root node which represents the files' directory (Figure B.4C). While the SST provides a high-level overview for source code comparison, the actual differences between two given revisions are computed and visualized in a tooltip to highlight their code changes in detail.

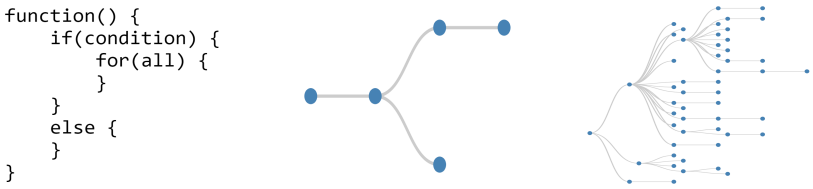


Figure B.4: Static scope tree. (A) Pseudocode. (B) SST extracted from A. (C) SSTs extracted from three source code files (from top to bottom: C++ source, GLSL, C++ header) and combined under a mutual root node.

Compression

Since every compilable development state is taken into account, the meta visualization can become very large. We therefore compact the revision tree based on a predefined comparison measure to enable the representation of several successive states within a single collapsed node. A node and its parent are represented by a single node, if both states are equal with respect to the chosen comparison method. The representing node inherits the children of all collapsed nodes and the visual results of all states being combined in such a node are displayed next to each other. If desired, the user can manually expand and collapse the node for detailed investigation. In a first iteration, we define the comparison measure as the similarity in source code structure. We assume that a structural change describes a major change in the source code. All successive nodes that represent states with the same source code structure are bundled in our default abstraction. The corresponding SST is only displayed once per bundle. One can easily think of other possible comparison measures, like defining thresholds for visual differences in the result image, number of code changes or performance differences. The question of which compression method is best suited for the exploration of visualization source code evolution is left open for future research. Alternatively, letting the user tag states of interest for comparison could be a good alternative to automatic compression approaches.

Result Comparison

The identification of subtle visual differences between the results of multiple development states can be challenging. For this reason, we provide an additional comparison view (Figure B.3D) that shows the per-pixel variance of the individual results for states collapsed within a single node. The resulting image is black if all compared images are exactly equal. The higher the color difference between the images in a certain location is, the brighter is this part of the computed image. Our approach of showing visual results next to each other and visualizing the small differences in an additional view benefits from the advantages of both a juxtaposition and explicit encoding [73]. It enables the user to identify visual differences on both the large and small scale. Combining the visual differences with the changes in source code and aligning them along the algorithm's evolution, provides the visualization developer with an enriched insight into the process that would require significant effort to be achieved with common development tools.

B.4.3 Parameter Management

Many visualization algorithms define several input parameters, which can significantly alter the visual result. Investigating the impact and interplay of individual parameters on different stages of an algorithm is a non-trivial task. Simple systems supporting this task can modify certain parameters on the application level and visualize the result directly. Some of them store the visual result of several parameter settings to ease the task of comparing different parameter sets. When the user wants to test if the change of a parameter has different impact on individual features of their implementation, these systems provide little support. The user would need to run the available tool for each state of interest, change the parameter settings in every instance, and compare across these instances.

Our system provides the possibility of freely exposing all input parameters of a state and mapping them to individual type-specific elements of the interface which provide convenient interaction facilities for parameter changes similar to common property sheets, using simple syntactic constructs specific to the host language. Based on this mechanism, we are further able to provide more intuitive, specifically tailored interaction elements for common parameters in the visualization domain. The virtual camera is one of the most common parameter sets in three-dimensional visualization, typically specified using an interaction technique such as the Arcball [180]. It controls the rotation of the camera around an object and can be easily expanded to support translation and scaling. We integrate this well-known technique via a plugin mechanism into our system. The plugin defines three keywords for the camera's position, look-at point and orientation. When the user utilizes these keywords as names for the corresponding parameters in the algorithm's source code, they are automatically coupled. We display an enlarged version of the current state's visual result for detailed exploration, that can be seen in Figure B.5. Interacting with the view automatically runs the algorithm with the new parameter set and updates the result view on-the-fly. This approach expands the visualization algorithm by a direct, interactive component without any extra effort. Our system provides the ability to automatically and transparently make other such interaction facilities available for developers to directly integrate into their algorithm.

To support the task of comparing the visual outcome of different parameter settings on several states, we perform the parameter change on all visual representations of states within the current development branch. This means that, for instance, moving the camera of the current state will automatically update all other states to the same camera position. The user can then explore whether states with different feature sets undergo different degrees of visual change. If a parameter is not present in a certain state, it is simply ignored. This semi-automatic approach enables the user to investigate the impact of a parameter change across all implemented feature sets. It further provides insight into the sensitivity of different features to certain parameter changes and into the robustness of investigated parameters in terms of their compatibility with multiple features.

B.4.4 System Interactions

Although all described methods can be utilized by themselves, they further benefit from being visualized next to each other and interconnected by user interactions. To provide

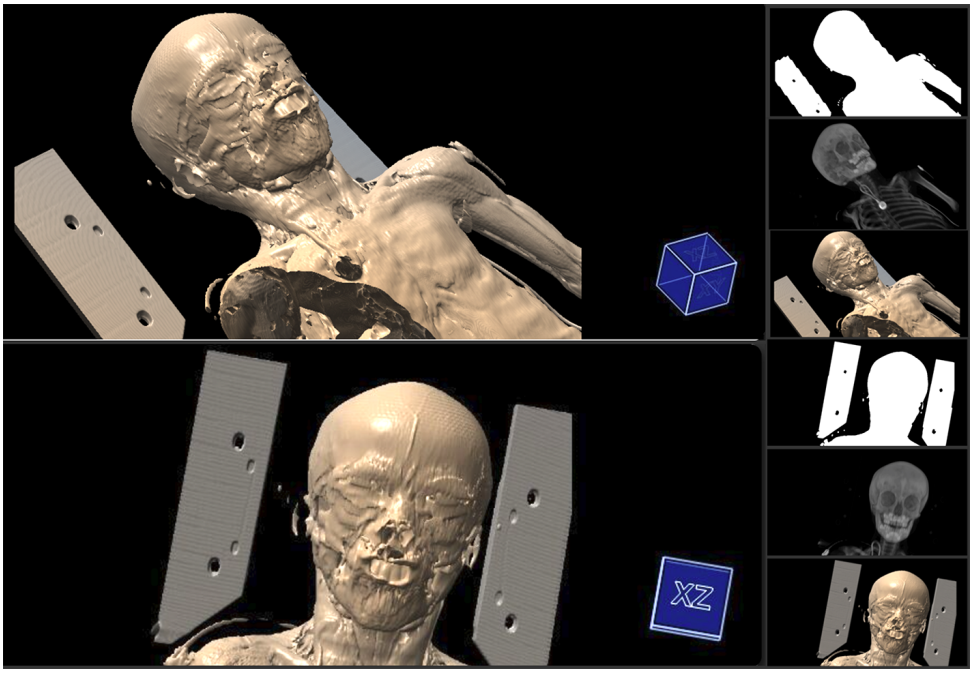


Figure B.5: The Live View enables direct user interaction including rotation, transformation, and scaling to investigate the visual result of a given state in detail. These interaction functionalities are automatically integrated, as soon as the developer utilizes the predefined keywords. The newly chosen viewpoint is automatically applied to all other visual results to enable consistent comparison between states.

a better picture of the possible interactions, we present an overview of our graphical user interface in Figure B.6.

The visualization of our revision management system, the evolution of source code structure, and the evolution of visual results all communicate progress over time. Their respective components are each related to certain states of the development process. For every compiling state of source code, a node in the revision graph is displayed, the source code's structure is visualized, and a view of the visual result is shown in an image. All three visual representations are aligned along a common time axis. This approach further emphasizes the evolution from one revision to the other in a clearer manner by following the time axis from top to bottom, as shown in Figure B.3.

We display additional overlays for a more convenient comparison of several states. When hovering over a result image, the system displays a tooltip which includes all source code differences between the hovered and the current state. To easily find these differences in the code editor, all lines of code that need to be removed and added to get from one state to the other are highlighted and their line numbers are displayed. This interaction mechanism allows for an in-depth inspection of the source code changes. The current state can be easily switched by clicking either on the result image of a state, or its node representation in the revision tree. Both elements, as well as the corresponding static scope tree, are highlighted, so that the user always knows which

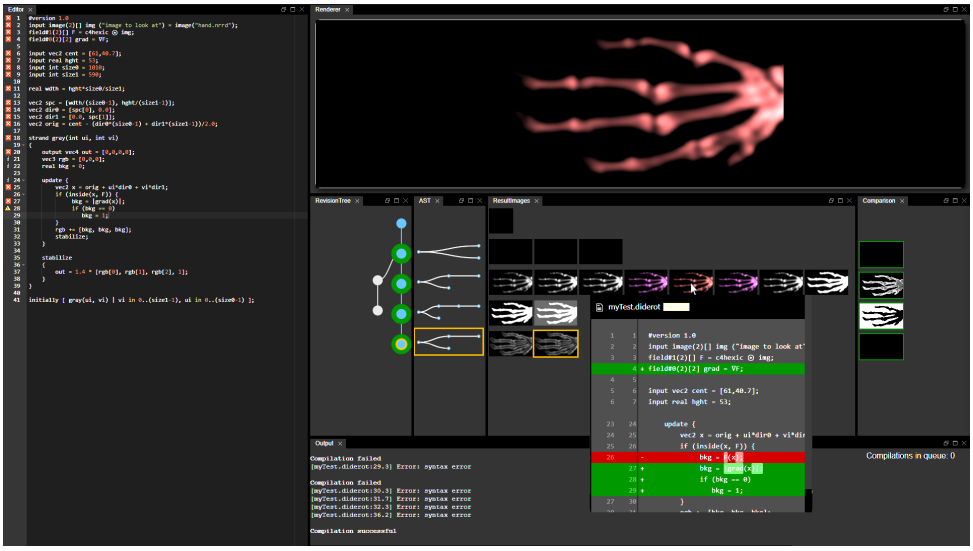


Figure B.6: Graphical User Interface. The interface of our application is inspired by common Integrated Development Environments. Every view can be individually positioned and resized based on the user's preferences by easy drag&drop interactions. The code editor is shown to the left and includes the source code of the current state's visualization algorithm. The whole evolution of the algorithm, up to the current state (highlighted in orange), is presented in our meta visualization. The top right view would normally display the visual result of the current state, but since the user hovers over one of the states in the meta visualization, its result is shown instead, enabling easy comparison between the two. Additionally, the source code differences between the hovered and the current state are directly visible in the overlaying tooltip. The bottom right view displays the compilers output after each compilation attempt.

state they are currently working with.

Since the revision tree collapses nodes if their source code has the same SST, it is necessary to provide the user with tools to investigate all available states. A right click on one of the collapsed nodes will expand all hidden nodes and branches. In order to keep the visual components in alignment, the SSTs and result images of the currently shown branch will be repositioned. Since such a displacement might confuse the user's mental map, hovering over a SST highlights all result images that share the respective SST. Following the same principle, hovering over a comparison image highlights all result images that were taken into account to compute it. The expanded nodes in the revision tree can be manually collapsed again by performing a right click on any of the nodes which share the same SST. Clicking on a node in the revision tree which is not within the current branch will switch branches and update all views to visualize the meta information of the selected branch. All nodes of the currently visualized branch are marked in blue in the revision tree, while all other nodes are colored in grey. It is important to keep these non-active branches within the visualization, because otherwise there would be no interaction available to activate them again.

When hovering over a result image, it is enlarged for easier inspection. It provides the user with a preview of what is to be expected, when switching to that state. Contin-

uous alternation of hovering over two images allows for easy comparison and detection of visual differences. To detect even the smallest differences, the comparison image is also enlarged when being hovered.

B.5 Implementation

Our application is divided into a server and a client component, each benefiting from different programming languages and execution environments, whose technical details will be briefly outlined in this section. We provide a short description of the components' functionality, communication, and the events being triggered by user interaction.

When setting up a machine for software development, many tools, like compilers, libraries, or an IDE, need to be installed. The chosen tools and developed source code often vary based on the underlying operating system or hardware used. Finding the right tools for a given task can be very time-consuming, especially for beginners in software development. We therefore chose a client-server architecture to relieve the user from setting up the environment needed for compiling and executing the source code on the client side. It enables us to run a large amount of compilations in the background without affecting the user's workflow. Our solution can therefore even be used on low-end clients.

In order to make our application widely available, we implemented the client side as a web application. In addition to standard web technologies (HTML5, CSS3, and JavaScript), we utilize the Ace Web Editor [1] as environment for writing visualization source code, and D3 [34] to create node-link diagrams within the meta visualization. The window layout is based on Golden Layout [7] and is inspired by common Integrated Development Environments enabling the developer to add, delete and move modules based on their personal preferences. The visual presentation of source code differences is handled by the diff2html library [5]. Our server is written in C++ and handles the compilation of source code, storage of states, visualization of results, and the communication with web clients. The communication channel between client and server is based on the JSON-RPC protocol. Data is requested via AJAX calls and exchanged in the JSON format. We further use Git [6] as the version control system by utilizing the functionality provided by the libgit2 library [9]. The current version of our application supports Diderot [113] as a programming language specifically designed for the visualization domain, as well as C++ and GLSL. While Diderot code is commonly manageable within a single source file, C++ and GLSL code can become quite large. We provide multi-file support to split up source code into several files. In order to support additional programming languages, the system requires access to a corresponding toolchain, the definition of commands to build both a library and an executable from the source code, and access to the result images. Everything else is handled automatically, e.g., error messages from the toolchain are forwarded to the client and source code is stored in Git.

In the beginning of the development session, the user chooses a programming language. Based on this choice, the server prepares the appropriate compiler and runtime environment to compile and execute all incoming source code. When the user writes code in the editor, our system waits until the user stops typing for a certain amount of time (default 1.5 seconds), before sending the code to the server and compiling it to an

application. The static scope tree is automatically extracted during the build process. If the compilation was successful, the source code is stored in the Git repository and the compiled executable is cached for future execution. The visual result is sent to the client along with the current state of the revision graph and the SST of the source code. If the compilation fails, an error message is sent to the client. When the user interacts with the live view, the visualized state is rerun with the new parameters on the server side and the results are sent to the client. The same parameter set is used to run every stored revision of the current branch and thereby updates all visual results, which are then sent to the client in an asynchronous manner. In order not to introduce any decrease in performance compared to common development environments, where the user would manually start the compilation process, we prioritize the compilation and update of new source code over older revisions. We realize this by utilizing a priority queue, where compilations are assigned the highest and updates the lowest priority. When two operations in the queue have the same priority, we perform the latest request first. For further performance improvements, older compilation requests in the queue can be dropped when a newer revision was successfully compiled. Parameter updates across the revision tree are only performed when the hardware resources are available. If the user wants to switch to another state in the revision graph, the server finds the given state by its unique ID in the Git repository and sends the source code to the client. In the same way, source code differences are computed between two revisions on the server side and sent to the client for display. When managing several source files, the user can switch between these files by clicking the corresponding tab in the code editor. We check if the file contains the content of the currently selected state and perform an update if necessary.

B.6 Usage Examples

Having described the individual components of our system and how their capabilities and interconnections employ our conceptual methods, we now want to illustrate the system's advantages over commonly used systems in real-world scenarios. As interactive processes are inherently difficult to capture in still images, we encourage the reader to also refer to our supplementary demonstration video. We present the implementation of two usage scenarios of our approach: a three-dimensional flow visualization and a visualization with stylized line primitives. We highlight the specific integrated features and discuss how they enhance the user's experience during the development process.

B.6.1 Flow Visualization

The user's task is to visualize specific properties of a vector field, starting from flow magnitude, over extremum lines, to normalized helicity. Since the given vector field has three spatial dimensions, the visual result shall be three-dimensional as well, so a ray casting algorithm will be used. In order to easily follow the development process and to get a better impression of how our system's support functionality looks like, we display all intermediate results of the now following description in Figure B.7.

The process starts with an initial state which only consists of Diderot's boilerplate code and a black image as its output (Figure B.7A). With the three-dimensional result

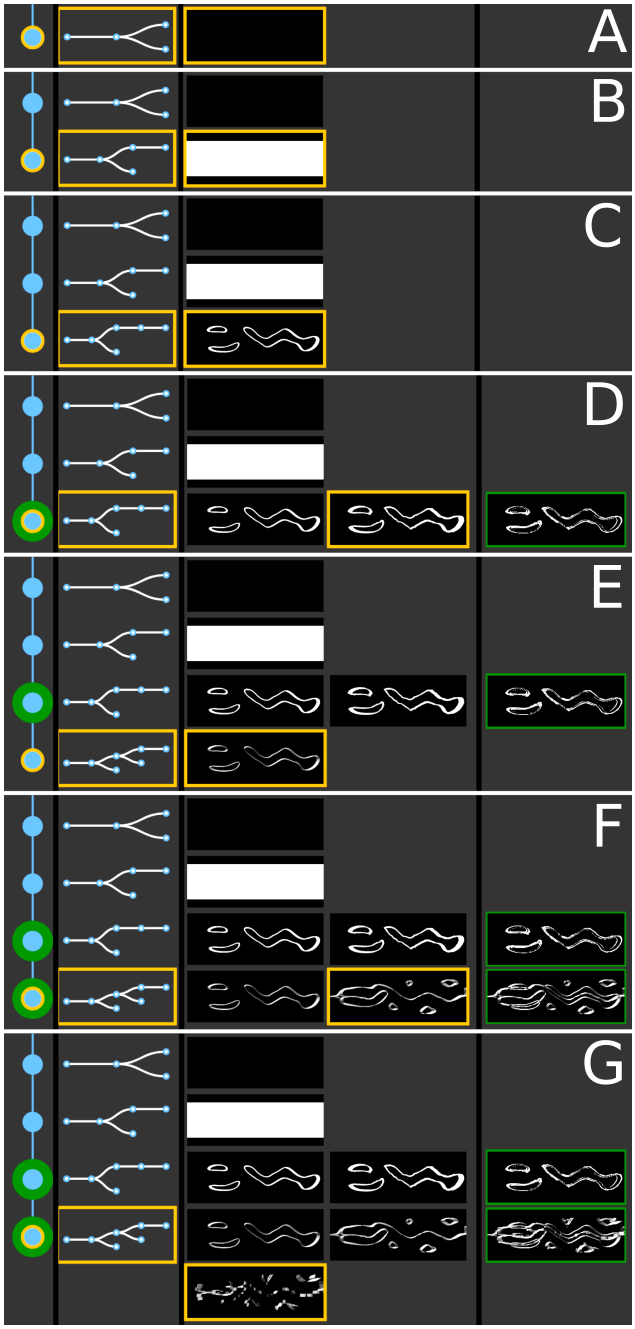


Figure B.7: Visualization of a visualization source code evolution. (A) Initial state. (B) Implement ray tracing and find volume. (C) Implement flow visualization based on flow magnitude. (D) Test trilinear interpolation instead of B-spline interpolation. The comparison image highlights minor differences. (E) Implement shading. (F) Change the flow feature from flow magnitude to extremum lines. (G) Test trilinear interpolation instead of B-spline interpolation.

in mind, we add camera parameters to the algorithm, cast a ray at every pixel of our camera's resolution and display data in white, when it is hit by a ray (Figure B.7B). At this point already, by utilizing the predefined parameter names for the camera's position, look-at point and up-vector, our application will automatically assign values to these parameters, allowing the user to zoom, pan, and rotate around the data set and select the best possible viewpoint. In order to gain a better impression of the flow, we compute the flow's magnitude and only display it for values above a constant threshold (Figure B.7C). At this point, we start experimenting with the interpolation kernel, which is responsible for creating a continuous field from our given input data (Figure B.7D). Diderot has convenient built-in mechanisms to change this reconstruction method. Investigating the result images clearly shows a much smoother flow when using B-spline interpolation compared to trilinear interpolation. The comparison image highlights subtle differences to the user. Now that the flow is already visible, we can improve shape perception by adding a light source into our environment and performing gradient-based shading (Figure B.7E). Since the basic visualization algorithm is functioning, we can start to integrate additional flow features. Visualizing surfaces around extremum lines and vortex structures is easily done by modifying the computations over the corresponding vector field (Figure B.7F). After these additions, we notice artifacts when experimenting with different interpolation methods (Figure B.7G), both in the case of trilinear interpolation and even when using Catmull-Rom splines. By hovering over the images of previous states, a tooltip reveals our last code changes and thereby highlights the different vector field computations. We check the maths and consider our algorithm as correct. The comparison to even earlier states ensures us that no drastic changes were made and the issue must be related to the interplay between flow features and the reconstruction kernel. Also, our basic algorithm seems to be correct, since the visualization of the flow magnitude does not show any problems. The only difference and possible cause of the issue in this scenario therefore seems to be the gradient computation. If we, as a user, had the necessary background knowledge, we would know that trilinear interpolation does not provide the continuity necessary to compute appropriate derivatives. Our application was therefore able to reveal the complex correspondence between interpolation kernels and gradient computation by displaying the visual results and source code differences in an explorable manner. Exposing such high-level relationships without explicitly knowing the reasons for their existence, requires the human-in-the-loop for further sense-making, but provides a hint to in-depth understanding of the algorithm's underlying functionality. The user can now further compare the results of functioning states and visual differences in all the implemented functionality, choose the most suitable version for their task and add further features if necessary. When investigating the evolution of the static scope tree over the whole process, it is noticeable that the source code structure expanded until shading was implemented, but stayed constant afterwards. This provides us with an indication that all main features being considered during this session require the same fundamental code structure in order to function, but mainly differ in the mathematical formulae being computed over the flow field.

B.6.2 Stylized Line Primitives

In our second example, we focus on efficiently drawing lines in a three-dimensional scene, by rendering them in 2D, but shading them as if they were 3D tubes [187]. Additionally, several styles can be applied to the lines to represent different features in the data. It differs from our previous example by being implemented in C++ and GLSL, and thereby utilizing multiple files – a C++ header, a C++ source and a GLSL file. Furthermore, the underlying source code and the corresponding source code changes contain many more lines of code (665 LOCs for the final state) than the previous example being written in a domain-specific language (72 LOCs for the final state). This scaling in data size comes with increased compilation times, bigger static scope trees and larger tooltips. The evolution of the algorithm's development is shown in Figure B.8. Since some of the source code changes being made are quite extensive, we focused on the most salient parts.

We skip the initial development involving data handling and the setup of the rendering process to focus on the evolution of the visualization techniques. We therefore start out with source code which already renders a red quad strip for each line in the data, following its curvature (Figure B.8A). All quads are, based on the user's viewpoint, computed and rendered in every frame. The definition of specific parameters in the code automatically provides the same interaction handling for changing the viewpoint as demonstrated in the previous example. In order to visually separate the lines from each other, we plan to add a halo to each line. We increase every quad's width by a margin and check for each fragment being drawn, if it lies within the line's width or exceeds it. If it exceeds the line's width, we paint the fragment black as a halo, otherwise we set its color to red (Figure B.8B). The visual difference becomes immediately apparent and provides us with a better understanding of the lines' paths and depths. In the next step, we want to give our lines a three-dimensional shape, which is why we define a texture on top of the quad, that contains the side vector, up vector and depth correction factor at each point (Figure B.8C). While all other source code changes are made in the GLSL file, this change is taking place in the C++ source file. Although the profile texture is now correctly set, the visual result appears to be the same and a look at the difference image confirms that they are identical. The change is not visible, because the texture is not yet applied and no shading is implemented. We implement Phong shading and now the lines appear as shaded three-dimensional tubes (Figure B.8D). Unfortunately, it is not as easy to perceive the depth of the tubes in the scene, especially if they do not intersect in the result image. We therefore improve the rendering by integrating depth enhancement (Figure B.8E). Interacting with the visual result and comparing it to the previous states demonstrates the advantages of the newly implemented feature. The tooltip shows that this feature only consists of a few lines of code and can therefore easily be added to other visualization algorithms as well. As a last step, we draw arrows on the tubes to provide the viewer with the information on the lines' flow direction (Figure B.8F). The static scope tree of the final state is shown in Figure B.4C. It provides an overview over the source code by clearly showing that it consists of three source files (because the root node has three children), where the C++ header file only consists of a single structural block, the GLSL file is a bit more structured, and the C++ source file contains the most structuring elements.

Looking at the performance of our system in this example, the compilation of source

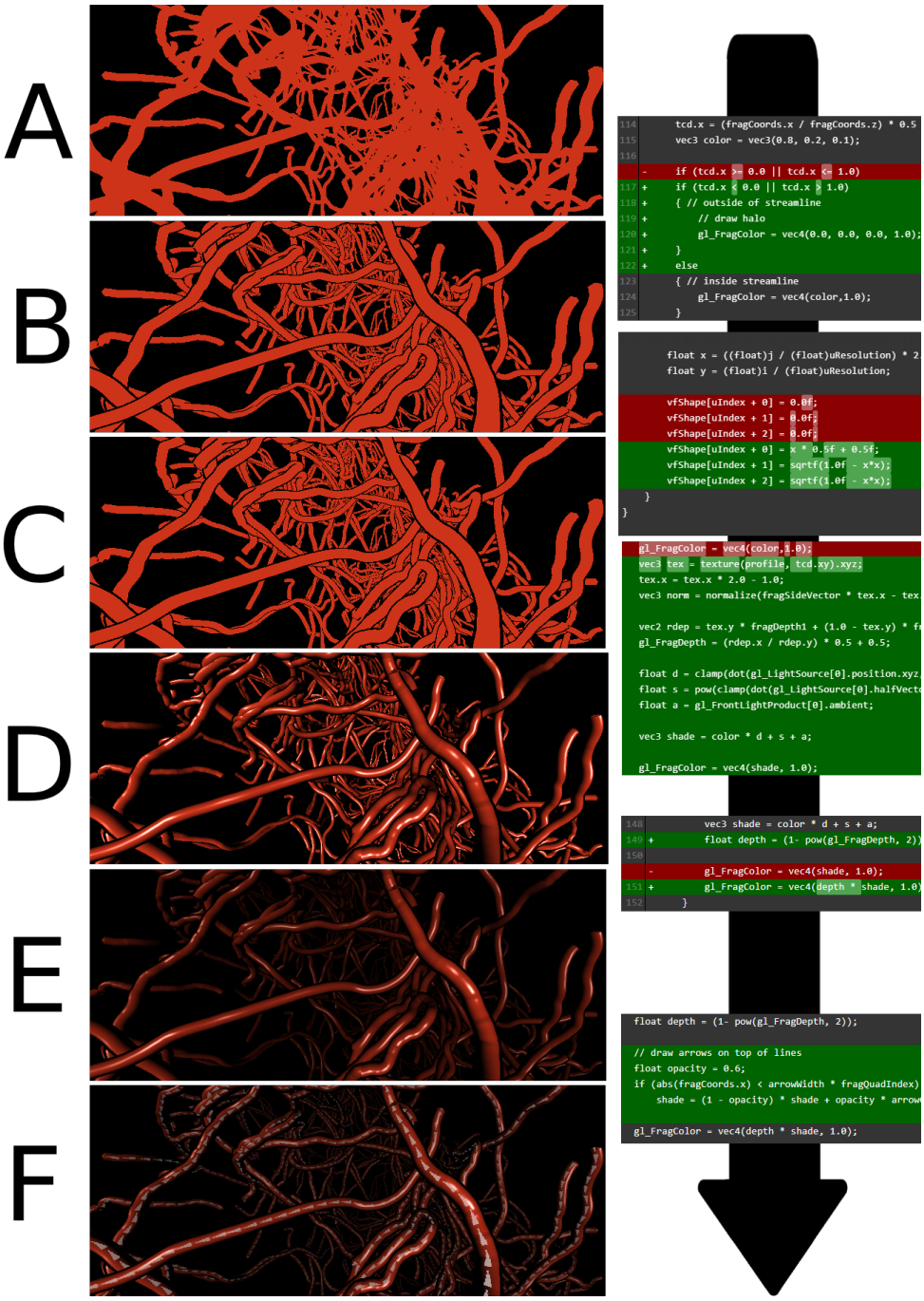


Figure B.8: Visualization with stylized line primitives. Only extracts of source code differences are shown. (A) Implement basic rendering of line primitives. (B) Add a black halo to all lines. (C) Change the three dimensional data profile of lines. (D) Implement shading. (E) Implement depth enhancement. (F) Draw arrows on lines, to describe the flow direction.

code took around 8 seconds, if it was successful. If it fails, the compilation process is canceled much faster and provides a description of the error. Since the compilation takes place on the server side, the user does not experience any slowdown in their coding workflow. Changing the viewpoint on the visual result took around 100ms per image. This means that when the user is only interested in the most current history (last ten states), their result images can be computed within a second, since images are updated starting from the latest state. Updating images for 100 states would take around 10 seconds, with the client view being progressively updated as new images become available. The number of triggered compilations and created states vastly vary based on the given task and the user's typing behavior. While typing, 10–15 compilations can be triggered every minute, of which 4–5 will be executed, while others are ignored in favor of newer revisions. Most of the time, only one of these compilations is successful and creates a state in the visualization. Users ended up with 30-50 states when working on this example.

We have shown that our novel approach is able to reveal complex correlations between the visual result and the underlying source code of a visualization algorithm. It provides the user with direct feedback, enabling them to discover implementation problems as soon as they appear. It opens up new ways of comparing visualization algorithms by utilizing novel viewpoints onto the available meta data and thereby generates greater knowledge about the algorithms themselves. Revealing these relations between algorithmic techniques, mathematical formulas, implementation in source code, and visual outcome can greatly benefit the task of comparing visualizations on all these levels and be especially beneficial for teaching visualization algorithms to students.

B.7 Evaluation

Given the subject matter of our work, we performed our evaluation in two rounds and in the form of an expert review [200]. In the first round we gathered qualitative feedback on the functionality and usability of our presented framework using the Diderot language. We selected 4 experts from academia with different specializations in the field of visualization. All participants rated their expertise in visualization between knowledgeable and professional. They had extensive experience in writing visualization source code (on a monthly to daily basis), but none of them were intrinsically familiar with Diderot. Based on the initial feedback, we improved our system to support C++ and GLSL code, handle multiple source files, and cache compiled source code for faster state switching. We then conducted a second round of evaluation to gather feedback on the improved state of the system by selecting 4 new experts with similar experience and an example using C++ and GLSL. All participants were familiar with different existing toolchains, ranging from C++ IDEs to web based development platforms and we asked them to assess our system in the light of their experience. None of the experts are co-authors of this paper, participated in the development of our system, or had used it prior to the review.

Following the guidelines of Tory and Möller [200], the evaluation was split into several sessions, interviewing one participant at a time and following the same protocol: At first each participant filled out a sheet of information describing their personal

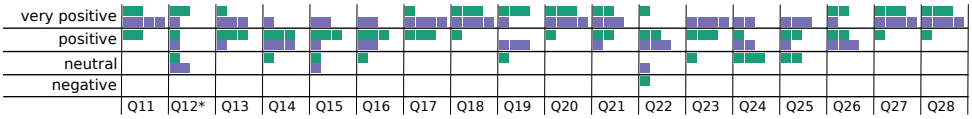


Figure B.9: Illustration of the participants' answers to statements of our questionnaire on a 5-point Likert scale. Very positive feedback was reflected by either strong agreement to a positively formulated question, or strong disagreement to a negatively formulated question (marked by an asterisk). Very negative feedback was not given by any participant. The color reflects the participants group, green for the first round and purple for the second round.

background and expertise in visualization algorithm development. They were further introduced to the concepts of our approach and the application's functionality. All interaction possibilities were summarized on a two-page handout given to the experts. We not only wanted to assess the functionality of our application, but also the meta visualization's ability to communicate information about the visualization algorithm's development process. We therefore provided the participants with the visualization examples from section B.6, depicted in Figure B.7 (round 1) and Figure B.8 (round 2). Based on this initial state, the participants were asked to perform a set of simple tasks and verbalize their thoughts and reasoning behind their actions (think-aloud protocol). They were allowed to ask questions about the tasks and application at any given time. The tasks were performed without time limit and designed to encourage the exploration of all aspects of our system. Tasks like *"Find a state with a bug"* and *"Figure out, what the reason for the bug is"* required the user to analyze the whole development process on the visual and algorithmic level. Asking the participants to *"Change the code, so that it produces a different output"* not only made them actively develop within our framework, but created personal and for us unexpected results that led to diverse usage of the provided system. When the users completed the tasks, they received a 28-statement questionnaire to answer on a 5-point Likert scale. It covered the difficulty and suitability of the given tasks, general functionality of the application and assessment of individual components (e.g. *"I found the abstract code view was giving a good impression of the source code's complexity."*). It further included the ten statements of the System Usability Scale (SUS), which allows for the uniform comparison of different systems in terms of usability and user satisfaction. Last but not least the participants were asked to openly summarize what they liked/disliked about the system, what they would like to change about it, and how they rate its practical impact. The participants' feedback to each question, except for the SUS (Q1-Q10), is illustrated in Figure B.9. All the answers can additionally be found in the supplementary material.

While we cannot discuss all the feedback in detail here, we want to highlight some of the most interesting insights gathered through our evaluation. All participants rated the system's functionality as useful (Q11) and rated the availability of such a system as beneficial for their work or studies (Q13). When assessing the availability of other tools with similar functionality (Q12), participants named Git and Shadertoy (in round 2), but also mentioned the superiority of our approach given the extended functionalities it provides. The participants mentioned several times that they would like to use our system as a rapid prototyping tool, or as a platform for teaching visualization algorithms to students, but that larger software projects would require additional features to support

the development of other parts than visualization or rendering modules.

Looking at the different components of our system, the result view was easy to understand (Q18), easy to use (Q19) and helpful in completing the given tasks (Q20). The convenience of seeing all the visual outputs, investigating the source code differences by simply hovering over them and the intuitiveness of switching back and forth between these states, were highly praised by several participants. The fact that changing the viewpoint in one state also updates the viewpoint in all other result images, was recognized as being helpful for being able to compare the results to each other. One participant raised the reasonable doubt that finding a certain state within the result images after its parameters were changed might be difficult, since it is then harder to recognize as being the same state. The revision tree gathered a more mixed feedback, with most experts finding it easy to use (Q23) and, to a lesser degree, finding it easy to understand (Q22). Several participants mentioned that the knowledge of how version control systems work is beneficial in understanding the visualization and using it. We believe that most of the problems with this view originate from the fact that several branches can be collapsed in the same node and only results for the current branch are shown at a time. Observing the participants' interactions with the system showed that this behavior was only a limitation at the first attempt, but once they had familiarized themselves with the functionality, there was a noticeable improvement in how fluently they used the system. Interestingly, feedback to the static scope tree representation was more positive for experts in the second round, where the trees were larger and showed the structure of multiple files. It seems that these experts had an easier time understanding the representation (Q24) and judge the source code's complexity (Q25) within several files. It was mentioned that code structure allows only for partial assessment of the source code's complexity, since for example a single line of code can perform complex computations, and a single block can contain one or a hundred lines of code. Additional code metrics would be necessary. While some participants did not look at the SST at all, others wanted to use it but felt a lack of integration with other features, as well as difficulties comparing multiple trees. Based on this, we believe that additional features like using the SST as a code navigation tool, highlighting structures on code edits, and visualizing structural differences, can increase the utility of such a representation. Alternatively, more direct visualizations of the source code structure, e.g., in the form of a pixel-based visualization, should be taken into account.

The generally good responsiveness of the system when switching between states and investigating the different visual outputs and source code changes was positively mentioned by several participants. The feedback further improved in the second round, where caching of states led to even faster response times. The intermediate SUS score went up from 78.75, which corresponds to an adjective rating between "Good" and "Excellent" [21], to 88.75 ("Excellent"). This confirms our impressions and is a promising starting point for seeing our application in daily action. Quotes like "The programming task becomes more explorative and free." and "It is actually a lot of fun to go through the changes." emphasize our framework's investigation capabilities with respect to the algorithm's evolution. They show how our tool might positively influence the development workflow of visualization developers in the future.

The most sought after features were the comparison of states among different branches and the availability of the difference image for two specific states instead of showing it for collapsed groups only. Participants asked for a better link between

the source code and the tooltip displaying code difference. This includes clicking on lines in the tooltip for navigation purposes, as well as highlighting code differences in the editor itself. Other feature requests included more control over how the states are grouped within the views, a merge tool, which combines the source code of two different states into one, and a way of deleting revisions. Otherwise, additional performance measures, more tools commonly seen in professional IDEs, and general customizability were requested.

B.8 Discussion

Our proposed approach for designing a meta visualization system to provide insight into the evolution of the prototyping process of a scientific visualization algorithm combines several interesting research fields. By utilizing methods from software visualization, visualization of time-oriented data, visualization of visualizations, and the analysis of the visual parameter space, it explores and raises new research questions. Based on the feedback we received, we believe it is a promising concept to provide visualization developers with novel information about their own research projects and a tool to investigate their own algorithm development. It further allows for an easier comparison of individual features to replace, improve, or combine them and enhance the visualization algorithm.

Ease of Use
By automatically compiling source code and visualizing the result in a live view, we follow in the footsteps of Vega-Lite [170] and ShaderToy [13]. We not only apply the concept of an automatically updating result view to a programming language which requires compilation in the first place, but further visualize all previous results to emphasize on the result evolution. The fact that all intermediate steps are stored automatically further integrates with the idea of focusing the user's attention on the programming task.

The comparison of visualizations is a complicated task which has not yet been solved in a general manner. We facilitate the comparison of visualizations on a large and small scale by providing a juxtaposition and explicit encoding of all visual results and displaying their correlation to the underlying source code. We thereby increase the awareness of how visualizations evolve and in which way different features apply visual changes to the result. Our novel approach of applying the same parameter set to all visual results of the algorithm's states, further improves the comparability and allows for the visual exploration of the parameter space.

Scalability
While our system works well for developing prototypes for scientific visualization algorithms, it benefits from certain conditions given in this specific scenario. The visual result is commonly aligned to the original spatial dimensions in the data, which is often achieved via a virtual camera model projecting the spatial data into two-dimensional space. This alignment results in better comparability of different states based on images than in abstract visualizations, where vastly differing mappings from data to the visual result exist. Although the evaluation of our system has shown the experts' in-

terest in utilizing our approach in other fields like information visualization or web development, it is unclear how the comparison task scales to these scenarios in a general manner. Additional study of this subject is required to extend our approach to other subfields in visualization.

Utilizing Diderot as a domain-specific language allows for comparatively short algorithms, that can be handled within a single source code file. When handling C++ and GLSL code, the structure and source code changes of multiple files need to be visualized, which results in both larger SSTs and longer tooltips. Scrolling along the time axis and tooltip does not scale indefinitely and if SSTs become very large, it is harder to make detailed comparisons between them. For example, SSTs in our use cases had a maximum depth of 5 and 6, and a maximum number of nodes of 7 and 51 respectively. In comparison, ParaView [17] as a full-fledged visualization application has a SST depth of 17 and 40000 nodes in its core alone. An additional overview visualization, or a different visualization approach in general, would be necessary to support large multi-file applications with a lot of source code and many structural code changes.

We can reduce possible delays in the visual feedback by utilizing parallel execution capabilities of programming languages and compiling and executing several different states of the visualization algorithm at the same time on the server side. Looking at large scale development projects, compilation time and runtime become increasingly limiting factors for the visual support given. Since our system is built around short-term visual feedback and runs several revisions of the source code with possibly several parameter sets, the outlined benefits decrease for compilation- and runtime-heavy visual applications. While the revision tree, SST, visual result and source code difference are still available, the live view and parameter changes on older revisions would be delayed or might not be created at all within the time frame given between revisions.

Future Work

While creating a visual result from each compilable source code state includes all the interesting cases that create visual changes, it might produce many unnecessary results, take up a lot of screen space, and provide only little information. We try to counteract this issue by bundling results to compress the time axis of our meta visualization. However, at present we do not provide any interaction technique to remove revisions from the view, highlight interesting ones, or bundle states based on the user's interests. Solutions for the given tasks would be necessary when using our application for an extended period of time and when many revisions are created. It would be interesting to explore which other measures than the SST similarity could be utilized as factors for revision bundling and how such measures comply with the users' intent. Possible measurements might be computations of visual differences, number of code changes, or performance differences. Additional user interactions for tagging, removing or grouping states of interest might provide a good alternative to automatic approaches.

In the same manner, the difference image provides automatically generated information for improving the task of comparing several visualizations to each other. The exploration of other measures than image variance that are able to quantify other aspects of image difference, or improve the comparison task in different manners, might be a fruitful research area that we want to continue to investigate. We will further concentrate on the question of how such similarity measures can be correlated to source code to localize code constructs and their impact on the visual result.

While certain issues of an implementation might only become apparent for particular parameter sets, others can depend on the input data. Methods to compare and investigate the behavior of several algorithms on multiple input data sets need to be studied. Additionally, user tasks vastly vary, so that different supportive information about the source code and visualization need to be provided. Such options for task-specific customization for visualization experts need to be analyzed and integrated into our system. Since all the important information describing the visualization algorithm's evolution is stored on a server, many more interesting applications like building and sharing visual notebooks, or collaborating with multiple users on the same visualization algorithm come to mind. Our web client will be publicly available in such a way as to enable programmers to easily experiment with our interactive environment.

Lessons Learned

Our system gives a direction for future development of visualization algorithms. Working with the system and evaluating it with experts in the field revealed several points of interest for future implementations of similar tools.

The accessibility of the system as a stand-alone website is particularly useful for teaching purposes, since no setup of applications and no prior knowledge is required to get the system running. The opposite is true for experts, who often have a running toolchain in place, have projects stored in existing repositories, and are used to their programming environment. They would greatly benefit from the visual support system being independent of the code editor and storage solutions. One could think of our system getting access to an existing repository and visualizing all available revisions, either as a web-based tool, or as a plugin to existing IDEs.

The evaluation made clear that different users have varying opinions on feature behavior. For the revision tree, some users preferred having the latest state at the top, instead of the bottom of the visualization. Users did not agree on if the source code difference should be shown from the hovered to the current or to the previous state. It could further be displayed vice versa, from the previous to the hovered state, which might depend on the task given. Some participants mentioned that the system was trying to update too often, although they were not finished editing the source code yet. These examples show that the system is required to provide extensive customization options to adjust the interface and feature behavior to the user's preferences.

The decision of using a modularized interface turned out very useful in terms of options for customization and extensibility. Following a similar approach on the server side allows for easy extension to support multiple programming languages. Overall, while our current prototype received positive user feedback and the proposed visualization and interaction methods were appreciated, we fully acknowledge that our system currently lacks several usability and customization features commonly found in professional IDEs. However, we plan to continue the development of our approach, with a specific focus on addressing its applicability to larger scale software projects.

B.9 Conclusion

We presented a novel approach for designing a meta visualization that enables the comparison of visual results of scientific visualization algorithms and their underlying

source code at the same time. This concept yields additional insight into their relationship and thus enables programmers in the field of visualization to find correlations between visual changes and differences in the algorithms themselves. Our approach supports programmers during the prototyping phase in keeping track of their development, while relieving them from repetitive tasks and thereby increasing their productivity. The frequent task of switching between different development states and comparing their visual outcome has been simplified to a one-click action by providing direct user interactions in the meta visualization. The problem of finding a given state with specific features is further supported through state identification via result images. We also showed how external functionality can be linked into the developed visualization algorithm, by providing an interactive view of the visual result and the on-the-fly coupling of specialized interaction functionality with program parameters. These parameters are applied to all states of the development process to enable the instant assessment of their impact.

B.10 Acknowledgements

The research presented in this paper was supported by the MetaVis project (#250133) funded by the Research Council of Norway.

B

Paper C

SplitStreams: A Visual Metaphor for Evolving Hierarchies

Fabian Bolte[1], Mahsan Nourani[2], Eric D. Ragan[2] and Stefan Bruckner[1]

[1] University of Bergen, Norway
[2] University of Florida, United States

Abstract

The visualization of hierarchically structured data over time is an ongoing challenge and several approaches exist trying to solve it. Techniques such as animated or juxtaposed tree visualizations are not capable of providing a good overview of the time series and lack expressiveness in conveying changes over time. Nested streamgraphs provide a better understanding of the data evolution, but lack the clear outline of hierarchical structures at a given timestep. Furthermore, these approaches are often limited to static hierarchies or exclude complex hierarchical changes in the data, limiting their use cases. We propose a novel visual metaphor capable of providing a static overview of all hierarchical changes over time, as well as clearly outlining the hierarchical structure at each individual time step. Our method allows for smooth transitions between tree maps and nested streamgraphs, enabling the exploration of the trade-off between dynamic behavior and hierarchical structure. As our technique handles topological changes of all types, it is suitable for a wide range of applications. We demonstrate the utility of our method on several use cases, evaluate it with a user study, and provide its full source code.

C.1 Introduction

Hierarchically structured data can be found in many places and has been addressed by many visualization techniques. Ancestry, taxonomy, topology, company staff, file systems, text articles, source code, and population data are just a few examples of data with inherent hierarchical structures that can be represented by a tree with parent-child relationships. Data that does not have inherent hierarchical structure, such as

This article was published in IEEE Transactions on Visualization and Computer Graphics (2020).

scalar data [87] and large text corpora [58], can be clustered to build a hierarchy for providing a better overview and understanding of the data. The corresponding trees have previously been visualized by explicit and implicit methods in two- and three-dimensional space, as well as hybrids, in all kinds of visual layouts [175]. Visualizing the evolution of such data, its hierarchical structure, and temporal changes requires the integration of time as yet another dimension. Several approaches have utilized existing tree visualization techniques to display each timestep in either a juxtaposed layout or as an animation. Further efforts have been made to optimize these layouts with respect to stability of object positions over time for preserving the users' mental map [78, 184, 188, 190, 203, 207]. More recent approaches began to adopt the ThemeRiver [81] metaphor to convey the evolution of tree nodes in hierarchically structured data by individual streams [56, 121, 137, 223]. All these methods are either limited by only supporting a static hierarchy over the whole time series [56], only allowing for data where each parent's value is sufficiently larger than the sum of its children's values [137], or only allowing for a subset of trackable changes in the hierarchy [121]. While these techniques do a good job in presenting hierarchical changes over time, they suffer from deficiencies in conveying the hierarchical structure at specific timesteps when no hierarchical changes are in view and create ambiguities in the interpretation.

We present a novel visual metaphor for representing time-dependent, hierarchically structured data in a static visualization. We take a nested streamgraph [137] as a basis, introduce *splits* to cut the streams at certain points in time, and add a horizontal margin. By increasing the margin based on a stream's hierarchy level, we reveal the underlying hierarchical structure of the data. We enable fine-grained control between visual continuity of individual streams and the visual clarity of hierarchical structures at a given point in time. The presented approach allows for a clearer representation of hierarchies even in cases when color cannot be used to encode the hierarchy level. In contrast to previous approaches such as Chronicler [223], our method handles complex hierarchical changes, like a node becoming the parent of its ancestor, in a consistent and unambiguous manner. We propose a novel visual encoding for such cases and, in conclusion, cover all hierarchical changes that occur in the visualization of hierarchically structured data over time. Our main contributions can be summarized as follows:

1. We introduce a novel visual metaphor to emphasize the data-inherent hierarchical structure and its changes over time. Our approach is based on simple and clear shapes that are capable of conveying the hierarchical structure independent of the color scheme applied.

2. We conducted a user study to evaluate users' performance in analyzing hierarchically structured data with the help of treemaps over time, nested streamgraphs, and SplitStreams and demonstrate that our approach successfully addresses deficiencies of previous methods.

3. We publish our implementation as an open source library for easy reproduction of existing visualization techniques like one-dimensional treemaps over time and nested streamgraphs, as well as the exploration of novel visualization techniques introduced by this paper.

C.2 Related Work

Our visual metaphor allows for the smooth transition between one-dimensional treemaps and stream-based, time-dependent visualizations. This approach combines the advantages of both techniques and allows for novel visual layouts to emphasize hierarchical changes based on the application and user task. The visualization of static and dynamic hierarchies has been widely studied and inspired our approach.

General Hierarchies: A multitude of techniques for the visualization of static hierarchies have been proposed in previous work. Treevis.net [175] summarizes many of these approaches and provides search and filter functionality. The authors further categorize methods into explicit and implicit visualization techniques, as well as hybrid forms. Explicit methods are mostly based on a node-link diagram, where every data item is represented by a node (e.g., a circle) and relations among these nodes are presented by a link (e.g., a line connecting two circles). Implicit methods, on the other hand, are not required to draw links between nodes, but utilize positioning of individual nodes to represent hierarchical relationships. Treemaps [107] are one of the most influential examples of implicit hierarchy visualizations, representing nodes as rectangles and nesting child nodes within the space of their parent element. The space is vertically or horizontally split, creating rectangles proportionally sized to the numerical values of the child nodes. Significant research efforts have been devoted towards optimizing many aspects of both implicit and explicit techniques, including considerations such as layout and aesthetics. One particular approach we build on are one-dimensional treemaps [108], which nest nodes inside their parents, but always split the space along the same dimension. As demonstrated by ArcTrees [149], this approach frees the second screen dimension to represent additional information, such as the temporal evolution of the hierarchy.

Juxtaposed Hierarchies: For the visualization of hierarchies over time, one can in principle use any existing visualization method to display a static hierarchy in each timestep, either in a side-by-side (juxtaposed) manner or using animation. While TimeTree [46] displays a node-link diagram and provides a slider to navigate through time, TreeJuxtaposer [148] integrates node-link diagrams in a juxtapositional manner. Isenberg and Carpendale [96] analyzed tree comparison tasks for juxtaposed tree layouts in an interactive multi-user setup. Several works have stabilized the positioning of individual data items in treemaps and optimized the layout strategy for easier comparison and tracking of items [78, 184, 188, 190, 203, 207]. Vernier et al. [211] performed a quantitative comparison of 13 different treemap layouts and collected a benchmark dataset consisting of 2720 evolving hierarchies for that purpose. While juxtaposed trees provide a good understanding of hierarchical structures at a given point in time, they lack a clear representation of time-related changes. They require the user to keep a mental map of all nodes and track their position and information encoding (e.g., color, size). Therefore, such techniques fail in conveying the evolution over time for larger trees and longer time series.

Static Hierarchies: Several approaches deal with data where values associated with nodes or links change dynamically, but the hierarchy stays constant over the whole

time span. SemaTime [185] and Timeline Trees[40] display time-dependent information for each leaf node of a static hierarchy. Burch and Weiskopf [42] visualize dynamic values along the links of connected nodes. Based on the ThemeRiver metaphor [81], and extensions like Byron and Wattenberg's geometry and aesthetic optimizations [43], similar approaches have been developed to visualize data values over time together with their hierarchical structure. BookVoyager [218] displays the hierarchy explicitly, as an indented tree, separated from the stream-based visualization. TouchWave [25] implicitly integrates the interaction with the hierarchy into a streamgraph. Both approaches can be utilized to hide individual streams for better readability and scalability. HierarchicalTopics [64] compute a meaningful hierarchy for non-hierarchical data to utilize the same interaction techniques. MultiStream [56] integrates multiple interaction methods to improve the focus and context awareness, enable linking and brushing, and tackle scalability issues with respect to time and hierarchical complexity. However, these types of methods are limited to the visualization of a static hierarchy. Our goal is to represent hierarchical changes over time in addition to the evolution of data values.

Dynamic Hierarchies: In the tree-ring metaphor by Therón [198], nodes of a tree are placed on concentric circles based on their date of addition to the tree, where larger rings correspond to later points in time. Since streams provide an intuitive visual representation of changes over time, several methods adopt this approach to visualize dynamically changing hierarchies. Outflow [224] draws streams between nodes of individual timesteps and applies a hierarchical clustering to address scalability issues. Burch et al. [41] draw an indented tree at each timestep and connect related nodes via links. Based on narrative charts [146] where elements split and merge over time, Textflow [57] visualizes hierarchical relationships as contours in the background of storylines. Tanahashi and Ma Cui [192] optimize such layouts to create visually pleasing results and Liu et al [135] further emphasize the representation of hierarchical structures. Cui et al. [58] introduce specific visual encodings to improve the understanding of hierarchical changes in stream-based visualizations, as well as tree cuts, which define the visibility of nodes at each time step.

Nested Tracking Graphs [137] introduced a stream layout similar to ThemeRiver [81] for hierarchically structured data by nesting streams inside each other. This representation allows for additions, deletions, merges, and splits in the hierarchy, but can only be applied to data where every node has a significantly larger value than the sum of values of its children. This requirement ensures that the hierarchical nesting is visible. Temporal treemaps [121] extended this approach by a hierarchy-aware ordering for the reduction of edge crossings and visual cushions to support data where parents inherit their values from their children. The presented method is limited in the number of possible hierarchical changes, because only moves, splits, and merges of streams along siblings are taken into account. Movements across hierarchy levels, where a node changes its parent, are not visually represented. Chronicler [223] visualizes the evolution of source code and allows for the movement of nodes across the hierarchy. While all these stream-based techniques manage to visualize changes over time, they suffer from limited clarity in conveying the hierarchical structure at a given timestep. This issue becomes more apparent as the number of nodes or the number of timesteps increases. As demonstrated in Figure C.1, the main problem is that the hierarchical nesting of streams only becomes visible at times of hierarchical change. As long

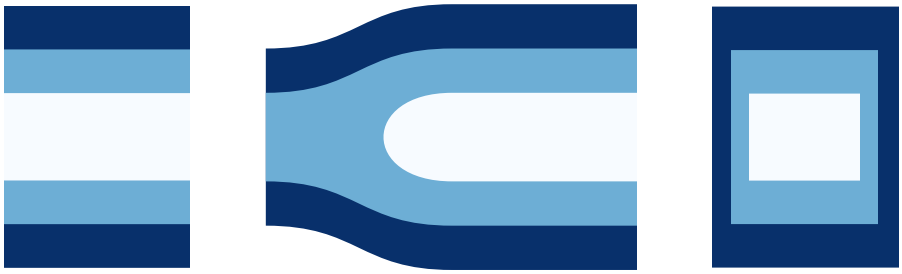


Figure C.1: Perception of hierarchies. (a) If the hierarchy is only encoded by the nesting of streams, it is not clear if the visualization shows three streams (white contained by light blue, contained by dark blue) or five individual streams. (b) When a hierarchical change is present, the two-dimensional containment provides an intuitive understanding of the nested structures. (c) In our approach, we split the streams and introduce margins to the visualization, to clearly represent hierarchical structures at any given point in time.

as no hierarchical change occurs, the number of individual streams and their respective nesting is only conveyed by color, but the interpretation of shapes can be ambiguous. In order to get a complete picture of the hierarchy at a certain timestep, at least one hierarchical change for every single stream needs to be visible, which negatively affects the readability when a large number of timesteps or nesting levels are present. We therefore propose a new visual metaphor which can not only reproduce these existing visualization techniques but can further emphasize the hierarchical structure at every single point in time without entirely relying on color coding to represent the depth of a node.

C.3 Overview

C.3.1 Data

We work with hierarchically structured data in which both the values and the underlying hierarchy may vary over time. For consistency with previous work, we refer to elements within the hierarchy (*tree*) as *nodes*. Each node has a value, exists for a certain period of time, and maintains a parent-child relationship (*link*) to other nodes. Every node can have an arbitrary number of children, but at most one parent. Nodes which have the same parent are called *siblings*. Nodes without children are referred to as *leaves*. A node without a parent is called *root* and defines the highest hierarchy level. In case several root nodes exist, we create an artificial root as the parent of all root nodes. The *depth* of a node is the length of the shortest path between this node and the root. A *hierarchy level* describes a set of all nodes with the same depth.

In order to visualize the mapping between nodes from one tree to the next, tree changes need to be defined. Every node of a tree therefore requires an identifier. This ID can be unique throughout all timesteps to clearly identify the existence of a node at each point in time. Alternatively, the ID can be unique to the tree of the current timestep only, in which case every node needs to store the ID of its predecessor and/or successor

in addition to its own ID. Nodes that are added to the tree in a particular timestep can be identified by not having a predecessor. In the same manner, nodes which were deleted do not have a successor.

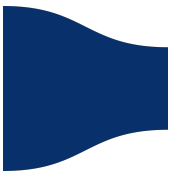
There are different ways of assigning scalar values to nodes. In some data, the value of a node is described as the sum of the values of its children, so that only leaf nodes contain a value. In the following, we will refer to the sum of values of a node's children as *aggregate* and refer to a node which uses this aggregate as its value as *aggregated node*. An example would be population data, where each person has a value of 1 and the population within a state is calculated by the sum of all people living in the state. The population of a country is then inherently defined as the sum of all state populations and so on, until the world population builds the root node, containing the number of living people as an aggregate. In other cases, the value of a node can exceed its aggregate and requires the definition of its own value. Instead of population, we could consider the area of buildings in a city as values. The area of a city district, which would represent the parent node, does not only contain buildings, but additional empty space and requires its own area value. In the same manner, states cover space which is not part of any city, and the world covers area which is not part of individual continents.

C.3.2 Visual Encoding

Nodes and their relationships may undergo many changes over time which need to be visually represented. Nodes can change their value, be added, split, merged, and deleted. They can further build new hierarchical relationships by changing their order among siblings or changing their parent. In the following, we will describe all possible modifications in detail and introduce their visual representation in our method. All higher-level operations on trees, like node swaps or rotations, can be represented by combinations of these cases.

C.3.2.1 Content Change

When the value of a node changes from one timestep to the next, it is represented by a proportional change in height. The change is visualized by an interpolation of the node's current representation and its position and value at the previous timestep. If values are only defined in leaf nodes of the tree, then the change in value propagates up and updates the values of ancestors in an iterative manner.



If the parent of a node has a higher value than its aggregate, then there is visual space available to move children without changing the sibling order. A node's position defines the distance of this node to the parent's starting point on the Y axis. The positional range by which a node can change while neither affecting hierarchical structure nor sibling order is limited by the position and size of the surrounding siblings.

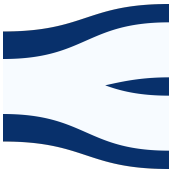
Further changes to the content of a node can occur, e.g. in a document, where the text of a paragraph might change without changing its length. In this case the value and position of the node would stay unaffected and our visual representation would not change. While such changes can, for instance, be covered by changes in color encoding

or tooltip information, they are specific to the underlying data and user task and will therefore not be discussed in this paper.

C.3.2.2 Add & Remove

When a node is added to the data, we cannot interpolate between the current timestep and the node's previous representation as it did not exist in the previous timestep. These cases could be visualized by interpolating between a zero value at the previous timestep and the value of the node at the current timestep. Since the node did not exist in the previous hierarchy, it is not clear where the zero value should be located. This is especially problematic when the hierarchical structure was very different in the previous timestep. We therefore introduce a half ellipse in front of the stream to communicate the addition of nodes to the hierarchical structure, similar to the caps introduced by Chronicler [223]. Analogously, the deletion of a node is represented by adding a half ellipse to the end of the stream.

C.3.2.3 Split & Merge



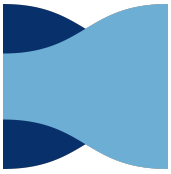
If a node has multiple predecessors, it means that multiple nodes merged into a single node from one timestep to another. If a node splits, it has multiple successors. The split in the data is visually represented by a split of the stream. The stream begins with the representation of the previous node and then splits into the representations of all involved nodes at the current timestep. In the same manner, merges of nodes are represented by a merge of individual streams into a single one. Depending on the locations of merges and splits, stream crossings might be inevitable [121].

C.3.2.4 Move

We will now discuss move operations that always introduce stream crossings to the visualization. We differentiate between three cases: Movement within a node, across nodes, and along ancestors, which can be seen as a special case of the former.

Movement Within a Node: Movement within a node is defined by a reordering of a node's children. This type of movement can only occur when there is a predefined order to the tree nodes. As an example, a tree in one timestep might consist of a root node with two children A and B. In the next timestep, the positioning of the children has switched, so that they are stored in the order (B,A) within the root node. When interpolating between nodes at their respective positions, the child streams will cross each other. The number of edge crossings within the dataset largely depends on the ordering. If the order of children does not matter for the represented data, the number of edge crossings can be reduced by applying a suitable sorting algorithm.

Movement Across Nodes: We define a node changing its parent from one timestep to the next as a hierarchical change across nodes. Since our visualization is built around nested streams, the stream representing such a structural change will at least have to cross the border of its parent element. Depending on the depth difference between the nodes and based on the current order of the tree, the stream might cross several other streams.



Movement Along Ancestors: A special case of nodes moving across hierarchy levels is nodes becoming the parent of one of their previous ancestors. Let us assume a tree that only consists of a root node with a single child. If these two nodes switch their position from one timestep to the other, then the child becomes the parent of its previous parent. Such a scenario could for example occur when switching the inner and outer loop of an algorithm during development. The two structural nodes would change their position and swap the hierarchical level they live in. The problem that occurs lies in the drawing of streams, where streams of higher depth are always drawn on top of the streams they are nested in. Since, in this case, the nesting changes, we would need to draw the child on top of its parent in one timestep, but switch this order for the next timestep. In order to solve this conflict, we are required to cut one of the two streams in half and draw the first half on top and the second half underneath the other stream or vice versa.

To come up with a visually more pleasing solution, we imagine the streams as two sheets lying on top of each other, cut a hole into one of the streams, and thread the other through the hole. The technique scales to an arbitrary number of such movements by adding more holes and threads to the streams. This representation is not only required when a child node switches positions with its parent, but whenever a node becomes an ancestor of any of its previous ancestors. The detection of such cases requires a complete tree traversal.

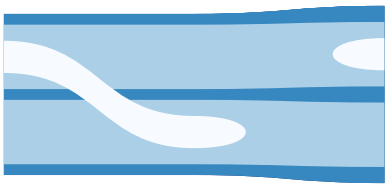
C.4 SplitStream Generation

In the following section we describe our metaphor for visualizing evolving hierarchies. We will demonstrate how the general definition of our method allows for the smooth transition between one-dimensional treemaps and nested stream visualizations, as well as the generation of our own visual approach. We further provide a detailed explanation of parameters to adjust the visualization based on the data and task at hand.

C.4.1 Hierarchy-Change Ratio

When visualizing hierarchies over time, one can display a static tree representation for each timestep in a juxtaposed manner. One example would be to compute a treemap for each timestep and show all these treemaps next to each other sorted by time. In the following, we will only focus on one-dimensional treemaps, which draw each node as a

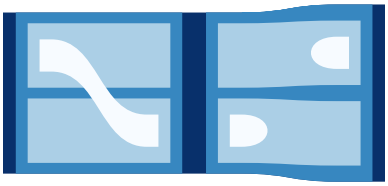
A One-dimensional treemap over time. The white rectangle changes its parent element from step 1 to step 2. Between step 2 and step 3, the white node is deleted and a new white node is created. These hierarchical changes are not visually distinguishable.



B Nested streamgraph. Hierarchical changes are clearly visualized, but the structural nesting is merely encoded in color. Aggregated nodes, such as the dark blue root node, are not visible.

C One-dimensional treemap over time with a fixed width for each node.

D Connecting the nodes of individual treemaps by streams, to represent hierarchical changes.



E SplitStreams. Each treemap is split in the center and all streams are inset based on their hierarchy level. The hierarchical changes over time, the nested structures at each point in time, and aggregated nodes are now clearly visible.

Figure C.2: Treemaps (a) visualize hierarchies at specific timesteps and nested streamgraphs (b) display structural changes over time. SecStreams (e) combine these approaches to accomplish both tasks in one static visualization. All depicted visualizations share the same dataset and a common time axis.

rectangle, stack all children of a node on top of each other and nest them inside their parent element (Figure C.2A). The height of each rectangle corresponds to the value of the node. The hierarchy is represented by assigning a smaller width to deeper nodes. This representation provides a clear depiction of the hierarchy even for aggregated nodes. The main issue when showing static hierarchies per timestep is that changes from one step to another are not readily apparent. The user needs to track the elements' position and color, which becomes more difficult as the number of elements and changes increases. Furthermore, some hierarchical changes cannot be distinguished at all. When a node N changes its parent, its rectangle will be nested inside its parent A in one timestep and then move to be nested inside its new parent B. The very same visualization would be created if N is deleted in A and a new child M is added to B. We can therefore not distinguish between a move operation and the combination of a delete and an add operation (Figure C.2A).

Nested streamgraphs are trying to tackle this problem by visualizing hierarchical changes directly. One can think of this as drawing one-dimensional treemaps at each point in time, setting their width to 0, so that every treemap only consists of a single vertical line, and connecting the representations of each node at every timestep by a curve (Figure C.2B). Because the changes themselves are displayed, we can now easily distinguish the movement of a node from a delete and add operation. Furthermore, the comparison of values is now guided by connected lines and not dependent on the comparison of heights alone. The main problem we face when using streams instead of treemaps is that the hierarchical nesting is not explicitly visualized. The margins that used to display the nesting in treemaps were removed in favor of drawing continuous streams. While color can be utilized to represent hierarchy levels, the number of distinguishable shades of a color limits the levels of hierarchy we can display. Additionally, when a node changes its hierarchy level, it is unclear what color this node should be assigned. On top of that, aggregated nodes are not visible anymore, as shown in Figure C.2B.

We introduce SplitStreams, a hybrid approach which is meant to combine the benefits of both treemaps and nested streams, while reducing the drawbacks of both approaches. The goal is to visualize hierarchical changes over time directly, while still conveying the nesting of nodes at all timesteps. In order to accomplish this task, we draw a treemap at every point in time and set the width of all nodes to a fixed value (Figure C.2C). We then connect nodes between treemaps via streams, which leaves us with a visualization similar to the nested streams, but with horizontal lines in the treemap areas (Figure C.2D). Finally, we split the graph at every timepoint, in the center of each treemap, and move every stream by a certain *margin* away from the split position. The margin that is applied to each node is based on its hierarchical level, which reintroduces the representation of nested structures (Figure C.2E). What we end up with is a visualization of changes, where each block displays the hierarchical structure at two points in time, one in the beginning and one in the end. The part in the middle displays the change between both hierarchies.

In order to draw both, the treemaps at each timestep and the streams between them, we must divide the available space between two points in time and reserve a certain portion of it for each visualization method. If we reserve more space for the treemap, hierarchical changes will be more cramped and less visible. If we dedicate more space to be used by the stream representation, there will be less space for introducing margins

and displaying nested structures at a given point in time. We call this trade-off the *hierarchy-change ratio* (HCR). An HCR of 1 means that we are only showing hierarchy, a treemap at each point in time. An HCR of 0 will only represent change, leading to a nested streamgraph. This parameter allows for the smooth transition between one-dimensional treemaps and nested streamgraphs.

C.4.2 Splits and X-Margins

Treemaps utilize two spatial dimensions to visualize hierarchical nesting, so that rectangles of nodes are completely contained by the rectangle of their parent node. Nested streamgraphs replace one of these spatial dimensions to represent time, which helps to visualize changes, but reduces the user's capability to read hierarchical structure at a given timestep. Nodes of higher hierarchy levels are occluded by the continuous representation of their children. In order to visualize hierarchical structures, we introduce splits to cut the stream representation. We then define an x-margin, which opens up each split, by clipping the width of individual streams. When the width of a stream is reduced, its underlying streams become visible. By choosing a margin for each stream based on its hierarchy level (depth in a tree), we can display the complete hierarchical structure present at that point in time. One should keep in mind, that if the margin becomes too large and the space provided for each point in time is too small, nodes with high depth values will not be visible anymore. We therefore propose to provide screen space for each timestep based on the following formula:

$$HCR \cdot dist(t_i, t_{i+1}) > m_x\big(d_{max}(t_i)\big) + m_x\big(d_{max}(t_{i+1})\big) \tag{C.1}$$

where $dist$ is the distance between two points in time on the x-axis, d_{max} is the maximum depth of the hierarchy at time t and m_x is the margin defined for the depth given. We hereby assure that the distance between two timepoints is large enough to represent every single node given the desired margin. By adding the HCR to the equation, we further ensure that the margin will only be applied in the treemap areas, where streams are drawn as horizontal lines, so that streams will not change their starting and end position when being clipped. In order to keep the time axis linear, the distance between all timesteps should be fixed and larger than the maximum distance required to display the margins without reducing any node's width to zero.

For most of our examples, we utilize a fixed x-margin, which linearly decreases the width of nodes along the depth of the hierarchy by a fixed amount (Equation C.2). Depending on the task the visualization is supposed to solve, a non-linear scaling of margins might be better suited. The margins can for instance be steered to focus on a certain level of hierarchy by applying a bigger margin to nodes of a specific depth. By reducing a stream's width the deeper it is in the hierarchy (Equation C.3), we can emphasize hierarchical structures for deeper nodes (Figure C.3 left). When choosing a higher width reduction for nodes closer to the root (Equation C.4), leaf nodes are barely separated from their parents, keeping the appearance of streams as continuous as possible (Figure C.3 right). In order to avoid introducing overlaps between streams, we define margins in a recursive manner, so that every node's width is at least as much reduced as the width of its parent element. The margins can be represented as functions:

$$m_{fixed} = m_x\big(p(n)\big) + value \tag{C.2}$$

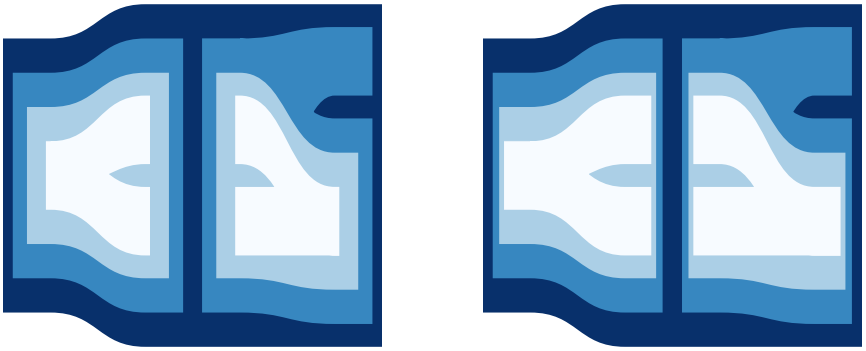


Figure C.3: Left: Hierarchical Margin. Streams receive a higher margin, the deeper their hierarchy level is. Right: Reversed Hierarchical Margin. Streams closer to the root receive larger margins. Although both cases feature the same margin on the root node, the latter case provides a clearer representation of changes over time, because leaf nodes are not pushed so far away from each other.

$$m_{hier} = m_x\big(p(n)\big) + d(n) \cdot value \tag{C.3}$$

$$m_{hier^{-1}} = m_x\big(p(n)\big) + \frac{value}{d(n)} \tag{C.4}$$

where m_x is the margin, n is a node in the tree at timepoint t, p is its parent, d is the node's depth, and *value* is a fixed number which steers the size of the margin. For the root, which does not have a parent node, we set the margin to 0. It should be noted that a single stream can have different margins at different points in time, for instance if it moves into a new parent and thus changes its hierarchy level.

While both treemaps and SplitStreams utilize splits and margins to visualize nested structures, there are two notable differences in their generation. First, to keep the appearance of a continuous stream in SplitStreams, we do not introduce a margin for the root node. Second, as can be seen in Figure C.2, splits for treemaps and SplitStreams are located at different positions on the time axis. For treemaps, we split streams between timesteps to separate them from each other. In our method, we split streams at timesteps to separate the changes. Although splits could be located at any possible position on the time axis, we did not encounter any meaningful examples for other split positions.

C

C.4.3 Y-Padding and Y-Margin

In the same manner as an x-margin can be applied to streams to better communicate the hierarchical nesting of nodes, a y-spacing can help to ensure that a node's representation is completely covered by its parent, hence improving the perception of the hierarchy. In particular for aggregated nodes, hierarchies are only visualized in the x-dimension where split. Y-spacing must be applied with care, as it distorts the represented values. In the following we will discuss two different approaches in detail and demonstrate their advantages and drawbacks.

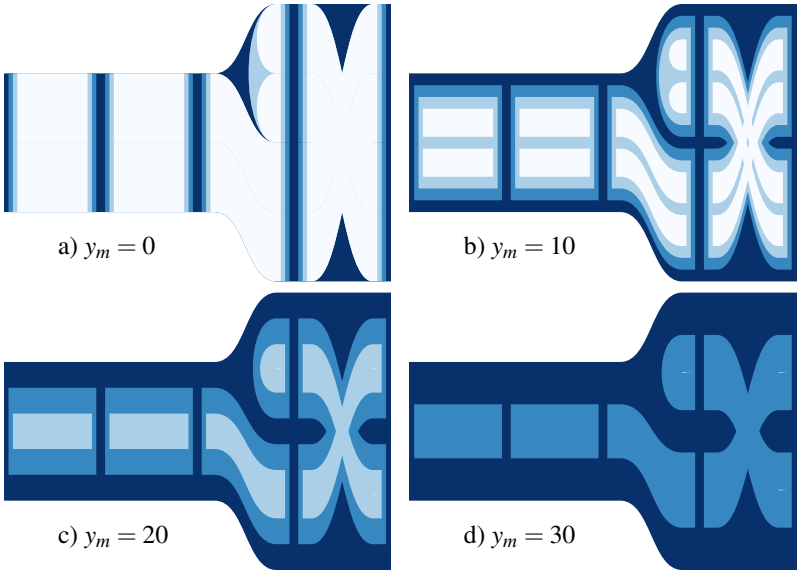


Figure C.4: Y-Margin. For aggregated nodes (a), a y-margin can improve the perception of hierarchies (b). Increasing the y-margin (c,d) creates a form of semantic zoom, since smaller nodes disappear, as the y-margin gets larger.

Adding a y-padding to the visualization means increasing the value of every node which is not a leaf node. Let us consider a simple example, where the tree at a given point in time consists of a root node with one child. The child node has a value of 1, so the root node's aggregate is 1. If we draw the streams for that tree, the root will be completely occluded by its child. We could therefore increase the root's value to make it visible. While hierarchies would be better perceived that way, it also introduces an error because the root node is now represented larger than before. The deeper our hierarchy is, the more padding needs to be introduced. In many cases we want to see additional spacing between siblings, so the padding scales with both depth and the number of children. The most critical part is, that if a node changes its hierarchy level from one timestep to another but keeps the same value, that this value will be represented with different heights, because the applied y-padding depends on the depth of the node. This can have a major impact on the interpretation of values in the visualization. The benefit in perceiving hierarchical structures is in this case counterbalanced by reduced accuracy in value perception. However, for tasks in which the node values are not present or are not of primary importance, but the hierarchical nesting and structural changes matter, y-padding can be utilized as showcased in Figure C.6, to increase the perception of hierarchies. We gave every leaf node a value of 1 and added a y-padding of 1 plus the number of children to every aggregate.

Instead of increasing a node's parent with y-padding, we can reduce a node's height directly by a fixed value. We call that value *y-margin*. The error being introduced is very similar to that of y-padding, with the difference that the node's height will reach 0 as soon as the y-margin is larger than the node's value. This approach can be utilized

as a form of semantic zoom, by increasing the y-margin step by step. As a result, nodes with small values will disappear and nodes of higher hierarchy levels become visible. This approach allows for data inspections independent of the hierarchical level, but only dependent on the node's values. We demonstrate the semantic zoom with different levels of y-margin in Figure C.4. In the same way the x-margin can be defined by different functions (Equation C.2-4), the y-margin can be defined to allow for task-dependent steering of the zoom functionality.

C.4.4 Algorithm

In order to generate a SplitStream, we need to traverse the given hierarchy at every point in time and calculate a position for every node of the tree. The nodes are then individually traversed through time and drawn based on their characteristics. We need to handle cases of nodes being created, being deleted, splitting, and merging, in addition to the connection of nodes by streams. When all streams are drawn, we can introduce splits to cut them open based on the defined margin function. In the following, we will describe our method in more detail.

If values are only defined in the leaf nodes of the data, or the hierarchy has no values defined at all, we need to compute missing values. We therefore recursively traverse every tree in a depth-first approach, until reaching a leaf node. If the leaf has no value defined, we set it to 1. We can then define every node's size by computing its aggregate. To make the hierarchies more visible in the final visualization, we can add a y-padding here.

In case the data does not specify positions of nodes, we can define them in such a way that all children are equally spread within their parent element. We initialize the position of the root element with 0 and iterate through all nodes of the tree except for leaves to define their *spacing* attribute as follows:

$$spacing_n \leftarrow (n.size \text{ - } n.aggregate)/(\#children+1). \tag{C.5}$$

It describes the space between all children of the given node n in the resulting stream visualization. We can then compute the position of every child by:

$aggregate \leftarrow 0$
for all i:=1 **to** #children **do**
  $child[i].pos \leftarrow i \times spacing + aggregate$
  $aggregate \leftarrow aggregate + child[i].size$

When the position and size of every node are set, we can connect them to streams. We can identify all streams by looking for nodes which do not have a predecessor. Starting with these nodes, we follow all their succeeding nodes, draw a stream between the two nodes and repeat this operation in a recursive manner for all following nodes. The algorithm to draw all streams can be found in section C.9A. In order to handle the special case in which a node moves into one of its ancestors, we need to check for every node which changes its parent, if that parent was an ancestor of this node in the previous timestep. If this is the case, we split the stream into two streams and mark them to draw their case-specific encoding at the beginning or the end.

Splits can be integrated into the drawing algorithm directly, or be applied as a post-processing step. The fact that splits can occur at any point in time and can have an arbitrary size, makes the analytical integration somewhat cumbersome. Instead, we apply splits after the drawing process by going through every stream and cutting it at positions where splits occur. For every split we can find the two nodes of the stream which are closest to the split (left and right) and remove a part of the stream equivalent to the respective margins of these nodes. For SplitStreams, we apply splits at every timestep, so that the nodes to the left and to the right of the split both refer to the same node from the timestep the split is applied to. This ensures that the hierarchy is represented in the same way for changes occurring before and after the timestep.

C.4.5 Implementation

We implemented SplitStreams as a JavaScript library based on D3 [34]. To showcase our results, we utilized standard web technologies (HTML5, CSS3) and Vue.js [15]. Streams are created as SVG paths following the outline of all nodes a stream contains. Hierarchical changes are displayed by Cubic Bezier curves with control points set to the center between both points in time, but horizontally to the nodes' y-position. This assures G^1 continuity at the common points. We utilize SVG clippaths to apply x-margins by removing parts of the paths after they have been drawn.

In order to evaluate the performance of our current implementation, we ran the algorithm for generating SplitStreams (section C.9A) for a subset (1313 evolving hierarchies) of Vernier et al. [211]'s benchmark datasets ranging from small datasets up to approximately 100K nodes. The results are plotted in Figure C.5 (right) and the table on the left shows exact timings for selected representative datasets. The timings do not include rendering of the created SVG image as performance varies widely across browsers and devices. In the worst cases that we observed, for very complex topologies, the SVG rendering time was approximately the same as the generation itself. The complete implementation of our method, including many of the presented examples, can be found at *https://github.com/cadanox/SplitStreams*.

# nodes	time [ms]
14	2
347	13
2471	50
12214	247
47371	956
99548	2404

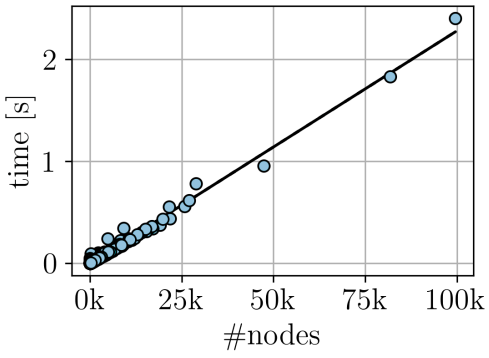


Figure C.5: Algorithm performance for 1313 datasets. The benchmark was executed in Google Chrome v77 on a Windows 10 machine with an Intel Core i7-6700 CPU.

C.5 Use Cases

In order to showcase some of the results that can be achieved by applying our visual metaphor to real data, we selected a taxonomy of medical terms, as well as a public Github repository to demonstrate their evolution over time.

C.5.1 MeSH Taxonomy

Taxonomies are used in science to classify entities, like the phylogenetic tree which shows evolutionary relationships between species. The U.S. National Library of Medicine maintains a taxonomy of Medical Subject Headings (MeSH), currently including nearly 60000 medical terms in a hierarchically-organized vocabulary [10]. The taxonomy requires changes based on scientific progress and new insights gained, and versions are made available for the last ∼20 years. The visualization of changes over the years can help in finding inconsistencies and provide pointers to changes that should be made for the next revision. We selected two branches that show several interesting features of the hierarchy and visualized their evolution. Changes are specified between every two successive years. When there were no changes in the data for several consecutive years, we contracted them into one block.

In Figure C.6, we show the changes in medical terms for the Urinary Tract between 2005 and 2012 with the main branches Urethra (a), Ureter (b), Kidney (c), and Bladder (d). These classes are more clearly separated by SplitStreams than by a nested streamgraph. The two highlighted branches (orange) represent the Kidney Glomerus, once nested inside Nephrons and once inside the Kidney Cortex. In 2006, several nodes were added and the same structural changes were applied to both branches throughout the years. In 2012, the new class Glomerular Filtration Barrier (e) is introduced to the Kidney Glomerus in both Kidney Cortex and Nephrons. However, the two classes Glomerular Basement Membrane (f) and Podocytes (g) are only defined as children of the barrier in Kidney Cortex. This difference is likely to be a mistake in the data and should be fixed in the next revision of the taxonomy. The addition of splits helps to highlight the occurrence of hierarchical changes and to identify the depth of involved nodes.

The evolution of medical terms for the Digestive Physiology between 2006 and 2017 can be seen in Figure C.7. In 2007, the two classes Processes and Phenomena are introduced on the root level (a). The hierarchy stayed consistent for a year, until Phenomena were removed again (b) and their children moved into the root node (c). In 2017, the Processes class is removed as well and all its children move to the root node while receiving new IDs (d). We can see that a few nodes were removed and added, instead of appearing as a continuous stream. This finding can indicate missing ID changes in the data, or an error in the data processing pipeline.

While most changes can be detected in the nested streamgraph representation, it becomes increasingly harder to understand the hierarchical structure and the depth of changing nodes. SplitStreams help us to map the continuous streams to their discrete timescale, highlight positions of hierarchical changes, identify the depth of individual nodes, and provide us with a structured representation of the underlying hierarchy. We can get a more general intuition of the depth layers in which changes occur and thereby judge their impact on the data.

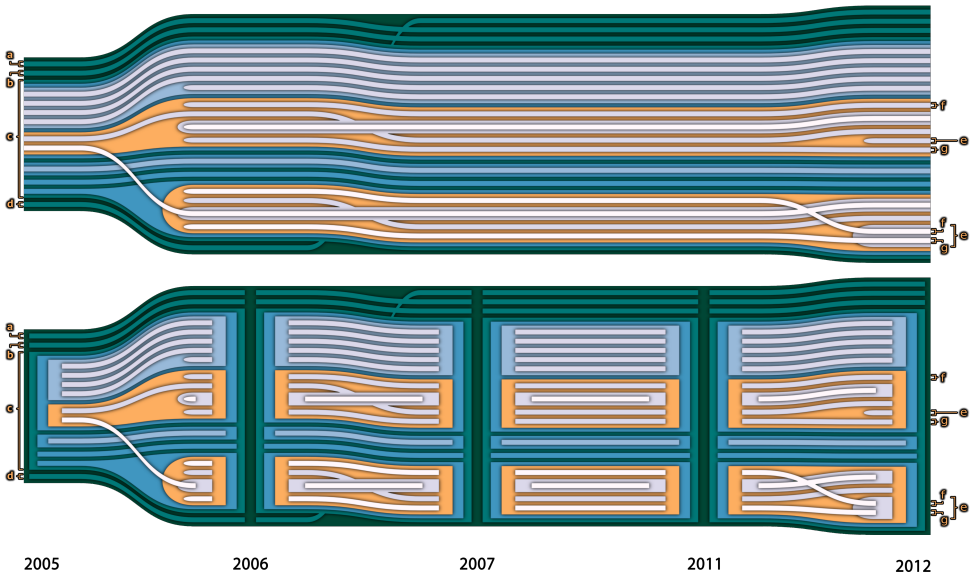


Figure C.6: MeSH Urinary Tract 2005-2012. Hierarchical changes of nodes being added, switching their sibling order, and moving into a new parent, are visible in both representations. SplitStreams enable the user to, e.g., count the number of timesteps shown (5) and count the number of main classes in the Urinary Tract (4). The additional space used for displaying lower level nodes eases the interaction task.

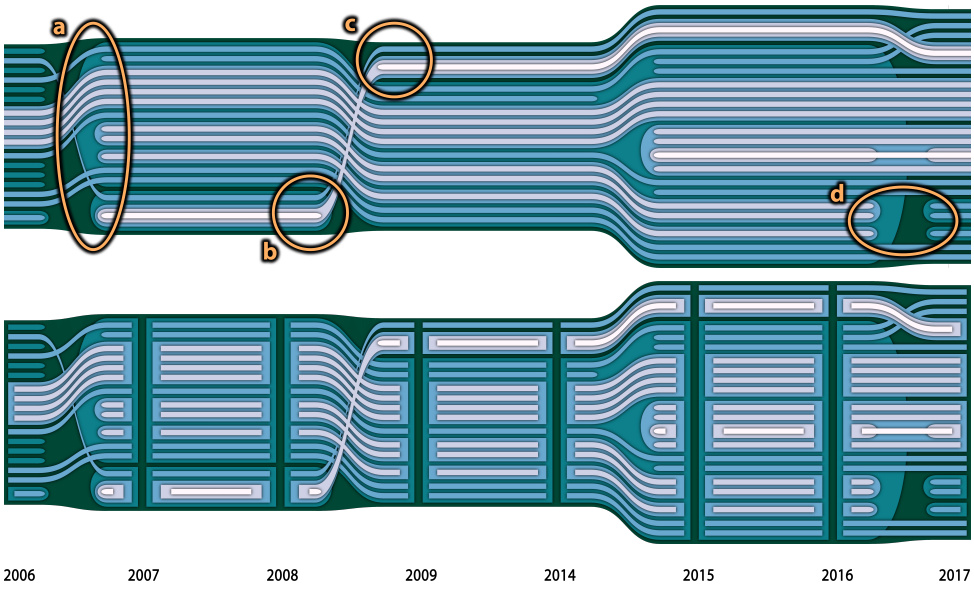


Figure C.7: MeSH Digestive Physiology 2006-2017. a) Addition of Processes and Phenomena to the root node. b) Removal of the Phenomena class. c) Individual Phenomena move into the root node. d) Some changes of class IDs are not listed in the data.

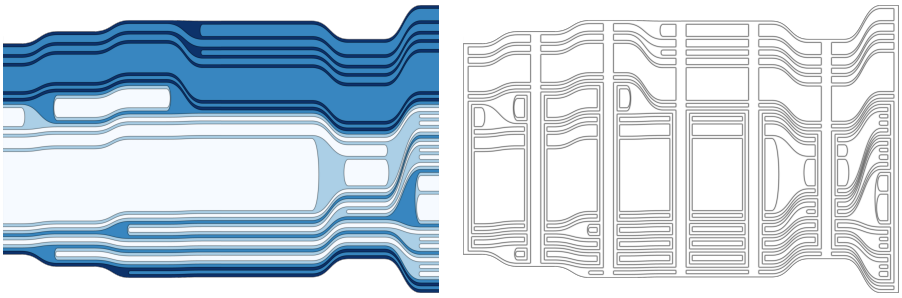


Figure C.8: Dear ImGui Github Repository (7 monthly revisions). While nested streamgraphs (top) utilize color to convey the hierarchical structure of the data, SplitStreams represent it in the form of clear shapes. The hierarchy is even visible, when only drawing the outline of each node.

C.5.2 Leaflet Github

Vernier et al. [211] collected 2720 datasets showing the evolution of values over time, roughly half of which feature a hierarchy. While the authors utilized the data to benchmark different treemap layouts with regards to their stability, we can visualize the very same data in our static, time-dependent hierarchy representation. Dear ImGui [4] is a graphical user interface library for C++. In Figure C.8 we show the evolution of 7 monthly revisions of the ImGui Github repository, once as nested streamgraph and once as SplitStream. The folder structure builds the hierarchy at each point in time and each file's value is determined by the number of lines of code the file contains. While the changes of values and hierarchical changes (addition, deletion of files) are visible in the nested streamgraph, it can be difficult to understand the hierarchical structure given in each point in time. While nested streamgraphs utilize color to represent a node's depth, SplitStreams make use of clear shapes and show depth via containment. A better understanding of the hierarchy eases the task of correlating changes in the hierarchy to files in the repository. Being able to count the number of elements in each node (e.g., 5 in the first revision) gives us a better impression of the distribution of files and the size of higher-level folders. Being able to see a clear separation at every timestep further provides us with a better intuition for the time scale being used. All benchmark datasets from Vernier et al. [211] can be investigated through the exemplary implementation of our method in the supplemental material.

C.6 Evaluation

We conducted a controlled user study to investigate and compare our visualization technique with existing methods. In this section, we briefly explain the experimental design and results.

C.6.1 Motivation

Since SplitStreams are an extension of the nested streamgraph (NSG) approach [137, 223], we want to see if the introduced design changes improve the users' performance

in understanding the hierarchical structure at specific points in time. Treemaps are a well established technique for visualizing hierarchies, so that we can use them as a baseline for how well hierarchies can be perceived in a visualization. Temporal Treemaps [121] utilize cushions to emphasize hierarchical structure in NSG and can therefore be seen as the logical competitor to our approach. Since cushions can be applied to all our tested visualizations, and the introduction of yet another visual encoding would increase the number of independent variables and thereby decrease the statistical power of the analysis and results, we do not consider them for this comparison.

C.6.2 Hypotheses and Goals

Based on the main tasks this visualization is meant to solve, we considered the performance in understanding hierarchical structure at a given point in time, understanding changes in hierarchy, and comparing node values over time.

Given that our method uses the same shape-based design as *Treemaps* to visualize node containment, we hypothesized that users would have a better performance using *SplitStreams* compared to *NSG* in understanding hierarchies at a given point in time. Since hierarchical changes over time are equally represented in *NSG* and *SplitStreams*, we expected users' performance to be significantly better than in the *Treemaps* visualization, where changes are not explicitly displayed. Finally, we tested the hypothesis that both *SplitStreams* and *NSG* would be superior to the *Treemaps* design for the task of comparing node values, because the streams help to identify the areas of interest.

C.6.3 Experiment Design and Task

To test our hypotheses, we designed an experiment in which the participants would view images of visualization techniques and answer one question per image. We defined one independent variable for this experiment, *visualization type*, with three levels: *SplitStreams*, *NSG*, and *Treemaps*. In order to not overwhelm the participants and avoid learning effects, the study followed a between-subject design, where each participant would complete exactly one condition. Each participant completed 14 basic analysis tasks each using a unique visualization image. To avoid potential confounds, all conditions had the same tasks, featuring the same datasets, but showing different images based on the condition created by the system described in subsection C.4.5.

The questions were based on a file system scenario with folders and sub-folders and designed to cover three different analysis tasks. The first task type focused on understanding the hierarchy at a certain point in time (e.g., count the number of siblings or ancestors a certain node has at a certain timestep). The second task type focused on understanding changes in hierarchies over time (e.g., count the number of times a node moved). The last group focused on node-value comparisons over time (e.g., how many nodes shrank during a given period of time). We had 13 questions with a distribution of 5-5-3 for the different task types, and one simple attention check question that was excluded from the analysis. The order of trials was randomized to avoid potential confounds. To reduce task complexity, trials included no more than 20 streams and 5 timesteps in one image. Additionally, the number of hierarchical changes was kept low to avoid frustration particularly in the *Treemaps* condition. All questions for all conditions can be found in the supplemental material.

Participants provided answers via numeric text entry and could make multiple attempts. Each trial participants were required to enter the correct answer or reach a 5-minute time limit before continuing to the next question. Correctly guessing the correct answer without inspecting the visualization was highly unlikely due to the unbounded nature of the textbox input.

Metrics used to assess user performance included: *error of first response*, *time for first response*, *number of attempts*, and *total time per task*, recorded for each trial. Error was calculated as the absolute difference between the answer given in the first attempt and the correct answer. To record and report these measures in accordance to our study goals, we calculated the average of each metric based on the question type.

C.6.4 Participants and Procedure

The evaluation was conducted as an online user study using Amazon Mechanical Turk (AMT). A total of 120 participants completed the study, of which 102 passed the attention check and were included for the analysis. We ended up having 34, 35, and 33 participants per *Treemaps*, *NSG*, and *SplitStreams* conditions. Participants first reviewed a set of instructions based on their assigned condition, followed by 3 example trials and 3 practice trials. After each example and practice trial, they were shown feedback with the correct answer and a visual explanation for how the answer was achieved. Next, participants completed the main set of questions to provide the results for analysis. In order to avoid any learning effects, participants were not given feedback with the correct answers after answering during the main tasks.

C.6.5 Results

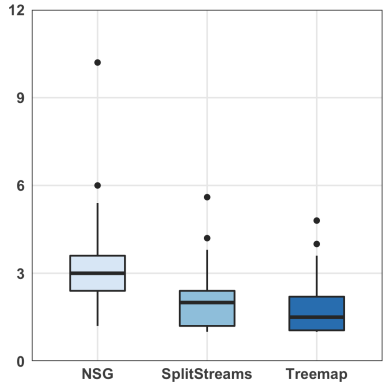
We compared the results from *SplitStreams*, *NSG*, and *Treemaps* conditions to understand their differences based on task types. Due to the data not being normally distributed, we used the Kruskal-Wallis non-parametric test to evaluate the main effect differences among the study conditions and a Wilcoxon post-hoc test for pairwise comparison. The study results are shown in Table C.1. Due to space restrictions, we do not report the test results from *node-value comparison* tasks in this table as we only observed one significant effect among all four measures.

Figure C.9A and Figure C.9B show the average *number of attempts* and *total time per task* for the task type *understanding hierarchies*. The results show that the participants from both *SplitStreams* and *Treemaps* conditions had a significantly lower *number of attempts* compared to the *NSG* condition. Also, participants were significantly faster in answering this type of question with *SplitStreams* than with the *NSG* visualization. These two observations align with our first hypothesis that our approach improves user performance compared to *NSG* by using the same shape-based design as *Treemaps*.

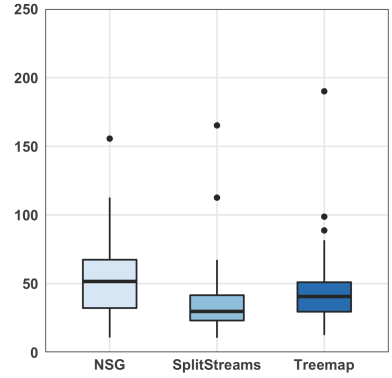
For the task focusing on *understanding the changes of hierarchies over time*, our results demonstrate that *error of first response* and *number of attempts* were significantly higher in the *NSG* condition than in *SplitStreams* and *Treemaps* (Figure C.9C). However, users of *Treemaps* were significantly slower in answering such questions than in the other two conditions (Figure C.9D). Combining these findings, we can conclude that users from both *SplitStreams* and *Treemaps* conditions were not significantly dif-

Measure		Understanding hierarchies	Understanding changes of hierarchies over time
Average # of Attempts	Main Effect	$\chi^2(2) = 24.39, (p < 0.001)$ *	$\chi^2(2) = 14.65, (p < 0.001)$ *
	Posthoc Test	***SplitStreams*** vs. *NSG* $(p < 0.001)$ *	***SplitStreams*** vs. *NSG* $(p < 0.01)$ *
		Treemaps vs. *NSG* $(p < 0.001)$ *	***Treemaps*** vs. *NSG* $(p < 0.01)$ *
Total Time per Task	Main Effect	$\chi^2(2) = 7.80, (p < 0.05)$ *	$\chi^2(2) = 10.58, (p < 0.01)$ *
	Posthoc Test	***SplitStreams*** vs. *NSG* $(p < 0.05)$ *	***SplitStreams*** vs. *Treemaps* $(p < 0.01)$ *
			NSG vs. *Treemaps* $(p < 0.05)$ *
Error of First Attempt	Main Effect	$\chi^2(2) = 26.80, (p < 0.001)$ *	$\chi^2(2) = 14.60, (p < 0.001)$ *
	Posthoc Test	***SplitStreams*** vs. *NSG* $(p < 0.001)$ *	***SplitStreams*** vs. *NSG* $(p < 0.01)$ *
		Treemaps vs. *NSG* $(p < 0.001)$ *	***Treemaps*** vs. *NSG* $(p < 0.01)$ *
Time for First Attempt	Main Effect	$\chi^2(2) = 4.75, (p = 0.09)$	$\chi^2(2) = 15.32, (p < 0.001)$ *
	Posthoc Test		***NSG*** vs. *Treemaps* $(p < 0.001)$ *
			SplitStreams vs. *Treemaps* $(p < 0.05)$ *

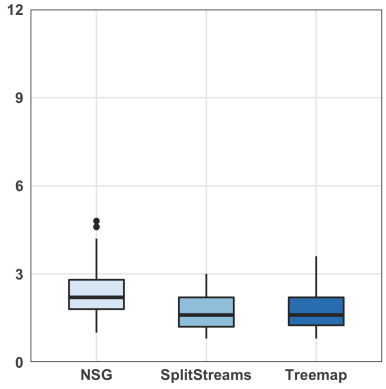
Table C.1: The study results for the task types of *understanding hierarchy* and *understanding changes of hierarchies over time* using a non-parametric Kruskal-Wallis test and a Wilcoxon post-hoc test. We only show the pairwise comparison in the posthoc test if the two conditions had a significant difference. Significant p-values are marked with an asterisk, and condition names with superior performance are shown in bold. Note that the results from the *node-value comparison* task are **not included** in this table for simplicity.



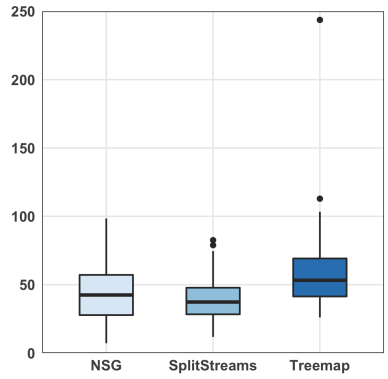
A Average Number of Attempts in *Understanding Hierarchies*



B Total Time (s) per task in *Understanding Hierarchies*



C Average Number of Attempts in *Hierarchy over Time*



D Total Time (s) per task in *Hierarchy over Time*

Figure C.9: Distribution of the results for the average number of attempts and the total time per task in seconds.

ferent in getting the correct answers, but people who used our technique could reach the conclusion faster.

Another interesting finding shows that users of *NSG* spent significantly less time on their first attempt compared to users of *Treemaps*, while also having a significantly higher *error of first response*. This could mean that users of *NSG* were more confident, hence providing their answer faster, but were wrong more often. One possible interpretation of this result could be that stream-based techniques put less cognitive load on users than *Treemaps*.

Our evaluation of node-value comparison tasks showed a significant difference for *error of first response* ($\chi^2(2) = 9.10$, $p < 0.05$). The post-hoc test showed that users had significantly lower errors with *SplitStreams* than with *NSG* ($p < 0.01$). This suggests that our approach helps users understand changes of values over time better than with *NSG* at the first glance. We do not have enough evidence to support or reject our hypothesis that *Treemaps* users would perform worse than stream-based approaches in this task.

Overall, the evaluation was unable to detect major drawbacks in *SplitStreams* as compared to the tested alternatives. Our findings provide evidence to support our hypotheses that by using *SplitStreams*, we can utilize the benefits of both *NSG* and *Treemaps* while avoiding some of their individual shortcomings.

C.7 Discussion

The technique presented in this paper can be used to generate one-dimensional treemaps over time, nested streamgraphs, as well as new representations introduced by x- and y-margins.

The results of our study demonstrate that *SplitStreams* provide a similar performance to *Treemaps* in tasks involving understanding hierarchical structures in an isolated time period. This shows the approach can convey a similar level of detail in terms of hierarchical structure while also providing additional information on changes in the data. With both stream-based approaches providing statistically-significantly faster user responses and better performance than *Treemaps* in tasks for understanding changes over time, *SplitStreams* represents a general-purpose technique with good performance in the tested use cases.

When it comes to scalability, the number of nodes and timesteps that can be visualized by SplitStreams without introducing clutter are in the same order of magnitude as in treemaps and nested streamgraphs. Based on our experience and using current screen resolutions, a static hierarchy might display up to several hundred distinguishable nodes, but this capacity is highly diminished with the introduction of hierarchical changes. While value changes, additions, and deletions of nodes can occur in higher quantities, the visual overlap introduced by node movements can greatly affect the user's capabilities of reading the visualization. Although we try to handle visual clutter through the introduction of a y-margin as a form of a semantic zoom, the exploration of large datasets (e.g., the visualization of the whole MeSH taxonomy in section C.5) requires additional mechanisms to ensure scalability. While SplitStreams and treemaps allow for a better hierarchical perception in deep hierarchies, their introduced margins require more space along the time axis than nested streamgraphs. To be precise, a deeper hierarchy and larger margin definition requires more distance between individual timesteps (Equation C.1). On the other hand, the margin space can not only be exploited to enhance and highlight the hierarchical structure with, e.g., halos or drop shadows, but also eases the selection of a node of interest in an interactive environment. The discontinuity along the temporal axis introduced by *SplitStreams* had no detectable negative impact on user performance in our study, but might affect the aesthetic appeal or visual complexity of the representation.

While we currently apply splits to all streams at every single point in time, we could apply margins to a subset of streams and timepoints. Such an approach could, for instance, emphasize hierarchical changes by only applying margins to streams for which a hierarchical change happens. Conversely, since nested streams do a good job at representing hierarchical changes, it would be possible to selectively emphasize the hierarchy when no changes occur. We believe that such a selective application of our technique could allow for an enhanced depiction of features of interest.

We introduced three different functions to apply x-margins at points of splits. While

these functions can be defined in many ways and adjusted based on specific user tasks, their advantages and disadvantages deserve further study. The same is true for y-spacing in particular because it affects node value perception.

Existing work has shown that the order of streams can be optimized to reduce the number of stream crossings [121]. However, the proposed strategy is limited to data where hierarchical changes occur along siblings. Our technique would benefit from an adapted ordering algorithm to consider hierarchical changes of all types.

C.8 Conclusion

We presented a novel visual metaphor for the visualization of hierarchically structured data over time. Our approach allows for the clear representation of hierarchies at certain points in time, while simultaneously conveying the temporal evolution of data values and changes in the hierarchy. Compared to existing techniques, all possible hierarchical changes are supported and represented. The evaluation confirms that our approach provides equivalent performance to treemaps and nested streamgraphs in the analyzed tasks they perform best in and therefore makes a good general-purpose technique. We provide a JavaScript library for the easy reproduction of all demonstrated examples and for integration into other projects at *https://github.com/cadanox/SplitStreams*.

C.9 Acknowledgements

The research presented in this paper was supported by the MetaVis project (#250133) funded by the Research Council of Norway. The work is also supported in part by NSF 1565725 and the DARPA XAI program N66001-17-2-4032.

Appendix

In the following we describe Algorithm 1 from section C.4.4 in pseudocode.

Algorithm 1 Streamgraph Generation

```
1:  procedure TRAVERSESTREAM(node)
2:      if All previous nodes of node have been visited then
3:          mark node as visited
4:      if Node has next node then
5:          for all next do
6:              if not next was visited then
7:                  t1 ← t(node) + 0.5 × HCR × (t(next) − t(node))
8:                  t2 ← t(next) − 0.5 × HCR × (t(next) − t(node))
9:                  draw straight stream from t(node) to t1
10:                 draw curved stream from t1 to t2
11:                 draw straight stream from t2 to t(next)
12:                 TRAVERSESTREAM(next)
13:     else
14:         tEnd ← t(node) + 0.5 × HCR
15:         draw end cap from t(node) to tEnd
16:
17: procedure DRAWSTREAMS
18:     for all streams do
19:         tStart ← t(firstNode) − 0.5 × HCR
20:         draw start cap from tStart to t(firstNode)
21:         TRAVERSESTREAM(root)
```

C

Paper D

Organic Narrative Charts

Fabian Bolte and Stefan Bruckner

Abstract

S toryline visualizations display the interactions of groups and entities and their development over time. Existing approaches have successfully adopted the general layout from hand-drawn illustrations to automatically create similar depictions. Ward Shelley is the author of several diagrammatic paintings that show the timeline of art-related subjects, such as *Downtown Body*, a history of art scenes. His drawings include many stylistic elements that are not covered by existing storyline visualizations, like links between entities, splits and merges of streams, and tags or labels to describe the individual elements. We present a visualization method that provides a visual mapping for the complex relationships in the data, creates a layout for their display, and adopts a similar styling of elements to imitate the artistic appeal of such illustrations. We compare our results to the original drawings and provide an open-source authoring tool prototype.

D.1 Introduction

Movie Narrative Charts [146] are hand-drawn illustrations that display characters as streams and show their interactions in different locations over time. They have inspired a multitude of works in the visualization community and several attempts have been made to create digital replicas. In the same manner, we were inspired by Ward Shelley's diagrammatic paintings, which cover complex relationships and a multitude of textual annotations to display the evolution of art related subjects. We aim to create a digital chart that captures the organic appearance, displays the complex relationships between individual entities, and imitates the artistic appeal of these drawings. The digital support for such diagrams would not only allow for the fast creation of visually appealing results from varying data sources, but further ease the lengthy planning process that artists undergo to create a single piece of work. The provision of an authoring

This article was published in Eurographics 2020 - Short Papers (2020).

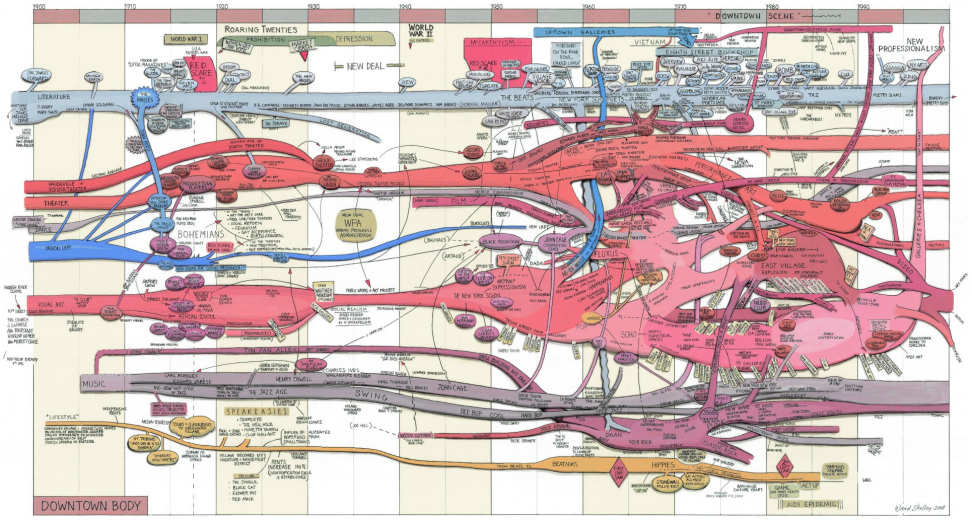


Figure D.1: Downtown Body by Ward Shelley [16]

tool for artists could allow them to focus more on the artistic part, such as the creation of a general theme and bigger picture, that automatic systems will not be able to create.

We contribute a more organic layout for narrative charts that combines nesting, splits, merges, parent changes, and multiple label types. We apply image processing techniques to create a visual appeal similar to hand-drawn paintings by artists, while still complying with the SVG standard for further editing. Our prototype of an interactive authoring tool is made publicly available.

D.2 Related Work

While a lot of research deals with time-related data, layouting techniques, and arts, we put our focus on stream-based visualizations.

Ogawa and Ma [150] automated the creation of narrative charts, whose aesthetics were improved by Tanahashi and Ma [192]. StoryFlow [135] applies an adaption of a layered graph layout [189] to improve the performance of the aesthetics optimization. Finally, Tang et al. [193] analyzed the differences between automatic layout approaches and artist desires and developed iStoryline as an authoring tool for the creation of storyline visualizations. In narrative charts, streams move together in groups, but can not be individually divided or merged. In our case, on the other hand, we are looking at streams with changing height values that feature direct connections, merges, splits, and a predominant amount of text labels.

TextFlow [57] and Xu et al. [230] display topic splits and merges based on the ThemeRiver metaphor [81], facilitate a force layout to generate aesthetically pleasing results, and integrate word clouds into flow charts. Our approach adds an artistic style closer to hand-drawn charts, supports a larger amount of text labels directly integrated into the layout algorithm, and provides interactions for an authoring tool prototype.

D.3 Overview

Ward Shelley's diagrammatic paintings typically describe the evolution of a given topic over time. *Downtown Body* (Figure D.1), for example, shows how different branches of art, like theater, music, and visual arts, changed and interacted throughout the 20th century. His drawings not only provide a general overview of topics and their relevance over the years, but further highlight individual milestones of every time period. By analyzing his work we identified three elements that define the main structure of his paintings. We will in the following refer to them as *streams*, *labels*, and *links*. We further analyzed the artistic *style* of the paintings to take inspiration for creating similar digital results.

Streams: All elements in Shelley's paintings follow a common timeline. A stream is defined for a certain period of time and describes one of the main topics of interest. For example, one stream might represent *Literature* from 1898 until the year 2000, while another stream called *Film* only starts to show in the year 1942. Streams are represented by colored lines whose thickness can change over time to convey the changing relevance of this topic (Figure D.2B A). In addition to the main topics, there might be relevant subtopics that the artist wants to highlight. The *New York School* and *Early Gentrification* in *Visual Arts* can be seen as such an example. They are typically represented by a stream of slightly different shade than the main topic, nested inside the main stream (Figure D.2B C).

Streams and their represented topics can merge and split at specific points in time. When multiple streams merge, they will be represented by a single line after the given timepoint. In the same manner, a splitting stream will be drawn as multiple lines after the given point in time and represent multiple topics.

Labels: One of the major elements that contributes to the vast amount of information these drawings contain is text. Every stream is labeled by its represented topic when it appears. Additional text labels are describing the narrative of the stream by naming major events or core contributions to that field. In the *Literature* topic, such labels include authors like *Emma Goldman*, general movements like the Beat Generation (*The Beats*), newsletters, book stores, and events like *Poetry Slams*. The majority of labels is connected to at least one of the topic streams and referring to a certain point or period in time.

Labels come in many different forms and shapes, but can be classified into three major classes: *outside*, *inside*, and *on top* of streams. An example for each type can be found in Figure D.2B. Outside labels are typically surrounded by another shape that is connected to the referred to stream through a line in the same color. Inside labels have a shape similar to outside labels, but are missing the connecting line. Instead, they touch the stream they refer to, or are drawn inside it. Their background color is different to the color of the stream, but often similar in hue. A label drawn on top of streams is not surrounded by a shape, but directly drawn onto the stream itself. Its text bends along the overall stream shape.

All text labels are black with all letters capitalized. Their font sizes can vary to emphasize on the importance of individual elements. While inside and outside labels can feature arbitrary surrounding shapes, they are mainly surrounded by ellipses and rectangles. By utilizing unique shapes for specific labels they can stand out from the rest.

Links: Links communicate the connection of two entities, joining streams with streams, labels with labels, labels with streams, and even other links to streams or labels. They are defined by a start and an end point, each of which is defined by a point in time and a connected entity. If the entities at the start and end point feature different colors, the link is typically drawn in the color of the start entity. Links can further be drawn as thin black lines or arrows. Compared to streams, links typically do not change in size and rarely contain labels. Furthermore, while streams typically exist throughout a larger time period, links are rather short-lived, which leads to a general difference in appearance. While streams follow the time axis of the diagram, links are typically almost perpendicular to it.
Style: Ward Shelley's paintings feature an iconic time axis where time is discretized into blocks of alternating color. These blocks are drawn on the time axis itself, as well as in the background of the painting, where they can use different colors and a different discretization. Colors on the time axis are generally more saturated.

All shapes are typically filled by a solid color categorizing the topic of the element. Nested streams utilize a different shade of the same color to appear similar. The elements feature a strong black outline, as well as an inner and outer shadow towards the bottom left. The general shape of elements appears organic and follows a common scheme to create a bigger picture like blood vessels in the body (*Downtown Body*), or a tentacled beast (*History of Science Fiction*). This reoccurring schematic style inspired the name for our organic narrative charts. In some of the drawings we found additional complimentary embellishments like pictures of human bodies (*Carolee Schneemann*) or album covers (*Frank Zappa*).

D.4 Method

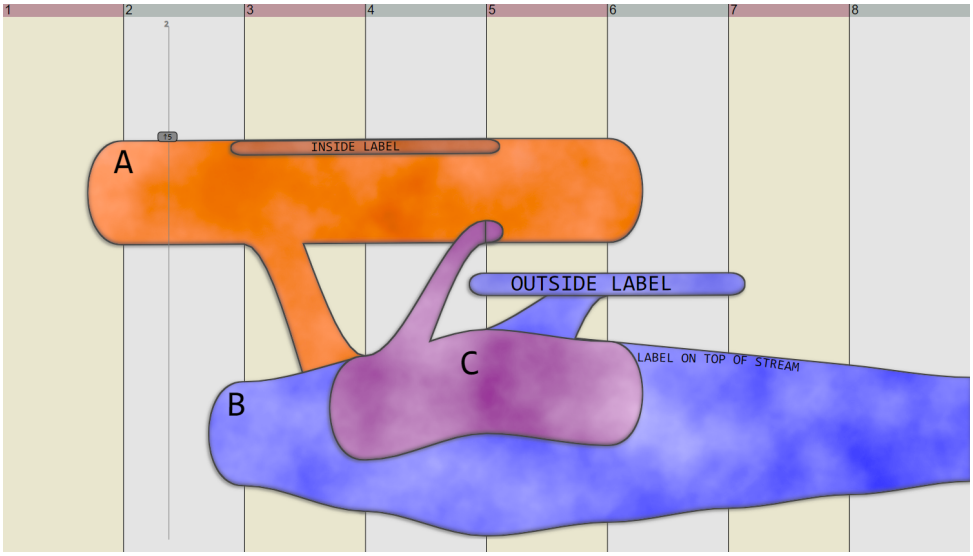
Graph Layout: We transform our data into a directed acyclic graph (DAG), similar to approaches found in TextFlow [57] and StoryFlow [135], to benefit from existing graph drawing algorithms when creating a layout. For each stream, we create one node at each discrete point in time (e.g., one node per year) within the time interval it is defined in and connect consecutive nodes via edges. For each label, we create a node at the referred to point in time, as well as an equal amount of nodes to the left and right, based on the length of the text. This representation of labels by several nodes enables the bending of text along the time axis. For outside labels we add an additional edge to a stream node.

We build on top of the SplitStreams [32] approach to support a nested graph model, where every node can contain multiple other nodes, as well as splits and merges of streams. We utilize the same nesting mechanism to support labels inside and on top of streams. Links are represented by an edge between two nodes, which must refer to different timepoints to satisfy the DAG constraints. If links span over several points in time, we generate intermediate link nodes as described by Sugiyama et al. [189].

We utilize a force layout to generate the organic appearance of streams. Each node is fixed in the time dimension and its position can only vary in the second dimension based on the applied forces. Initially, all nodes are stacked above each other in the same order as defined in the data. A gravitational force applied to all nodes pushes them towards the center of the picture. A repulsive force between all nodes increases the spacing

Streams	**ID**	t_0	t_1	**color**	**size**	**parent**
	A	2	6	#D73		
	B	3	9	blue	5/10	
	C	4	6	purple		B
Links	**from**		t_0	**to**	t_1	**merge**
	A		3	B		true
	C		4	A		
Lables	**stream**	**t**		**text**	**type**	**size(em)**
	A	4		inside label	in	3
	B	6		outside ...	out	5
	B	7		... on top ...	on	3

A CSV data. The specified data format enables a simple definition of all diagram elements. Each stream has an ID and is defined in a given time period. Size is a list of tuples, where the first element refers to a point in time and the second to the size. A parent is only defined for nested streams.



B Visual result of the CSV data given in a. The force layout found a good separation of nodes and the artistic stylization was applied. Hovering over A shows the draggable button to change the node size at the given timepoint.

Figure D.2: We defined 3 streams A, B, C, where C is nested inside B. B has a defined size of 10 at timepoint 5, and its size at other timepoints is calculated by linear interpolation between 10 and the default value (5) at the start and end point. C has a link to A, which results in a small representation of C nested inside A. A has a merge link to B, which means that no such nested node is created. We define 3 labels of different types, resulting in different representations.

between individual streams and labels. We make sure that nodes stay inside the picture by introducing a collision detection at the border. A similar constraint is applied for nested nodes, so that they can not escape their parent elements. For simplification, collisions are not computed for actual stream shapes, but based on the individual nodes in each timestep.

We introduce different forces along different edge types for a fine-grained layout control. Edges between nodes of the same stream feature rather strong forces to keep the streams as straight as possible and to reduce stream crossings. These forces can be loosened to increase the wiggle of lines and thereby change the general stream appearance. Edges between labels and streams make sure that labels are drawn in close proximity to their connected stream. Finally, links between different entities keep merged, split, and connected streams closer together. By modifying their force, we can control how likely a stream is to bend for such a connection.

Artistic Style: We draw Cubic Bezier curves between connected nodes of the graph. Each stream is given a solid color, on top of which we apply image processing techniques to acquire the desired style. We multiply the colored stream with a grey scale fractal noise to integrate irregularities. The colors appear more hand-drawn than the solid color that is free from any imperfections. We add a strong black outline and both an inner and an outer shadow to each element of the chart. The latter are achieved by multiplying each element's pixel image with a black and offset copy of itself.

Interactions: The manual creation of CSV data for the generation of organic narrative charts might be tedious. We therefore designed several interaction techniques that we consider to be crucial for the development of an authoring tool for artists.

By clicking and holding the mouse button (or touch) in an empty area and dragging it over the chart, we can define a start and end point for a new stream. The same drag&drop interaction can be utilized to create a link between two entities. If only one of the two points has an underlying entity and the other lies in an empty area, this interaction will create a new stream and a link connecting both entities. The user can define if the created connection is supposed to merge (or split) the connected stream, or if the new stream will be nested at the connection point.

When hovering over a stream, a draggable button will be shown that allows for the adjustment of the stream size at that point in time. In Figure D.2B we hover over stream A between timesteps 2 and 3. The size of each stream node is calculated through linear interpolation between consecutive size definitions and a default value at the start and end point. Clicking at a stream will create a label at that point in time and a popup window enables the user to enter the desired text and type for the label.

Implementation: Our prototype implementation is based on D3 [34] and standard web technologies (JavaScript ES6, HTML5, CSS3). Additionally to the interactions mentioned, we allow for the direct manipulation of the CSV data for detailed changes and update the chart on every change. We utilize the D3-integrated force layout and draw streams between connected nodes through the SplitStreams [32] library. New nodes are added into the existing graph and the force layout runs based on the previous node positions to minimize calculation times. We utilize SVG filters as image processing techniques to apply the artistic style. The output is in the standard SVG format, which allows for further processing in common vector graphics tools. The result of our technique applied to a manually created data set mirroring the data in the *Downtown Body* painting can be seen in Figure D.3.

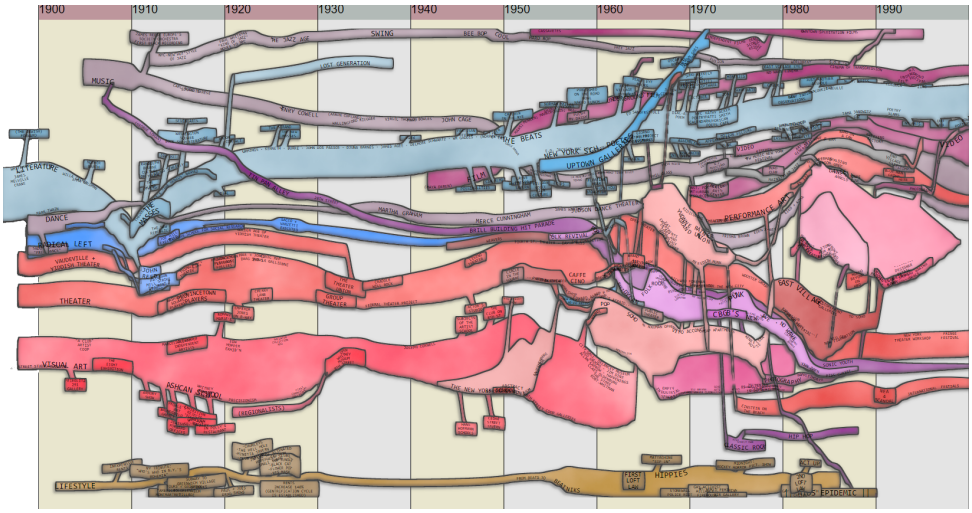


Figure D.3: Digital recreation of Ward Shelley's Downtown Body, including 44 streams, 61 links, and 369 labels.

D.5 Discussion and Limitations

The creation of our prototype yielded results of promising visual quality to improve existing narrative charts and added an artistic appeal that is normally only achieved by hand-drawn diagrams. One of the main problems we face in the creation process is the correct definition of parameters for the force layout to properly separate individual nodes. If the authoring tool is meant to be used by the general public, they should not be exposed to a manual control of these parameters, but instead be given a fully-automated, or at least largely simplified, control sequence. This is especially true for the positioning of labels, which might not be readable when curves bend too strong or overlaps occur.

We currently do not provide means for the integration of images for visual embellishment, and can not bend the time axis as featured in some of Ward Shelley's works. We can not capture the thematic content of the diagram to create a bigger picture like artists can convey. We therefore see our application more as a planning tool for artists to simplify the creation of such charts, as well as an option to easily create visually appealing diagrams from digital data.

D.6 Conclusion

We presented organic narrative charts, a digital recreation of Ward Shelley's diagrammatic paintings from data, that is capable of conveying the complex relationships of entities in his work and provides a similar visual appeal. We implemented a prototype of an authoring tool and compared the digital results to the original work. We provide the full source code at *https://github.com/cadanox/orcha*.

«Are we done?»
Fabian

Bibliography

[1] Ace. https://ace.c9.io/. Accessed: 2018-12-11. B.5

[2] Allen developing mouse brain atlas. https://developingmouse.brain-map.org/. Accessed: 2020-04-17. 4.2

[3] Apache subversion. https://subversion.apache.org/. Accessed: 2018-12-11. B.4.1

[4] Dear imgui. https://github.com/ocornut/imgui. Accessed: 2019-10-14. C.5.2

[5] diff2html. https://diff2html.xyz/. Accessed: 2018-12-11. B.5

[6] Git. https://git-scm.com/. Accessed: 2018-12-11. B.4.1, B.5

[7] GoldenLayout. http://golden-layout.com/. Accessed: 2018-12-11. B.5

[8] Leaflet. https://leafletjs.com/. Accessed: 2020-04-17. 4.2, 4.4

[9] libgit2. https://libgit2.github.com/. Accessed: 2018-12-11. B.5

[10] Mesh. https://www.nlm.nih.gov/mesh/meshhome.html. Accessed: 2019-10-10. C.5.1

[11] Observable. https://observablehq.com/. Accessed: 2019-04-29. B.2

[12] Overleaf. https://www.overleaf.com/. Accessed: 2018-12-11. B.2

[13] ShaderToy. https://www.shadertoy.com/. Accessed: 2018-12-11. 3.2.1, B.1, B.2, B.8

[14] VisGuides. http://visguides.org/. Accessed: 2019-08-27. A.4

[15] Vue.js. https://vuejs.org/. Accessed: 2019-03-31. C.4.5

[16] Ward Shelley. http://www.wardshelley.com/. Accessed: 2019-12-20. 3.3.2, 4.6, D.1

[17] AHRENS, J., GEVECI, B., AND LAW, C. ParaView: An end-user tool for large-data visualization. In *The Visualization Handbook*. Elsevier, 2005, pp. 717–731. 2.2, 2.2, B.2, B.8

[18] ALBERS, D., DEWEY, C., AND GLEICHER, M. Sequence surveyor: Leveraging overview for scalable genomic alignment visualization. *IEEE Transactions on Visualization and Computer Graphics 17*, 12 (2011), 2392–2401. 2.2

[19] ALTINTAS, I., BERKLEY, C., JAEGER, E., JONES, M., LUDASCHER, B., AND MOCK, S. Kepler: An extensible system for design and execution of scientific workflows. In *Proceedings of the International Conference on Scientific and Statistical Database Management* (2004), pp. 423–424. B.2

[20] BALL, T., AND EICK, S. Software visualization in the large. *IEEE Computer 29*, 4 (1996), 33–43. B.2

[21] BANGOR, A., KORTUM, P., AND MILLER, J. Determining what individual SUS scores mean: Adding an adjective rating scale. *Journal of Usability Studies 4*, 3 (2009), 114–123. 4.1, B.7

[22] BANGOR, A., KORTUM, P. T., AND MILLER, J. T. An empirical evaluation of the system usability scale. *International Journal of Human–Computer Interaction 24*, 6 (2008), 574–594. A.4

[23] BARNES, C., AND ZHANG, F.-L. A survey of the state-of-the-art in patch-based synthesis. *Computational Visual Media 3*, 1 (2017), 3–20. 2.3

[24] BATEMAN, S., MANDRYK, R. L., GUTWIN, C., GENEST, A., MCDINE, D., AND BROOKS, C. Useful junk?: The effects of visual embellishment on comprehension and memorability of charts. In *Proceedings of ACM CHI* (2010), p. 2573–2582. 3.1, A.3.4

[25] BAUR, D., LEE, B., AND CARPENDALE, S. TouchWave: Kinetic multi-touch manipulation for hierarchical stacked graphs. In *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces* (2012), pp. 255–264. C.2

[26] BAVOIL, L., CALLAHAN, S. P., CROSSNO, P. J., FREIRE, J., SCHEIDEGGER, C. E., SILVA, C. T., AND VO, H. T. VisTrails: Enabling interactive multiple-view visualizations. In *Proceedings of IEEE VIS* (2005), pp. 135–142. 2.2, 2.4, B.2, B.4.1

[27] BEHRISCH, M., BACH, B., HUND, M., DELZ, M., VON RÜDEN, L., FEKETE, J.-D., AND SCHRECK, T. Magnostics: Image-based search of interesting matrix views for guided network exploration. *IEEE Transactions on Visualization and Computer Graphics 23*, 1 (2016), 31–40. 3.1

[28] BEHRISCH, M., BLUMENSCHEIN, M., KIM, N. W., SHAO, L., EL-ASSADY, M., FUCHS, J., SEEBACHER, D., DIEHL, A., BRANDES, U., PFISTER, H., SCHRECK, T., WEISKOPF, D., AND KEIM, D. A. Quality metrics for information visualization. *Computer Graphics Forum 37*, 3 (2018), 625–662. A.3.1, A.3.2, A.2, A.4

[29] BERTINI, E., AND SANTUCCI, G. Quality metrics for 2D scatterplot graphics: Automatically reducing visual clutter. In *Proceedings of the International Symposium on Smart Graphics* (2004), pp. 77–89. A.3.2

[30] BERTINI, E., AND SANTUCCI, G. Improving 2D scatterplots effectiveness through sampling, displacement, and user perception. In *Proceedings of IVAPP* (2005), pp. 826–834. A.3.2

[31] BERTINI, E., TATU, A., AND KEIM, D. Quality metrics in high-dimensional data visualization: An overview and systematization. *IEEE Transactions on Visualization and Computer Graphics 17*, 12 (2011), 2203–2212. B.2

[32] BOLTE, F., NOURANI, M., RAGAN, E. D., AND BRUCKNER, S. SplitStreams: A visual metaphor for evolving hierarchies. *IEEE Transactions on Visualization and Computer Graphics* (2020). D.4

[33] BORKIN, M. A., VO, A. A., BYLINSKII, Z., ISOLA, P., SUNKAVALLI, S., OLIVA, A., AND PFISTER, H. What makes a visualization memorable? *IEEE Transactions on Visualization and Computer Graphics 19*, 12 (2013), 2306–2315. 1.1, 2.3, A.3.4

[34] BOSTOCK, M., OGIEVETSKY, V., AND HEER, J. D3: Data-driven documents. *IEEE Transactions on Visualization and Computer Graphics 17*, 12 (2011), 2301–2309. B.5, C.4.5, D.4

[35] BREHMER, M., AND MUNZNER, T. A multi-level typology of abstract visualization tasks. *IEEE Transactions on Visualization and Computer Graphics 19*, 12 (2013), 2376–2385. A.1

[36] BRODLIE, K., POON, A., WRIGHT, H., BRANKIN, L., BANECKI, G., AND GAY, A. GRASPARC - A problem solving environment integrating computation and visualization. In *Proceedings of IEEE VIS* (1993), pp. 102–109. 2.1

[37] BROOKE, J. SUS – A quick and dirty usability scale. In *Usability Evaluation in Industry*, vol. 189. CRC Press, 1996, pp. 4–7. 4.1, A.4

[38] BRUCKNER, S., ISENBERG, T., ROPINSKI, T., AND WIEBEL, A. A model of spatial directness in interactive visualization. *IEEE Transactions on Visualization and Computer Graphics 25*, 8 (2019), 2514–2528. 3.1, A.3.3

[39] BRUCKNER, S., AND MÖLLER, T. Result-driven exploration of simulation parameter spaces for visual effects design. *IEEE Transactions on Visualization and Computer Graphics 16*, 6 (2010), 1468–1476. 3.2.2

[40] BURCH, M., BECK, F., AND DIEHL, S. Timeline Trees: Visualizing sequences of transactions in information hierarchies. In *Proceedings of the Working Conference on Advanced Visual Interfaces* (2008), pp. 75–82. C.2

[41] BURCH, M., BLASCHECK, T., LOUKA, C., AND WEISKOPF, D. Visualizing hierarchy changes by dynamic indented plots. In *Proceedings of IVAPP* (2014), pp. 91–98. C.2

[42] BURCH, M., AND WEISKOPF, D. Visualizing dynamic quantitative data in hierarchies. In *Proceedings of IVAPP* (2011), pp. 177–186. C.2

[43] BYRON, L., AND WATTENBERG, M. Stacked graphs – Geometry & aesthetics. *IEEE Transactions on Visualization and Computer Graphics 14*, 6 (2008), 1245–1252. 1, 3.1, A.3.2, C.2

[44] CALLAHAN, S. P., FREIRE, J., SCHEIDEGGER, C. E., SILVA, C. T., AND VO, H. T. Towards provenance-enabling paraview. In *Proceedings of the International Provenance and Annotation Workshop* (2008), pp. 120–127. 2.2

[45] CAMISETTY, A., CHANDURKAR, C., SUN, M., AND KOOP, D. Enhancing web-based analytics applications through provenance. *IEEE Transactions on Visualization and Computer Graphics 25*, 1 (2018), 131–141. B.2

[46] CARD, S. K., SUH, B., PENDLETON, B. A., HEER, J., AND BODNAR, J. W. TimeTree: Exploring time changing hierarchies. In *Proceedings of IEEE VAST* (2006), pp. 3–10. C.2

[47] CHEN, M., AND GOLAN, A. What may visualization processes optimize? *IEEE Transactions on Visualization and Computer Graphics 22*, 12 (2015), 2619–2632. A.3.3

[48] CHEN, M., GRINSTEIN, G., JOHNSON, C. R., KENNEDY, J., AND TORY, M. Pathways for theoretical advances in visualization. *IEEE Computer Graphics and Applications 37*, 4 (2017), 103–112. 1, A.4

[49] CHEN, M., AND JAENICKE, H. An information-theoretic framework for visualization. *IEEE Transactions on Visualization and Computer Graphics 16*, 6 (2010), 1206–1215. A.3.3

[50] CHEN, M., WALTON, S., BERGER, K., THIYAGALINGAM, J., DUFFY, B., FANG, H., HOLLOWAY, C., AND TREFETHEN, A. E. Visual multiplexing. *Computer Graphics Forum 33*, 3 (2014), 241–250. 2.2, A.3.3

[51] CHI, E. H.-H., RIEDL, J., BARRY, P., AND KONSTAN, J. Principles for information visualization spreadsheets. *IEEE Computer Graphics and Applications*, 4 (1998), 30–38. 2.2

[52] CLAESSEN, J. H., AND VAN WIJK, J. J. Flexible linked axes for multivariate data visualization. *IEEE Transactions on Visualization and Computer Graphics 17*, 12 (2011), 2310–2316. 2.2

[53] CLEVELAND, W. S., AND MCGILL, R. Graphical perception: Theory, experimentation, and application to the development of graphical methods. *Journal of the American Statistical Association 79*, 387 (1984), 531–554. 3.1, A.3.1

[54] COLLBERG, C., KOBOUROV, S., NAGRA, J., PITTS, J., AND WAMPLER, K. A system for graph-based visualization of the evolution of software. In *Proceedings of ACM SOFTVIS* (2003), pp. 77–86. B.2

[55] CONNOR, C. E., EGETH, H. E., AND YANTIS, S. Visual attention: Bottom-up versus top-down. *Current Biology 14*, 19 (2004), R850–R852. A.3.1

[56] CUENCA, E., SALLABERRY, A., WANG, F. Y., AND PONCELET, P. MultiStream: A multiresolution streamgraph approach to explore hierarchical time series. *IEEE Transactions on Visualization and Computer Graphics 24*, 12 (2018), 3160–3173. C.1, C.2

[57] CUI, W., LIU, S., TAN, L., SHI, C., SONG, Y., GAO, Z., QU, H., AND TONG, X. Textflow: Towards better understanding of evolving topics in text. *IEEE Transactions on Visualization and Computer Graphics 17*, 12 (2011), 2412–2421. C.2, D.2, D.4

[58] CUI, W., LIU, S., WU, Z., AND WEI, H. How hierarchical topics evolve in large text corpora. *IEEE Transactions on Visualization and Computer Graphics 20*, 12 (2014), 2281–2290. C.1, C.2

[59] D'AMBROS, M., LANZA, M., AND LUNGU, M. Visualizing co-change information with the evolution radar. *IEEE Transactions on Software Engineering 35*, 5 (2009), 720–735. B.2

[60] DAUGMAN, J. G. Uncertainty relation for resolution in space, spatial frequency, and orientation optimized by two-dimensional visual cortical filters. *Journal of the Optical Society of America 2*, 7 (1985), 1160–1169. A.3.1

[61] DEMIRALP, Ç., SCHEIDEGGER, C. E., KINDLMANN, G. L., LAIDLAW, D. H., AND HEER, J. Visual embedding: A model for visualization. *IEEE Computer Graphics and Applications 34*, 1 (2014), 10–15. 3.1, A.3.3

[62] DIBIA, V., AND DEMIRALP, Ç. Data2vis: Automatic generation of data visualizations using sequence-to-sequence recurrent neural networks. *IEEE Computer Graphics and Applications 39*, 5 (2019), 33–46. 2.2

[63] DIEHL, A., ABDUL-RAHMAN, A., EL-ASSADY, M., BACH, B., KEIM, D., AND CHEN, M. VisGuides: A forum for discussing visualization guidelines. In *Proceedings of EuroVis (Short Papers)* (2018), pp. 61–65. A.4

[64] DOU, W., YU, L., WANG, X., MA, Z., AND RIBARSKY, W. HierarchicalTopics: Visually exploring large text collections using topic hierarchies. *IEEE Transactions on Visualization and Computer Graphics 19*, 12 (2013), 2002–2011. C.2

[65] EICK, S. G., STEFFEN, J. L., AND SUMNER, E. E. SeeSoft: a tool for visualizing line-oriented software statistics. *IEEE Transactions on Software Engineering 18*, 11 (1992), 957–968. B.2

[66] FILONIK, D., AND BAUR, D. Measuring aesthetics for information visualization. In *Proceedings of the International Conference on Information Visualization* (2009), pp. 579–584. A.3.4

[67] FIŠER, J., JAMRIŠKA, O., LUKÁČ, M., SHECHTMAN, E., ASENTE, P., LU, J., AND SỲKORA, D. StyLit: Illumination-guided example-based stylization of 3D renderings. *ACM Transactions on Graphics 35*, 4 (2016), 1–11. 2.3, 2.7

[68] FRIGO, O., SABATER, N., DELON, J., AND HELLIER, P. Split and match: Example-based adaptive patch sampling for unsupervised style transfer. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016), pp. 553–561. 2.3

[69] GATYS, L. A., ECKER, A. S., AND BETHGE, M. Image style transfer using convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016), pp. 2414–2423. 2.3

[70] GERMAN, D. M. An empirical study of fine-grained software modifications. In *Proceedings of the IEEE International Conference on Software Maintenance* (2004), pp. 316–325. B.2

[71] GILSON, O., SILVA, N., GRANT, P. W., AND CHEN, M. From web data to visualization via ontology mapping. *Computer Graphics Forum 27*, 3 (2008), 959–966. 2.2

[72] GLEICHER, M. Considerations for visualizing comparison. *IEEE Transactions on Visualization and Computer Graphics 24*, 1 (2017), 413–423. 2.2

[73] GLEICHER, M., ALBERS, D., WALKER, R., JUSUFI, I., HANSEN, C., AND ROBERTS, J. Visual comparison for information visualization. *Information Visualization 10*, 4 (2011), 289–309. 2.2, 2.3, B.4.2

[74] GOTZ, D., AND WEN, Z. Behavior-driven visualization recommendation. In *Proceedings of the International Conference on Intelligent User Interfaces* (2009), pp. 315–324. 2.2

[75] GOUGH, P. From the analytical to the artistic: A review of literature on information visualization. *Leonardo 50*, 1 (2017), 47–52. 2.3

[76] GRAHAM, M., AND KENNEDY, J. A survey of multiple tree visualisation. *Information Visualization 9*, 4 (2010), 235–252. 2.2

[77] HABER, R. B., AND MCNABB, D. A. Visualization idioms: A conceptual model for scientific visualization systems. In *Visualization in Scientific Computing*. IEEE, 1990, pp. 74–93. 2.2

[78] HAHN, S., TRÜMPER, J., MORITZ, D., AND DÖLLNER, J. Visualization of varying hierarchies by stable layout of Voronoi treemaps. In *Proceedings of IVAPP* (2014), pp. 50–58. C.1, C.2

[79] HANSEN, C. D., AND JOHNSON, C. R. *Visualization handbook*. Elsevier, 2011. B.3

[80] HARRISON, L., REINECKE, K., AND CHANG, R. Infographic aesthetics: Designing for the first impression. In *Proceedings of ACM CHI* (2015), pp. 1187–1190. 1.1, 2.3, 3.1, A.3.4, A.4

[81] HAVRE, S., HETZLER, E., WHITNEY, P., AND NOWELL, L. Themeriver: Visualizing thematic changes in large document collections. *IEEE Transactions on Visualization and Computer Graphics 8*, 1 (2002), 9–20. C.1, C.2, D.2

[82] HEALEY, C., AND ENNS, J. Attention and visual memory in visualization and computer graphics. *IEEE Transactions on Visualization and Computer Graphics 18*, 7 (2012), 1170–1188. A.3.1

[83] HEALEY, C. G. Formalizing artistic techniques and scientific visualization for painted renditions of complex information spaces. In *Proceedings of the International Joint Conference on Artificial Intelligence* (2001), pp. 371–376. 2.3

[84] HEALEY, C. G., BOOTH, K. S., AND ENNS, J. T. High-speed visual estimation using preattentive processing. *ACM Transactions on Computer-Human Interaction 3*, 2 (1996), 107–135. A.3.4

[85] HEER, J., MACKINLAY, J. D., STOLTE, C., AND AGRAWALA, M. Graphical histories for visualization: Supporting analysis, communication, and evaluation. *IEEE Transactions on Visualization and Computer Graphics 14*, 6 (2008), 1189–1196. B.2

[86] HEER, J., VAN HAM, F., CARPENDALE, S., WEAVER, C., AND ISENBERG, P. Creation and collaboration: Engaging new audiences for information visualization. In *Information Visualization*. Springer, 2008, pp. 92–133. 3.2

[87] HEINE, C., LEITTE, H., HLAWITSCHKA, M., IURICICH, F., DE FLORIANI, L., SCHEUERMANN, G., HAGEN, H., AND GARTH, C. A survey of topology-based methods in visualization. *Computer Graphics Forum 35*, 3 (2016), 643–667. C.1

[88] HENRY, N., FEKETE, J.-D., AND MCGUFFIN, M. J. Nodetrix: A hybrid visualization of social networks. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (2007), 1302–1309. 2.2

[89] HERSCHEL, M., DIESTELKÄMPER, R., AND BEN LAHMAR, H. A survey on provenance: What for? What form? What from? *The VLDB Journal 26*, 6 (2017), 881–906. B.2

[90] HOFFSWELL, J., SATYANARAYAN, A., AND HEER, J. Visual debugging techniques for reactive data visualization. *Computer Graphics Forum 35*, 3 (2016), 271–280. B.2

[91] HOLMQUIST, L. E., AND SKOG, T. Informative art: Information visualization in everyday environments. In *Proceedings of GRAPHITE* (2003), pp. 229–235. 2.3

[92] HOLTEN, D., AND VAN WIJK, J. J. Visual comparison of hierarchically organized data. *Computer Graphics Forum 27*, 3 (2008), 759–766. B.2

[93] HU, K., BAKKER, M. A., LI, S., KRASKA, T., AND HIDALGO, C. Vizml: A machine learning approach to visualization recommendation. In *Proceedings of ACM CHI* (2019), pp. 1–12. 2.2

[94] HUANG, M. L., HUANG, T.-H., AND ZHANG, J. TreemapBar: Visualizing additional dimensions of data in bar chart. In *Proceedings of the International Conference on Information Visualization* (2009), pp. 98–103. 2.2

[95] ISAACS, K. E., GIMÉNEZ, A., JUSUFI, I., GAMBLIN, T., BHATELE, A., SCHULZ, M., HAMANN, B., AND BREMER, P.-T. State of the Art of Performance Visualization. In *EuroVis - STARs* (2014), pp. 141–160. B.2

[96] ISENBERG, P., AND CARPENDALE, S. Interactive tree comparison for co-located collaborative information visualization. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (2007), 1232–1239. C.2

[97] ITTI, L., KOCH, C., AND NIEBUR, E. A model of saliency-based visual attention for rapid scene analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence 20*, 11 (1998), 1254–1259. A.3.1

[98] JAENICKE, H., AND CHEN, M. A salience-based quality metric for visualization. *Computer Graphics Forum 29*, 3 (2010), 1183–1192. 3.1, A.3.1

[99] JÄNICKE, H., WEIDNER, T., CHUNG, D., LARAMEE, R. S., TOWNSEND, P., AND CHEN, M. Visual reconstructability as a quality metric for flow visualization. *Computer Graphics Forum 30*, 3 (2011), 781–790. A.1

[100] JANKUN-KELLY, T., AND MA, K.-L. A spreadsheet interface for visualization exploration. In *Proceedings of IEEE VIS* (2000), pp. 69–76. B.3.1

[101] JANKUN-KELLY, T., AND MA, K.-L. Visualization exploration and encapsulation via a spreadsheet-like interface. *IEEE Transactions on Visualization and Computer Graphics 7*, 3 (2001), 275–287. 2.1, B.2

[102] JANKUN-KELLY, T. J., MA, K. L., AND GERTZ, M. A model for the visualization exploration process. In *Proceedings of IEEE VIS* (2002), pp. 323—-330. 2.1

[103] JANKUN-KELLY, T. J., MA, K.-L., AND GERTZ, M. A model and framework for visualization exploration. *IEEE Transactions on Visualization and Computer Graphics 13*, 2 (2007), 357–369. 2.1, A.3.3

[104] JARDINE, N., ONDOV, B. D., ELMQVIST, N., AND FRANCONERI, S. The perceptual proxies of visual comparison. *IEEE Transactions on Visualization and Computer Graphics 26*, 1 (2019), 1012–1021. 2.2

[105] JAVED, W., AND ELMQVIST, N. Exploring the design space of composite visualization. In *Proceedings of IEEE PacificVis* (2012), pp. 1–8. 2.2

[106] JOHANSSON, S., AND JOHANSSON, J. Interactive dimensionality reduction through user-defined combinations of quality metrics. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 993–1000. 3.1

[107] JOHNSON, B., AND SHNEIDERMAN, B. Tree-maps: A space-filling approach to the visualization of hierarchical information structures. In *Proceedings of IEEE VIS* (1991), pp. 284–291. C.2

[108] JOHNSON, B. S. *Treemaps: Visualizing hierarchical and categorical data*. PhD thesis, Department of Computer Science, University of Maryland, College Park, MD, 1993. C.2

[109] JOHNSON, S., SAMSEL, F., ABRAM, G., OLSON, D., SOLIS, A. J., HERMAN, B., WOLFRAM, P. J., LENGLET, C., AND KEEFE, D. F. Artifact-based rendering: Harnessing natural and traditional visual media for more expressive and engaging 3D visualizations. *IEEE Transactions on Visualization and Computer Graphics 26*, 1 (2019), 492–502. 2.3, 2.8

[110] KEY, A., HOWE, B., PERRY, D., AND ARAGON, C. Vizdeck: Self-organizing dashboards for visual analytics. In *Proceedings of ACM SIGMOD* (2012), pp. 681–684. 2.2

[111] KHAN, T., BARTHEL, H., EBERT, A., AND LIGGESMEYER, P. Visualization and evolution of software architectures. In *Proceedings of the Workshop on Visualization of Large and Unstructured Data Sets* (2011), pp. 25–42. B.2

[112] KIM, Y., AND VARSHNEY, A. Saliency-guided enhancement for volume visualization. *IEEE Transactions on Visualization and Computer Graphics 12*, 5 (2006), 925–932. A.3.1

[113] KINDLMANN, G., CHIW, C., SELTZER, N., SAMUELS, L., AND REPPY, J. Diderot: A domain-specific language for portable parallel scientific visualization and image analysis. *IEEE Transactions on Visualization and Computer Graphics 22*, 1 (2016), 867–876. 4.1, B.3.2, B.5

[114] KINDLMANN, G., AND SCHEIDEGGER, C. An algebraic process for visualization design. *IEEE Transactions on Visualization and Computer Graphics 20*, 12 (2014), 2181–2190. 3.1, A.3.3

[115] KIRBY, R. M., KEEFE, D., AND LAIDLAW, D. H. Painting and visualization. In *The Visualization Handbook*. Elsevier, 2005, pp. 873–891. 2.3

[116] KIRBY, R. M., MARMANIS, H., AND LAIDLAW, D. H. Visualizing multivalued data from 2D incompressible flows using concepts from painting. In *Proceedings of IEEE VIS* (1999), pp. 333–540. 2.3

[117] KLUYVER, T., RAGAN-KELLEY, B., PÉREZ, F., GRANGER, B. E., BUSSONNIER, M., FREDERIC, J., KELLEY, K., HAMRICK, J. B., GROUT, J., CORLAY, S., ET AL. Jupyter notebooks – a publishing format for reproducible computational workflows. In *Proceedings of ELPUB* (2016), pp. 87–90. B.2

[118] KOENIG, M., SPINDLER, W., REXILIUS, J., JOMIER, J., LINK, F., AND PEITGEN, H.-O. Embedding VTK and ITK into a visual programming and rapid prototyping platform. In *Proceedings of SPIE* (2006), vol. 6141, pp. 796–806. 2.2, B.2

[119] KOLESÁR, I., BRUCKNER, S., VIOLA, I., AND HAUSER, H. A fractional cartesian composition model for semi-spatial comparative visualization design.

IEEE Transactions on Visualization and Computer Graphics 23, 1 (2016), 851–860. 2.2

[120] KOOP, D., SCHEIDEGGER, C., CALLAHAN, S., FREIRE, J., AND SILVA, C. Viscomplete: Data-driven suggestions for visualization systems. *IEEE Transactions on Visualization and Computer Graphics 14*, 6 (2008), 1691–1698. 2.2

[121] KÖPP, W., AND WEINKAUF, T. Temporal Treemaps: Static visualization of evolving trees. *IEEE Transactions on Visualization and Computer Graphics 25*, 1 (2019), 534–543. 3.3.2, C.1, C.2, C.3.2.3, C.6.1, C.7

[122] KOSARA, R. Visualization criticism-the missing link between information visualization and art. In *Proceedings of the International Conference on Information Visualization* (2007), pp. 631–636. 2.3

[123] KOSARA, R. An empire built on sand: Reexamining what we think we know about visualization. In *Proceedings of the BELIV Workshop* (2016), pp. 162–168. A.4

[124] KYPRIANIDIS, J. E., COLLOMOSSE, J., WANG, T., AND ISENBERG, T. State of the "art": A taxonomy of artistic stylization techniques for images and video. *IEEE Transactions on Visualization and Computer Graphics 19*, 5 (2012), 866–885. 2.3

[125] LAMPE, O. D., CORREA, C., MA, K.-L., AND HAUSER, H. Curve-centric volume reformation for comparative visualization. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 1235–1242. 2.2

[126] LANG, A. Aesthetics in information visualization. In *Trends in Information Visualization*. University of Munich, 2010, pp. 8–14. 2.3

[127] LANZA, M. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of the International Workshop on Principles of Software Evolution* (2001), pp. 37–42. B.2

[128] LARAMEE, R. Using visualization to debug visualization software. *IEEE Computer Graphics and Applications*, 6 (2009), 67–73. B.1, B.3.1

[129] LAU, A., AND MOERE, A. V. Towards a model of information aesthetics in information visualization. In *Proceedings of the International Conference on Information Visualization* (2007), pp. 87–92. 2.3, A.3.4

[130] LEE, C. H., VARSHNEY, A., AND JACOBS, D. W. Mesh saliency. *ACM Transactions on Graphics 24*, 3 (2005), 659–666. A.3.1

[131] LEIN, E. S., HAWRYLYCZ, M. J., AO, N., AYRES, M., BENSINGER, A., BERNARD, A., BOE, A. F., BOGUSKI, M. S., BROCKWAY, K. S., BYRNES, E. J., ET AL. Genome-wide atlas of gene expression in the adult mouse brain. *Nature 445*, 7124 (2007), 168–176. 4.5, 4.2

[132] LI, Q., ZACHMANN, G., FENG, D., HUANG, K., AND MACHIRAJU, R. 2013 ieee scientific visualization contest winner: Observing genomics and phenotypical patterns in the developing mouse brain. *IEEE Computer Graphics and Applications 34*, 5 (2014), 88–97. 4.5, 4.2

[133] LI, Z. A neural model of contour integration in the primary visual cortex. *Neural Computation 10*, 4 (1998), 903–940. A.3.1

[134] LIU, B., WUENSCHE, B., AND ROPINSKI, T. Visualization by example-a constructive visual component-based interface for direct volume rendering. In *Proceedings of GRAPP* (2010), pp. 254–259. 2.2

[135] LIU, S., WU, Y., WEI, E., LIU, M., AND LIU, Y. Storyflow: Tracking the evolution of stories. *IEEE Transactions on Visualization and Computer Graphics 19*, 12 (2013), 2436–2445. 3.3.2, C.2, D.2, D.4

[136] LIU, Z., NERSESSIAN, N., AND STASKO, J. Distributed cognition as a theoretical framework for information visualization. *IEEE Transactions on Visualization and Computer Graphics 14*, 6 (2008), 1173–1180. 3.1, A.3.3

[137] LUKASCZYK, J., WEBER, G., MACIEJEWSKI, R., GARTH, C., AND LEITTE, H. Nested tracking graphs. *Computer Graphics Forum 36*, 3 (2017), 12–22. 3.3.1, C.1, C.2, C.6.1

[138] MA, K. Image graphs - A novel approach to visual data exploration. In *Proceedings of IEEE VIS* (1999), pp. 81–88. 2.1, B.2

[139] MACKINLAY, J. Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics 5*, 2 (1986), 110–141. 2.2, 3.1, A.3.1, A.3.3

[140] MACKINLAY, J., HANRAHAN, P., AND STOLTE, C. Show me: Automatic presentation for visual analysis. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (2007), 1137–1144. 2.2, A.3.1

[141] MALIK, M. M., HEINZL, C., AND GROELLER, M. E. Comparative visualization for parameter studies of dataset series. *IEEE Transactions on Visualization and Computer Graphics 16*, 5 (2010), 829–840. 2.2

[142] MARCUS, A., FENG, L., AND MALETIC, J. 3D representations for software visualization. In *Proceedings of ACM SOFTVIS* (2003), pp. 27–36. B.2

[143] MARKS, J., BEARDSLEY, P., ANDALMAN, B., FREEMAN, W., GIBSON, S., HODGINS, J., KANG, T., MIRTICH, B., PFISTER, H., RUML, W., ET AL. Design galleries: A general approach to setting parameters for computer graphics and animation. In *Proceedings of ACM SIGGRAPH* (1997), pp. 389–400. 2.1, 3.2.1, 3.2.2, B.2

[144] MATZEN, L. E., HAASS, M. J., DIVIS, K. M., WANG, Z., AND WILSON, A. T. Data visualization saliency model: A tool for evaluating abstract data visualizations. *IEEE Transactions on Visualization and Computer Graphics 24*, 1 (2018), 563–573. A.3.1

[145] MERINO, L., GHAFARI, M., AND NIERSTRASZ, O. Towards actionable visualisation in software development. In *Proceedings of IEEE VISSOFT* (2016), pp. 61–70. B.2

[146] MUNROE, R. Movie narrative charts, 2009. Diagram available at https://xkcd.com/657. C.2, D.1

[147] MUNZNER, T. A nested model for visualization design and validation. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 921–928. A.1, A.1

[148] MUNZNER, T., GUIMBRETIÈRE, F., TASIRAN, S., ZHANG, L., AND ZHOU, Y. TreeJuxtaposer: Scalable tree comparison using focus+ context with guaranteed visibility. *ACM Transactions on Graphics 22*, 3 (2003), 453–462. C.2

[149] NEUMANN, P., SCHLECHTWEG, S., AND CARPENDALE, S. ArcTrees: Visualizing relations in hierarchical data. In *Proceedings of EG/IEEE VGTC EuroVis* (2005), pp. 53–60. C.2

[150] OGAWA, M., AND MA, K.-L. Software evolution storylines. In *Proceedings of ACM SOFTVIS* (2010), pp. 35–42. 3.3.2, B.2, D.2

[151] ONDOV, B., JARDINE, N., ELMQVIST, N., AND FRANCONERI, S. Face to face: Evaluating visual comparison. *IEEE Transactions on Visualization and Computer Graphics 25*, 1 (2018), 861–871. 2.2

[152] PAGENDARM, H.-G., AND POST, F. H. Comparative visualization: Approaches and examples. In *Visualization in Scientific Computing* (1995), B. U. M. Göbel, H. Müller, Ed., Springer, pp. 95—-108. 2.2

[153] PIMENTEL, J. F., FREIRE, J., BRAGANHOLO, V., AND MURTA, L. Tracking and analyzing the evolution of provenance from scripts. In *Proceedings of the International Provenance and Annotation Workshop* (2016), pp. 16–28. B.2

[154] PINEO, D., AND WARE, C. Data visualization optimization via computational modeling of perception. *IEEE Transactions on Visualization and Computer Graphics 18*, 2 (2012), 309–320. 3.1, A.3.1

[155] PINTO, Y., VAN DER LEIJ, A. R., SLIGTE, I. G., LAMME, V. A. F., AND SCHOLTE, H. S. Bottom-up and top-down attention are independent. *Journal of Vision 13*, 3 (2013), 16. A.3.1

[156] PURCHASE, H. Which aesthetic has the greatest effect on human understanding? In *Proceedings of the International Symposium on Graph Drawing* (1997), pp. 248–261. 3.1

[157] PURCHASE, H. C. Effective information visualisation: A study of graph drawing aesthetics and algorithms. *Interacting with Computers 13*, 2 (2000), 147–162. 1

[158] PURCHASE, H. C., ANDRIENKO, N., JANKUN-KELLY, T., AND WARD, M. Theoretical foundations of information visualization. In *Information Visualization*. Springer, 2008, pp. 46–64. 3.1, A.3.3

[159] RAGAN, E. D., ENDERT, A., SANYAL, J., AND CHEN, J. Characterizing provenance in visualization and data analysis: An organizational framework of provenance types and purposes. *IEEE Transactions on Visualization and Computer Graphics 22*, 1 (2015), 31–40. 2.2

[160] RAMIREZ GAVIRIA, A. When is information visualization art? Determining the critical criteria. *Leonardo 41*, 5 (2008), 479–482. 2.3

[161] REN, L., CUI, J., DU, Y., AND DAI, G. Multilevel interaction model for hierarchical tasks in information visualization. In *Proceedings of the International Symposium on Visual Information Communication and Interaction* (2013), pp. 11–16. A.1

[162] RODIECK, R. Quantitative analysis of cat retinal ganglion cell response to visual stimuli. *Vision Research 5*, 12 (1965), 583–601. A.3.1

[163] ROTH, S. F., KOLOJEJCHICK, J., MATTIS, J., AND GOLDSTEIN, J. Interactive graphic design using automatic presentation knowledge. In *Proceedings of ACM CHI* (1994), pp. 112–117. 2.2

[164] SACHA, D., STOFFEL, A., STOFFEL, F., KWON, B. C., ELLIS, G., AND KEIM, D. A. Knowledge generation model for visual analytics. *IEEE Transactions on Visualization and Computer Graphics 20*, 12 (2014), 1604–1613. A.4

[165] SAKET, B., ENDERT, A., AND STASKO, J. Beyond usability and performance: A review of user experience-focused evaluations in visualization. In *Proceedings of the BELIV Workshop* (2016), pp. 133–142. A.3.4

[166] SAKET, B., SCHEIDEGGER, C., AND KOBOUROV, S. Comparing node-link and node-link-group visualizations from an enjoyment perspective. *Computer Graphics Forum 35*, 3 (2016), 41–50. 3.1, A.3.4

[167] SAKET, B., SCHEIDEGGER, C., AND KOBOUROV, S. G. Towards Understanding Enjoyment and Flow in Information Visualization. In *Proceedings of EuroVis (Short Papers)* (2015). A.3.4

[168] SAMSEL, F., BARTRAM, L., AND BARES, A. Art affect and color: Creating engaging expressive scientific visualization. In *Proceedings of the IEEE VIS Arts Program* (2018), pp. 1–9. 2.3

[169] SANTOS, E., LINS, L., AHRENS, J., FREIRE, J., AND SILVA, C. Vismashup: Streamlining the creation of custom visualization applications. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 1539–1546. 2.2

[170] SATYANARAYAN, A., MORITZ, D., WONGSUPHASAWAT, K., AND HEER, J. Vega-Lite: A grammar of interactive graphics. *IEEE Transactions on Visualization and Computer Graphics 23*, 1 (2016), 341–350. 3.2.1, B.1, B.2, B.8

[171] SCHEIDEGGER, C., VO, H., KOOP, D., FREIRE, J., AND SILVA, C. Querying and creating visualizations by analogy. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (2007), 1560–1567. 2.2, 2.5

[172] SCHMIDT, J., GRÖLLER, M. E., AND BRUCKNER, S. VAICo: Visual analysis for image comparison. *IEEE Transactions on Visualization and Computer Graphics 19*, 12 (2013), 2090–2099. 2.2

[173] SCHMIDT, J., PREINER, R., AUZINGER, T., WIMMER, M., GRÖLLER, M. E., AND BRUCKNER, S. YMCA - Your mesh comparison application. In *Proceedings of IEEE VAST* (2014), pp. 153–162. 2.2

[174] SCHROEDER, W. J., LORENSEN, B., AND MARTIN, K. *The visualization toolkit: An object-oriented approach to 3D graphics*. Kitware, 2004. 2.2, B.2

[175] SCHULZ, H. Treevis.net: A tree visualization reference. *IEEE Computer Graphics and Applications 31*, 6 (2011), 11–15. A.1, A.1, C.1, C.2

[176] SCHULZ, H.-J., AND HADLAK, S. Preset-based generation and exploration of visualization designs. *Journal of Visual Languages & Computing 31* (2015), 9–29. 2.2, 2.6

[177] SEDLMAIR, M., HEINZL, C., BRUCKNER, S., PIRINGER, H., AND MÖLLER, T. Visual parameter space analysis: A conceptual framework. *IEEE Transactions on Visualization and Computer Graphics 20*, 12 (2014), 2161–2170. 2.1, B.2

[178] SELIM, A., ELGHARIB, M., AND DOYLE, L. Painting style transfer for head portraits using convolutional neural networks. *ACM Transactions on Graphics 35*, 4 (2016), 1–18. 2.3

[179] SHNEIDERMAN, B. The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings of the IEEE Symposium on Visual Languages* (1996), pp. 336–343. 1, A.1

[180] SHOEMAKE, K. ARCBALL: A user interface for specifying three-dimensional orientation using a mouse. In *Proceedings of Graphics Interface* (1992), pp. 151–156. B.4.3

[181] SILVA, C. T., ANDERSON, E., SANTOS, E., AND FREIRE, J. Using vistrails and provenance for teaching scientific visualization. *Computer Graphics Forum 30*, 1 (2011), 75–84. 2.2

[182] SILVER, D. Object-oriented visualization. *IEEE Computer Graphics and Applications 15*, 3 (1995), 54–62. A.3.3

[183] SKAU, D., HARRISON, L., AND KOSARA, R. An evaluation of the impact of visual embellishments in bar charts. *Computer Graphics Forum 34*, 3 (2015), 221–230. 3.1, A.3, A.3.4

[184] SONDAG, M., SPECKMANN, B., AND VERBEEK, K. Stable treemaps via local moves. *IEEE Transactions on Visualization and Computer Graphics 24*, 1 (2018), 729–738. C.1, C.2

[185] STAB, C., NAZEMI, K., AND FELLNER, D. W. SemaTime - Timeline visualization of time-dependent relations and semantics. In *Proceedings of the International Symposium on Visual Computing* (2010), pp. 514–523. C.2

[186] STITZ, H., GRATZL, S., PIRINGER, H., ZICHNER, T., AND STREIT, M. KnowledgePearls: Provenance-based visualization retrieval. *IEEE Transactions on Visualization and Computer Graphics 25*, 1 (2018), 120–130. B.2

[187] STOLL, C., GUMHOLD, S., AND SEIDEL, H.-P. Visualization with stylized line primitives. In *Proceedings of IEEE VIS* (2005), pp. 695–702. B.6.2

[188] SUD, A., FISHER, D., AND LEE, H.-P. Fast dynamic Voronoi treemaps. In *Proceedings of the International Symposium on Voronoi Diagrams in Science and Engineering* (2010), pp. 85–94. C.1, C.2

[189] SUGIYAMA, K., TAGAWA, S., AND TODA, M. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics 11*, 2 (1981), 109–125. D.2, D.4

[190] TAK, S., AND COCKBURN, A. Enhanced spatial stability with Hilbert and Moore treemaps. *IEEE Transactions on Visualization and Computer Graphics 19*, 1 (2013), 141–148. C.1, C.2

[191] TAL, E. Measurement in science. In *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, 2017. A.2

[192] TANAHASHI, Y., AND MA, K.-L. Design considerations for optimizing storyline visualizations. *IEEE Transactions on Visualization and Computer Graphics 18*, 12 (2012), 2679–2688. 3.3.2, 3.3.2, C.2, D.2

[193] TANG, T., RUBAB, S., LAI, J., CUI, W., YU, L., AND WU, Y. iStoryline: Effective convergence to hand-drawn storylines. *IEEE Transactions on Visualization and Computer Graphics 25*, 1 (2018), 769–778. 3.3.2, D.2

[194] TATEOSIAN, L. G., HEALEY, C. G., AND ENNS, J. T. Engaging viewers through nonphotorealistic visualizations. In *Proceedings of the International Symposium on Non-Photorealistic Animation and Rendering* (2007), pp. 93–102. 2.3, 2.8

[195] TAYLOR, I., SHIELDS, M., WANG, I., AND HARRISON, A. Visual grid workflow in Triana. *Journal of Grid Computing 3*, 3–4 (2005), 153–169. B.2

[196] TELEA, A. Combining extended table lens and treemap techniques for visualizing tabular data. In *Proceedings of EG/IEEE VGTC EuroVis* (2006), pp. 51–58. 2.2

[197] TELEA, A., AND AUBER, D. Code flows: Visualizing structural evolution of source code. *Computer Graphics Forum 27*, 3 (2008), 831–838. B.1, B.2

[198] THERÓN, R. Hierarchical-temporal data visualization using a tree-ring metaphor. In *Proceedings of the International Symposium on Smart Graphics* (2006), pp. 70–81. C.2

[199] TORY, M., AND MÖLLER, T. Rethinking visualization: A high-level taxonomy. In *Proceedings of IEEE InfoVis* (2004), pp. 151–158. A.1

[200] TORY, M., AND MÖLLER, T. Evaluating visualizations: Do expert reviews work? *IEEE Computer Graphics and Applications 25*, 5 (2005), 8–11. 4.1, B.7

[201] TORY, M., POTTS, S., AND MÖLLER, T. A parallel coordinates style interface for exploratory volume visualization. *IEEE Transactions on Visualization and Computer Graphics 11*, 1 (2005), 71–80. 2.1

[202] TRACTINSKY, N., KATZ, A. S., AND IKAR, D. What is beautiful is usable. *Interacting with Computers 13*, 2 (2000), 127–145. 2.3, A.3.4

[203] TU, Y., AND SHEN, H.-W. Visualizing changes of hierarchical data using treemaps. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (2007), 1286–1293. C.1, C.2

[204] TUFTE, E. R. *The visual display of quantitative information*, vol. 2. Graphics press Cheshire, CT, 2001. 3.1, A.3.3

[205] TUKEY, J. W., AND TUKEY, P. A. Computer graphics and exploratory data analysis: An introduction. In *Proceedings of Computer Graphics* (1985), pp. 773–785. A.3.2

[206] VAN DEN ELZEN, S., AND VAN WIJK, J. J. Small multiples, large singles: A new approach for visual data exploration. *Computer Graphics Forum 32*, 3pt2 (2013), 191–200. 2.2, 2.1

[207] VAN HEES, R., AND HAGE, J. Stable and predictable Voronoi treemaps for software quality monitoring. *Information and Software Technology 87* (2017), 242–258. C.1, C.2

[208] VAN WIJK, J. J. The value of visualization. In *Proceedings of IEEE VIS* (2005), pp. 79–86. A.3.4

[209] VAN WIJK, J. J., AND VAN DE WETERING, H. Cushion treemaps: Visualization of hierarchical information. In *Proceedings of IEEE InfoVis* (1999), pp. 73–78. 3.3.2

[210] VERMA, V., AND PANG, A. Comparative flow visualization. *IEEE Transactions on Visualization and Computer Graphics 10*, 6 (2004), 609–624. 2.2

[211] VERNIER, E., SONDAG, M., COMBA, J., SPECKMANN, B., TELEA, A., AND VERBEEK, K. Quantitative comparison of time-dependent treemaps. *Computer Graphics Forum* (2020). 3.1, 4.2, C.2, C.4.5, C.5.2

[212] VICKERS, P., FAITH, J., AND ROSSITER, N. Understanding visualization: A formal approach using category theory and semiotics. *IEEE Transactions on Visualization and Computer Graphics 19*, 6 (2012), 1048–1061. A.3.3

[213] VIÉGAS, F. B., AND WATTENBERG, M. Artistic data visualization: Beyond visual analytics. In *Proceedings of the International Conference on Online Communities and Social Computing* (2007), pp. 182–191. 2.3

[214] VOIGT, M., PIETSCHMANN, S., GRAMMEL, L., AND MEISSNER, K. Context-aware recommendation of visualization components. In *Proceedings of the International Conference on Information, Process, and Knowledge Management* (2012), pp. 101–109. 2.2

[215] VOINEA, L., TELEA, A., AND VAN WIJK, J. J. CVSscan: visualization of code evolution. In *Proceedings of ACM SOFTVIS* (2005), pp. 47–56. B.1, B.2

[216] WANG, C., AND SHEN, H.-W. Information theory in scientific visualization. *Entropy 13*, 1 (2011), 254–273. A.3.3

[217] WANG, X., DOU, W., BUTKIEWICZ, T., BIER, E. A., AND RIBARSKY, W. A two-stage framework for designing visual analytics system in organizational environments. In *Proceedings of IEEE VAST* (2011), pp. 251–260. A.1

[218] WATTENBERG, M., AND KRISS, J. Designing for social data analysis. *IEEE Transactions on Visualization and Computer Graphics 12*, 4 (2006), 549–557. C.2

[219] WETTEL, R., AND LANZA, M. Visual exploration of large-scale system evolution. In *Proceedings of the Working Conference on Reverse Engineering* (2008), pp. 219–228. B.2

[220] WILKINSON, L., ANAND, A., AND GROSSMAN, R. Graph-theoretic scagnostics. In *Proceedings of IEEE InfoVis* (2005), pp. 157–164. A.3.2

[221] WILLIAMSON, J. R., AND GROSSBERG, S. A neural model of how horizontal and interlaminar connections of visual cortex develop into adult circuits that carry out perceptual grouping and learning. *Cerebral Cortex 11*, 1 (2001), 37–58. A.3.1

[222] WINKENBACH, G., AND SALESIN, D. H. Computer-generated pen-and-ink illustration. In *Proceedings of ACM SIGGRAPH* (1994), pp. 91–100. 2.3

[223] WITTENHAGEN, M., CHEREK, C., AND BORCHERS, J. Chronicler: Interactive exploration of source code history. In *Proceedings of ACM CHI* (2016), pp. 3522–3532. 3.3.1, B.1, B.2, C.1, C.2, C.3.2.2, C.6.1

[224] WONGSUPHASAWAT, K., AND GOTZ, D. Exploring flow, factors, and outcomes of temporal event sequences with the outflow visualization. *IEEE Transactions on Visualization and Computer Graphics 18*, 12 (2012), 2659–2668. C.2

[225] WONGSUPHASAWAT, K., MORITZ, D., ANAND, A., MACKINLAY, J., HOWE, B., AND HEER, J. Towards a general-purpose query language for visualization recommendation. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics* (2016), pp. 1–6. 2.2

[226] WOOD, J., KACHKAEV, A., AND DYKES, J. Design exposition with literate visualization. *IEEE Transactions on Visualization and Computer Graphics 25*, 1 (2019), 759–768. B.2

[227] WOODRING, J., AND SHEN, H.-W. Multi-variate, time varying, and comparative visualization with contextual cues. *IEEE Transactions on Visualization and Computer Graphics 12*, 5 (2006), 909–916. 2.2

[228] WU, Y., XU, A., CHAN, M.-Y., QU, H., AND GUO, P. Palette-style volume visualization. In *Proceedings of the EG/IEEE International Symposium on Volume Graphics* (2007), pp. 33–40. 2.2

[229] XU, L., LEE, T.-Y., AND SHEN, H.-W. An information-theoretic framework for flow visualization. *IEEE Transactions on Visualization and Computer Graphics 16*, 6 (2010), 1216–1224. 3.1, A.3.3

[230] XU, P., WU, Y., WEI, E., PENG, T.-Q., LIU, S., ZHU, J. J., AND QU, H. Visual analysis of topic competition on social media. *IEEE Transactions on Visualization and Computer Graphics 19*, 12 (2013), 2012–2021. D.2

[231] ZHANG, C., SCHULTZ, T., LAWONN, K., EISEMANN, E., AND VILANOVA, A. Glyph-based comparative visualization for diffusion tensor fields. *IEEE Transactions on Visualization and Computer Graphics 22*, 1 (2015), 797–806. 2.2

[232] ZHI, J., AND RUHE, G. DEVis: a tool for visualizing software document evolution. In *Proceedings of IEEE VISSOFT* (2013), pp. 1–4. B.2

[233] ZHOU, H., CHEN, M., AND WEBSTER, M. F. Comparative evaluation of visualization and experimental results using image comparison metrics. In *Proceedings of IEEE VIS* (2002), pp. 315–322. 2.2

[234] ZHOU, M. X., AND CHEN, M. Automated generation of graphic sketches by example. In *Proceedings of the International Joint Conference on Artificial Intelligence* (2003), vol. 3, pp. 65–71. 2.2

Graphic design: Communication Division, UiB / Print: Skipnes Kommunikasjon AS

uib.no

ISBN: 9788230846575 (print)
9788230850176 (PDF)