

Gizual User Interface: Browser-Based Visualisation for Git Repositories

Andreas Steinkellner



Gizual User Interface: Browser-Based Visualisation for Git Repositories

Andreas Steinkellner B.Sc.

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's Degree Programme: Software Engineering and Management

submitted to

Graz University of Technology

Supervisor

Ao.Univ.-Prof. Dr. Keith Andrews
Institute of Interactive Systems and Data Science (ISDS)

Graz, 09 Dec 2024

© Copyright 2024 by Andreas Steinkellner, except as otherwise noted.

This work is placed under a Creative Commons Attribution 4.0 International (CC BY 4.0) licence.

Gizual Benutzeroberfläche: Browserbasierte Visualisierung von Git-Repositories

Andreas Steinkellner B.Sc.

Masterarbeit

für den akademischen Grad

Diplom-Ingenieur

Masterstudium: Software Engineering and Management

an der

Technischen Universität Graz

Begutachter

Ao.Univ.-Prof. Dr. Keith Andrews
Institute of Interactive Systems and Data Science (ISDS)

Graz, 09 Dec 2024

Diese Arbeit ist in englischer Sprache verfasst.

© Copyright 2024 Andreas Steinkellner, sofern nicht anders gekennzeichnet.

Diese Arbeit steht unter der Creative Commons Attribution 4.0 International (CC BY 4.0) Lizenz.

Abstract

This thesis presents the user interface and frontend of Gizual, an open-source, fully web-based information visualisation tool for exploring Git source code repositories. It uses an underlying WebAssembly core to parse the repository history. All data processing is done locally inside the browser. Specific branches or commit ranges can be specified in Gizual's query interface, or through an interactive timeline. Files can be filtered based on file name or extension, or via a highly performant file tree component.

The visualisation itself is based on the idea of an information mural of listings, arranged in a masonry layout on an interactive two-dimensional canvas. The canvas is freely zoomable using mousewheel or pinch zoom. Files of source code are represented as tiles in the masonry grid. Lines of code are represented as coloured strips within each tile. Two styles of visual encoding are available: one based on the age of each line of code, and the other on the last author of the line of code.

The frontend is written in TypeScript, using React and MobX. The Mantine library is used as a component base, and UI components are styled using SCSS.

Kurzfassung

Diese Arbeit präsentiert die Benutzeroberfläche und das Frontend von Gizual, einem Open-Source Web-Tool zur Informationsvisualisierung, das zur Erkundung von Git-Repositories verwendet wird. Gizual nutzt einen WebAssembly-Kern, um die Daten aus dem Git-Repository zu verarbeiten. Alle Daten werden lokal innerhalb des Webbrowsers verarbeitet. Spezifische Git-Branches und Zeitbereiche können über die Query-Schnittstelle von Gizual oder eine interaktive Zeitleiste eingegeben werden. Dateien können über Dateinamen, Erweiterungen, oder durch einen performanten Verzeichnisbaum ausgewählt werden.

Die Visualisierung basiert auf dem Konzept eines Information Murals, bestehend aus Auflistungen von Quellcode, die in einem Raster auf einer interaktiven zweidimensionalen Leinwand platziert sind. Die Leinwand kann mit dem Mausrad oder per Touch-Geste vergrößert oder verkleinert werden. Einzelne Dateien werden in dem Raster als Kacheln repräsentiert. Einzelne Zeilen werden als farbige Bänder innerhalb einer Kachel dargestellt. Zwei Arten der visuellen Kodierung stehen zur Verfügung: eine basiert auf dem Alter der Zeile und die andere auf dem letzten Autor der Zeile.

Das Frontend ist in TypeScript geschrieben und nutzt React und MobX. Die Mantine-Bibliothek fungiert als die Basis für einzelne Komponenten in der Benutzeroberfläche, während SCSS für die Anpassung des Stils genutzt wird.

Contents

Contents	iv
List of Figures	vi
List of Tables	vii
List of Listings	ix
Acknowledgements	xi
Credits	xiii
1 Introduction	1
2 Visualising Software	3
2.1 Information Visualisation	3
2.2 Software Visualisation.	5
2.3 Visualising Software Repositories	5
3 Frontend Web Design	11
3.1 The Modern Web Browser	11
3.2 HTML	12
3.2.1 Semantic Markup	12
3.2.2 ARIA	12
3.3 CSS	14
3.3.1 CSS Syntax	14
3.3.2 CSS Origin Types	14
3.4 The Rendering Pipeline	16
3.4.1 Parsing and Tokenisation	16
3.4.2 Value Computation	16
3.4.3 Cascade	17
3.4.4 CSS Object Model (CSSOM).	17
3.4.5 Layouting	17
3.4.6 Painting	19
3.4.7 Composition	19

3.5	JavaScript	19
3.6	TypeScript	20
3.7	Frontend Frameworks	21
3.7.1	React	22
3.7.2	Angular.	23
3.7.3	Vue	23
3.7.4	Svelte	23
3.7.5	Choosing React	24
3.8	Building Web Applications	24
3.8.1	Build Tools and Bundlers	24
3.8.2	Package Managers	25
3.9	Responsive Web Design	26
4	Gizual Architecture	31
4.1	Architectural Requirements	31
4.1.1	Non-Blocking	31
4.1.2	Asynchronicity	32
4.1.3	Parallel Execution	32
4.1.4	Separation of Concerns	32
4.2	Architectural Overview	32
4.2.1	Explorer Pool	34
4.2.2	Renderer Pool	34
4.2.3	SQLite Database	34
4.2.4	UI Controllers	34
4.2.5	UI Components	35
4.2.6	Maestro.	35
4.3	State Management	36
4.3.1	Introduction to MobX	36
4.3.2	State Management in React.	36
4.3.3	Advanced Reactivity Within MobX.	37
4.3.4	MobX Usage in Gizual	39
4.4	Query Interface	39
4.4.1	Scope <code>commit-range</code>	39
4.4.2	Scope <code>files</code>	39
4.4.3	Scope <code>visualisation</code>	41
5	Gizual User Interface	43
5.1	Design Principles	43
5.2	Design Tooling	44
5.2.1	Figma	44
5.2.2	User Interface Component Libraries	45

5.3	User Interface Components in Gizual	47
5.4	Previous Design Iterations	51
5.5	Current User Interface.	52
5.6	Designing for Interactivity	53
6	Gizual Canvas	55
6.1	File Tiles	55
6.2	Masonry Canvas	60
6.2.1	Canvas Interactivity	60
6.2.2	Canvas Minimap	60
6.2.3	Canvas Legend	60
6.2.4	Author Panel	60
7	Visual Encoding in Gizual	63
7.1	Colour Spaces.	63
7.2	Gizual Colour Manager	64
8	Query Bar	67
8.1	Query Modules	67
8.1.1	The Time Module	67
8.1.2	The File Module	69
8.1.3	The Vis Module.	69
8.2	Implementation Details	71
8.2.1	Query Editor	71
8.2.2	Query Assistant	75
8.3	Previous Iterations and Concepts	76
8.3.1	QB1: Single Input Field.	76
8.3.2	QB2: Advanced Query Editor.	76
8.3.3	QB3: Hybrid Combination of Modules and JSON Input	76
9	Selected Details of the Implementation	79
9.1	Interactive SVG Timeline	79
9.1.1	Commit Timeline	79
9.1.2	Time Ruler	79
9.1.3	Range Selector	80
9.2	File Tree.	80
10	Outlook and Future Work	83
11	Concluding Remarks	85
A	User Guide	87
A.1	Opening a Repository	87

A.2	Navigating the User Interface	87
A.3	Modifying the Query	89
A.3.1	Customising the Commit Range	90
A.3.2	Customising Selected Files.	90
A.3.3	Customising the Visualisation.	90
A.4	Canvas Navigation	91
A.5	Inspecting Individual Files	94
A.6	Export	94
B	Developer Guide	95
B.1	Development Stack	95
B.2	Project Structure.	95
B.2.1	Gizual API	95
B.2.2	Gizual App	96
B.2.3	Maestro.	96
B.2.4	Explorer	96
B.2.5	Renderer	97
B.3	Data Flow	97
B.4	Build and Deploy	97
	Bibliography	99

List of Figures

2.1	Nightingale’s Rose Diagram	4
2.2	Seesoft	6
2.3	Seesoft: Fix-on-Fix Mode	6
2.4	Spider Sense: Seesoft View	7
2.5	Spider Sense: Treemap View	8
2.6	RepoVis	8
2.7	GitHub Contributors Statistics: Gizual Repository	9
3.1	Rendering Pipeline: Flow Chart	16
3.2	Frontend Frameworks: Keyword Trend Analysis	21
3.3	NPM Trends: Bundlers	25
3.4	Responsive Website	28
3.5	Responsively App	29
4.1	Software Architecture of Gizual	33
4.2	Sequence Diagram for getFileContent()	34
4.3	Maestro Controller.	35
4.4	Visualisation Types	42
5.1	Gizual User Interface	44
5.2	Gizual User Interface: Design Theme in Figma	45
5.3	Gizual User Interface: POC1 (SS2022)	51
5.4	Gizual User Interface: POC2 (WS2022)	52
5.5	Gizual User Interface: Main Regions	53
6.1	The Gizual Canvas.	56
6.2	Traditional Blame Output	57
6.3	Line and Mosaic Visualisation Tiles	57
6.4	Masonry Canvas: SVG Export	61
6.5	Masonry Canvas: Minimap	62
6.6	Masonry Canvas: Legend	62
6.7	Masonry Canvas: Author Panel	62
7.1	Author Panel	66

8.1	Query Bar	68
8.2	Query Bar: Module Arrangement	68
8.3	Query Bar: Range by Date Module	68
8.4	Query Bar: Range by Revision Module	68
8.5	Query Bar: Time Module with Timeline	68
8.6	Query Bar: File Module	68
8.7	Query Bar: Vis Module.	69
8.8	Query Bar: Vis Type Dialog	70
8.9	Query Bar: Query Editor	74
8.10	Query Assistant	75
8.11	Query Bar: QB1	76
8.12	Query Bar: QB2	77
8.13	Query Bar: QB3	77
9.1	Timeline Component	80
9.2	File Tree Component	81
A.1	Gizual: Welcome Screen	88
A.2	Gizual User Interface: Main Regions	89
A.3	Query Bar: Module Arrangement	89
A.4	Time Module and Timeline	90
A.5	Visualisation Type Dialog	91
A.6	Canvas	92
A.7	Palette by Author Visualisation	93
A.8	File Tile	93
A.9	Source Editor	94
B.1	Software Architecture of Gizual	98

List of Tables

3.1	CSS Selector Specificity.	15
3.2	Parsed and Tokenised CSS	17
3.3	CSS Value Computation.	18
3.4	CSS Style Cascade.	18

List of Listings

3.1	HTML5: Minimal Example	12
3.2	HTML5: No Semantic Markup	13
3.3	HTML5: Valid Semantic Markup	13
3.4	CSS Syntax	15
3.5	CSS Syntax: Button	17
3.6	JavaScript Example	19
3.7	TypeScript vs. JavaScript	20
3.8	Example package.json file	26
4.1	MobX Example: ToDo App	37
4.2	MobX Example: ToDo App With React State	38
4.3	MobX Example: ToDo App With MobX State.	38
4.4	Maestro State Management.	40
4.5	Query Schema	41
5.1	Gizual's Workspace Structure	47
5.2	Button Component: SCSS File	48
5.3	Button Component: TypeScript File	49
5.4	DatePicker Component: TypeScript File	50
6.1	Base Renderer Interface	58
6.2	File Lines Renderer	59
7.1	ColorManager Class.	65
8.1	Query Bar Source Code Structure	71
8.2	BaseQueryModule Component	72
8.3	ModuleProvider Component	73
B.1	Gizual's Project Structure	96

Acknowledgements

This thesis would not exist without the continuous support of my friends and family, for which I am eternally grateful. I want to thank my supervisor, Prof. Keith Andrews, for providing his guidance and knowledge throughout the entire duration of this thesis.

Furthermore, I want to thank my dear friend and colleague, Stefan Schintler, for the seamless collaboration on our shared project implementation of Gizual, and for providing extra motivation and keeping me accountable during my studies and throughout this thesis.

Ich danke meinen Eltern für die bedingungslose Unterstützung während des gesamten Studiums, und dafür, dass sie immer für mich da sind.

Zuletzt möchte ich mich bei all meinen Freunden bedanken. Mein besonderer Dank gilt dabei Lukas, Eileen und Lisa, die mir während einer schwierigen Zeit in meinem Privatleben den nötigen Halt gegeben haben.

Andreas Steinkellner
Graz, Austria, 09 Dec 2024

Credits

I would like to thank the following individuals and organisations for permission to use their material:

- The thesis was written using Keith Andrews' skeleton thesis [Andrews 2021].
- Figure 2.2 was extracted from Eick et al. [1992] and is used under §42f.(1) of Austrian copyright law.
- Figure 2.3 was extracted from Eick et al. [1992] and is used under §42f.(1) of Austrian copyright law.
- Figure 2.4 was extracted from N. H. Reddy et al. [2015] and is used under §42f.(1) of Austrian copyright law.
- Figure 2.5 was extracted from N. H. Reddy et al. [2015] and is used under §42f.(1) of Austrian copyright law.
- Figure 2.6 was extracted from Feiner and Andrews [2018] and is used under §42f.(1) of Austrian copyright law.
- Figure 3.2 was obtained from Stack Overflow [2024b] and is used under §42f.(1) of Austrian copyright law.
- Figure 3.3 was obtained from Potter [2024] and is used under the terms of Creative Commons Attribution 4.0.
- Chapter 4 and Appendix B were written jointly with Stefan Schintler.

Chapter 1

Introduction

The web is a highly complex and dynamic environment. New technologies frequently transform the way the web is used on a daily basis. Motivated by the possibilities of the web as a platform, an increasing number of tools and applications try to reach a broader audience by harnessing the web not only as a platform to host static content, but as a place in which to create truly dynamic applications. With increasing support for native binaries through the use of WebAssembly [W3C 2024d], the web is becoming a strong choice for developers around the world due to its widespread availability and ease-of-access. Modern toolchains try to eliminate the performance drawbacks of crafting non-native applications, allowing performance-critical tools to be created for the web.

This general shift in computing architecture makes the web an attractive target for information visualisation. In information visualisation, the characteristics of the human visual perception system are harnessed to create intuitive visual presentations of abstract information spaces [Tominski and Schuhmann 2020; Ware 2021; Andrews 2024]. Providing such visualisations on the web leads to visual experiences that users can easily grasp within tools they already understand. This reduces the barrier to entry and provides a sense of familiarity, allowing users to focus on the content rather than the tool or platform. Newspapers recognised this phenomenon when they started adding interactive visualisations to their web pages, offering novel insights into datasets that were otherwise difficult to grasp [McGhee 2010].

This thesis presents Gizual, a novel way to visualise Git software repositories on the web, with an emphasis on ease-of-use, responsiveness, and high performance. Gizual is an open-source web application written in TypeScript and Rust using React, WebAssembly, and other tools to visualise arbitrary Git repositories. It uses the metaphor of hanging listings of the desired files in a repository on a wall. Users can interact with a scalable and interactive canvas to focus on specific files in a repository, or see the state of files within a branch on an interactive timeline element. To ensure that user data always remains private, all processing is done locally by leveraging the power of web workers and WebAssembly.

The source code of Gizual is open-source and available on GitHub [Schintler and Steinkellner 2024]. A deployed version can be accessed at gizual.com. Whereas this thesis [Steinkellner 2024] focuses on Gizual's user interface, the companion thesis by Stefan Schintler [Schintler 2024] focuses on the implementation of Gizual's data layer, which enables browser-based exploration of Git repositories.

The main contribution of this thesis lies in the implementation of a user interface and frontend core, which allows users to easily interface with the underlying data from the Git module. Standard HTML and CSS techniques were leveraged to provide an easily maintainable and extendable core, conforming to the latest accessibility standards. This thesis also introduces a custom implementation of a web worker that renders the user's desired visualisation either in an HTML canvas for high-performance interactivity, or as an SVG files for easy export in a freely scalable format. An interchangeable set of query modules assists users in choosing the appropriate visualisation for their desired set of files. An interactive timeline displays commits over time.

Chapters 2 and 3 of this thesis discuss related work in the areas of software visualisation and frontend web design. Chapter 2 introduces information visualisation and its connection to visualising software. Chapter 3 explains the current state of frontend web design, including tools, concepts and frameworks that are used to bring web applications to life.

The second part of this thesis, in Chapters 4 to 9, describes the original ideas and concepts applied in Gizual. Chapter 4 gives an overview of the Gizual architecture and explains the tight connection between Gizual's data layer and user interface. Chapter 5 presents the user interface of Gizual and provides insights into the various composite UI elements. Chapter 6 describes the interactive Canvas, the file tiles, and their usage within the visualisation. Chapter 7 presents the visual encodings provided in the visualisation. Chapter 8 explains the modular Query Bar, which handles user input. Chapter 9 provides a more detailed explanation of both the interactive Timeline and the custom file tree. Finally, Chapter 10 looks ahead to some potential future improvements and enhancements.

Chapter 2

Visualising Software

Software projects usually require a symbiosis of understanding both the domain of a problem and the technological intricacies of crafting an elegant solution. Developers generally need to be able to work with a wide variety of tools ranging from project management to source control utilities. Additionally, it is fundamental that software engineers can mentally envision the project as a whole.

Since the inception of software engineering, researchers have tried to pinpoint a measure for success in developing successful software products [Linberg 1999; Procaccino et al. 2022; Reel 1999], and generally conclude with a multitude of different skills that the team and the individuals need in order for projects to be completed in a timely manner. Often, problems with failing software development projects can be traced back to poor communication, planning, budgeting, or management issues [Kaur and Sengupta 2011; Bjørner 2009]. Proper and thoughtful management of software engineering workloads can be taxing on project managers, because they sometimes lack a deep technical understanding of the underlying source code. Software visualisation can increase mutual understanding of interdependencies, progress, metrics, and issues. This chapter introduces the field of information visualisation and demonstrates its application for software visualisation.

2.1 Information Visualisation

Information visualisation aims to provide visual support for understanding connections within complex datasets. It is built around the fundamental properties of human perception, which determine how people experience and understand the world around them [Eden 2005]. An early example was a chart created by Florence Nightingale in 1856, when she was a nursing administrator of a British army hospital. Nightingale conducted a study to understand how general sanitary conditions affected survival rates in military hospitals [Small 2013]. In order to better convey her results that improved sanitary conditions reduced the number of deaths due to preventable disease, she created a visualisation that would be more easily digestible to readers, shown in Figure 2.1.

Visualisations more generally can be traced back around 4,500 years, with visualisations by the Mesopotamians etched into clay tablets [Agosti et al. 2013]. In general, the field of information visualisation gained a huge boost in popularity through the means of electronic devices, simplifying both the creation and consumption of visual content. Newspapers, magazines, and ordinary websites alike are now using visualisation techniques to analyse and present their data in more cohesive ways. With the widespread adoption of mobile devices with powerful touch-screen navigation, additional layers of interactivity are being added to visualisations.

Manovich [2011] proposes that information visualisation only relies on two core principles: reduction, and the use of spatial variables. Reduction is the process of simplifying and reducing real-world objects, people, or ideas to simple geometric shapes, such as circles, rectangles or triangles, and by encoding

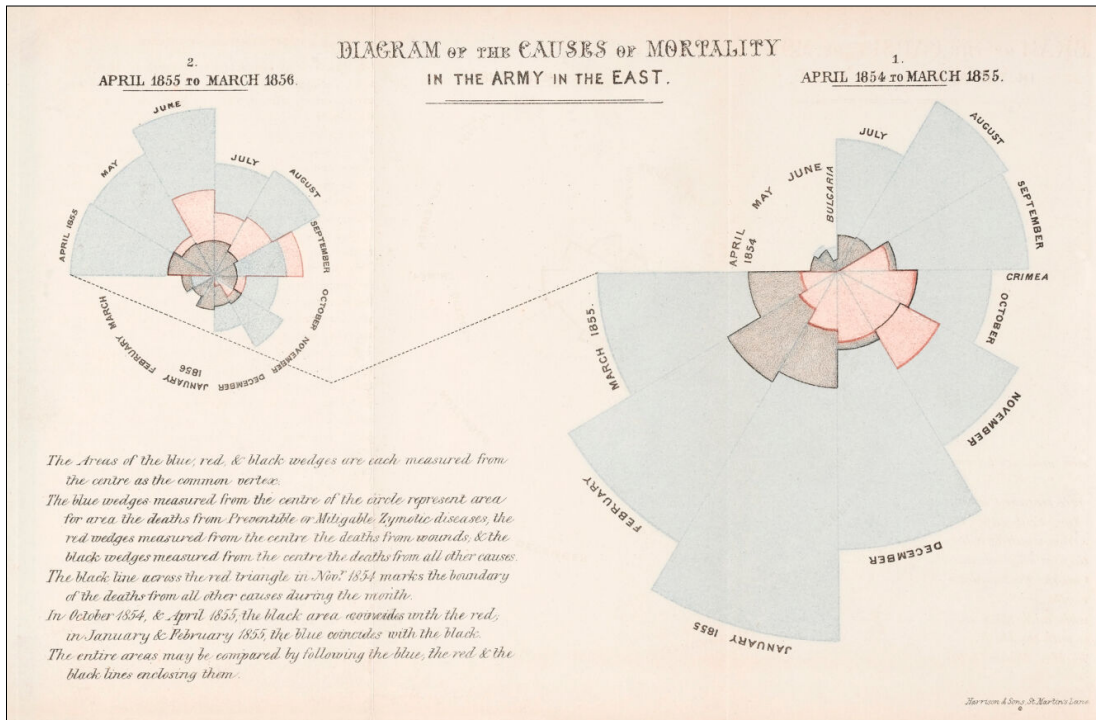


Figure 2.1: Florence Nightingale’s “rose diagram” visualisation of the reduction of deaths in military hospitals due to preventable diseases with increasing sanitary standards. [Image obtained from the Wellcome Collection and used under terms of the Public Domain Mark license.]

the connections between them as lines. Reduction can also be applied to the quantity of objects and connections visible at a single point in time. Effective information visualisation should also consider the limits of cognitive memory capacity.

The concept of spatial variables describes using properties such as position, size, shape, and curvature to represent differences between data points and relationships. This definition is coherent with the use of the Gestalt principles, and makes logical sense. More generally, the field of information visualisation is closely intertwined with design, sharing common important properties and ideas such as layout, alignment and visual coherence [Hu 2022]. Behrisch et al. [2018] conducted a study on 14 different subfields of information visualisation, and concluded that simple clutter reduction might not be sufficient to increase visualisation quality. Instead, they propose a set of techniques for each visualisation type that help facilitate pattern recognition. For all fields related to information visualisation, creating recognisable patterns seems to be a rewarding approach to making information accessible. An increasing focus has also been on the generation of highly sophisticated infographics, usually used by newspapers or magazines, which also heavily rely on these principles [Dur 2014]. Increasing amounts of available information generally lead to an increasing need for techniques to condense key points and make them easily digestible.

Unfortunately, it is out of scope for this thesis to describe the vast field of information visualisation and the rich history of contributions that shaped it in more detail here. There are numerous resources that serve as a starting point for further reading, such as Chen and Floridi [2013], Unwin [2020], IBM [2024], or Behrisch et al. [2018].

2.2 Software Visualisation

Software visualisation aims to provide deeper insight into the structure of code, its authors, or to provide possible hints about where to improve performance or optimise procedures. Petre and de Quincey [2006] defines the term software visualisation as visualisations which use visual representations to make software more visible. Knight and Munro [1999] propose the following definition: “Software visualisation is a discipline that makes use of various forms of imagery to provide insight and understanding and to reduce complexity of the existing software system under consideration.” A systematic literature review by Mattila et al. [2016] found that the main applications for software visualisation are in understanding software structure, behaviour, and evolution. According to their study, results are most often presented in graph or hierarchical structures.

The field itself gained popularity when computer programs were becoming increasingly capable, but their output were often just single-coloured rectangles on a screen. The main goal of software visualisation then was to transform these patterns of structural data into a web of interconnected nodes, which feels more natural for the brain to process [Price et al. 1992]. With modern computer hardware, visualisations can be much more elaborate and transformative to the consumption of information, since content is no longer constrained to fit into a two-dimensional plane. This can be especially useful when working on datasets which change over time. Previous research has demonstrated the usefulness of introducing multidimensionality to the output of software visualisation [Rainer 2010; Wetzel and Lanza 2007; Marcus et al. 2003], a technique that allows the exploration of data spanning multiple variables or dimensions.

In a meta-analysis by Merino et al. [2018], it was discovered that only 38% of proposed software visualisation tools include evaluation methodologies. The researchers identified that this lack of evaluation, or the vague definition of visualisation-related goals, is a major source of failure for these tools. Supported by this data, it appears to be highly beneficial to plan for evaluation of any given software visualisation tool during its design stage.

2.3 Visualising Software Repositories

Since there are usually a number of metrics and data points behind each line of code in a typical software repository, researchers have been looking for ways to visualise these additional metrics in addition to the textual content of the line of code.

Seesoft was the first visualisation tool for software repositories. It was created by Eick et al. [1992], and could analyse up to 50,000 lines of code simultaneously, combining various visualisation interactivity techniques, such as binning and brushing. Instead of only rendering a static visualisation of the data, a highly sophisticated graphical user interface allowed users to customise the output of the visualisation according to their current needs. Seesoft and its derivatives are the main inspiration for the ideas proposed in this thesis. As shown in Figure 2.2, Seesoft uses rectangular boxes to lay out files. Each of these file boxes contains a number of strips corresponding to lines within the file. The colour of each strip is based on the age of that specific line of code, with blue representing changes that were made a long time ago (cold), and red representing changes that were made more recently (hot). Special modes allowed different colouring algorithms to highlight other metrics of interest. The Fix-on-Fix mode, shown in Figure 2.3, indicates which parts of the code were issued with bug fixes multiple times in a row.

In addition to being used as a tool to gain a general overview of the state of a source code repository, Seesoft also contained functionality for detailed code profiling [Eick and Steffen 1992]. Further research was conducted to confirm the positive influence of graphically displaying textual information in source code repositories [Eick 1994].

Spider Sense was created to provide developers with a visualisation-backed toolkit for repository activity analysis, automated testing, and development activities [N. H. Reddy et al. 2015]. It features a Seesoft view, shown in Figure 2.4, where source code is displayed in a zoomed out overview and the lines

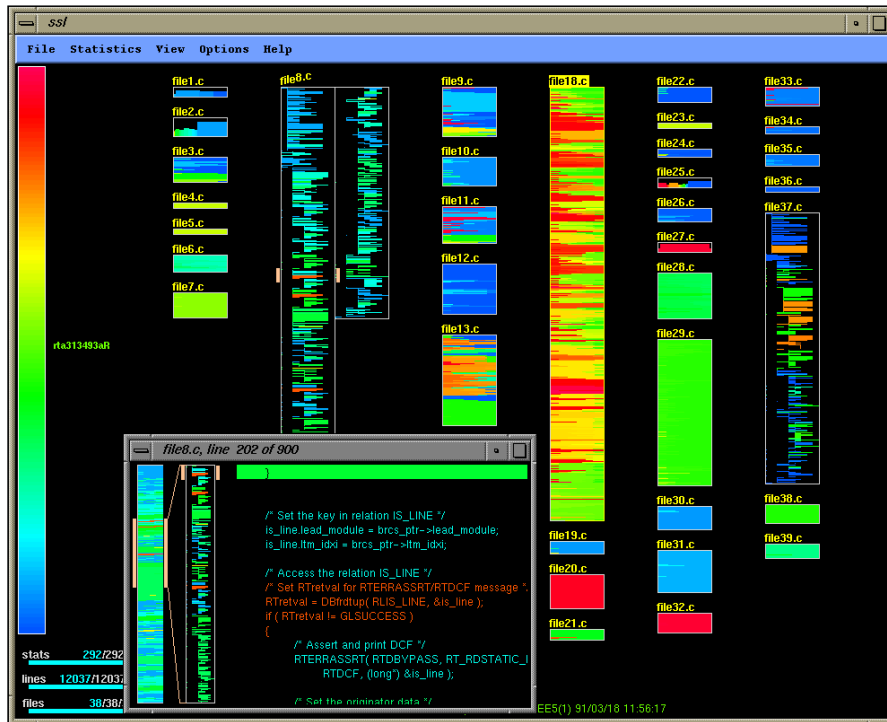


Figure 2.2: The Seesoft tool by Eick et al. [1992]. Files are laid out as rectangular boxes, and each coloured strip within a box represents a line of code. The colour of the strip is based on the age of the line of code. Blue strips represent changes that were made long ago (cold). Red strips represent more recent changes (hot). [Image extracted from Eick et al. [1992] and used under §42f.(1) of Austrian copyright law.]

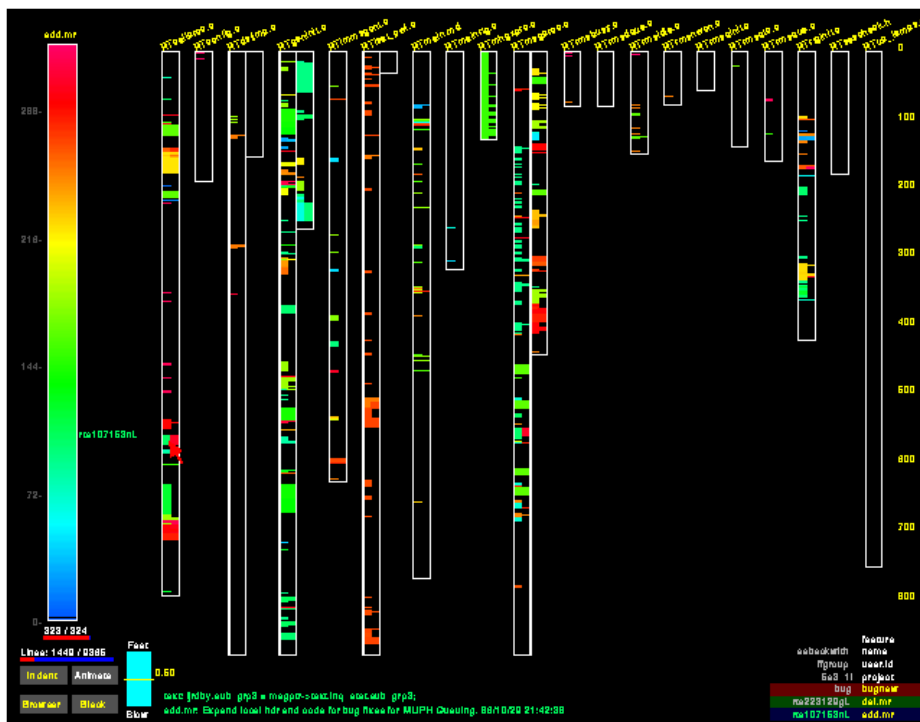


Figure 2.3: The Seesoft tool in Fix-on-Fix mode, proposed by Eick et al. [1992]. It highlights lines within files which were changed multiple times in a row. [Image extracted from Eick et al. [1992] and used under §42f.(1) of Austrian copyright law.]

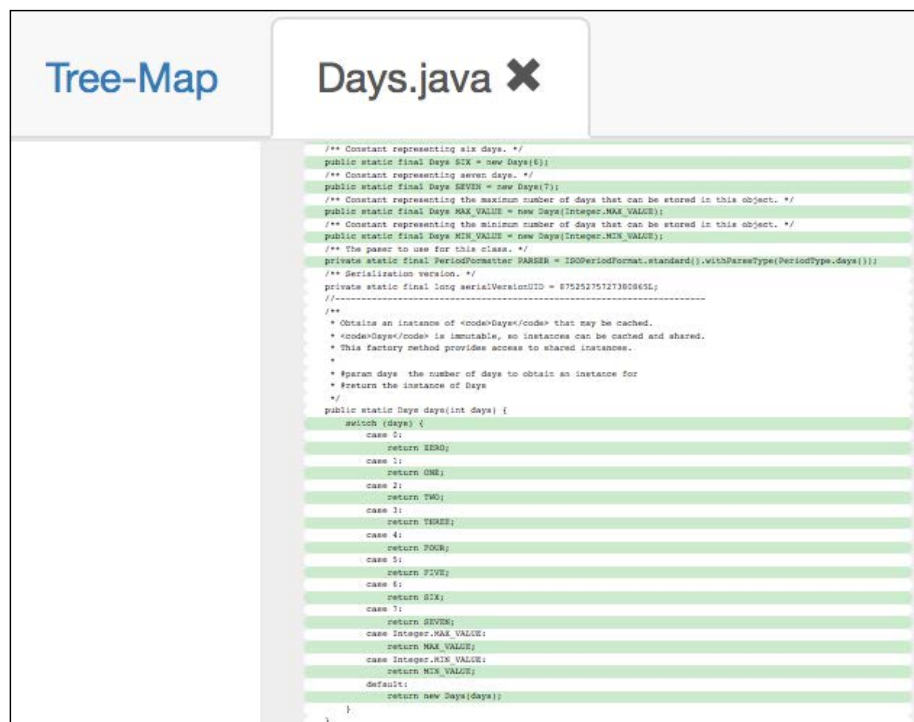


Figure 2.4: The Spider Sense Seesoft view of a Java source file, with lines of code corresponding to a particular metric colour-coded in green. [Image extracted from N. H. Reddy et al. [2015] and used under §42f.(1) of Austrian copyright law.]

of code corresponding to a particular metric are colour-coded in green. Spider Sense also has a treemap view, shown in Figure 2.5, which displays an overview of the entire folder hierarchy of a project.

The RepoVis project [Feiner and Andrews 2018] featured a Seesoft-like visualisation and combined it with full-text search for projects maintained in Git repositories. It can be seen in Figure 2.6. Sophisticated legend and timeline components provided a rich user experience. Tooltips provided additional utility through interactivity, and searching was available via full-text input or predefined tags. Similar to other related projects in this field, repository analytics are best performed on small to medium size repositories. Larger and more complex Git repositories impose a performance bottleneck on Git history parsing and full-text search.

Source produces an animated tree view of the change history within a Git repository [Caudwell 2010]. Active contributors float in proximity to the files they recently contributed to, and a dynamic camera reframes the scene to focus on the most relevant sections.

Nowadays, some statistics and metrics developed years ago have been integrated into popular source code repository management software, such as GitHub [GitHub 2024b], GitLab [GitLab 2024], or Bitbucket [Bitbucket 2024]. For example, Figure 2.7 shows the contributor statistics of the Gizual source code repository on GitHub. It gives an overview of the commits and total number of additions and deletions within the repository by each contributor. Statistics like these can help with the onboarding process of new developers, or give the team a general sense of code ownership within a project.

Sourcegraph [Sourcegraph 2024] takes this idea one step further and provides an entire code intelligence platform to their users. The idea behind the product is to provide a sophisticated tool to facilitate all sorts of analysis operations on a given source directory, with support for full text search, detailed custom-tailored statistics, and a powerful query language.

Although all of these programs and utilities provide an abundance of data, a study by Merino et al. [2016] found that developers are often still hesitant to use dedicated visualisation tools for their software

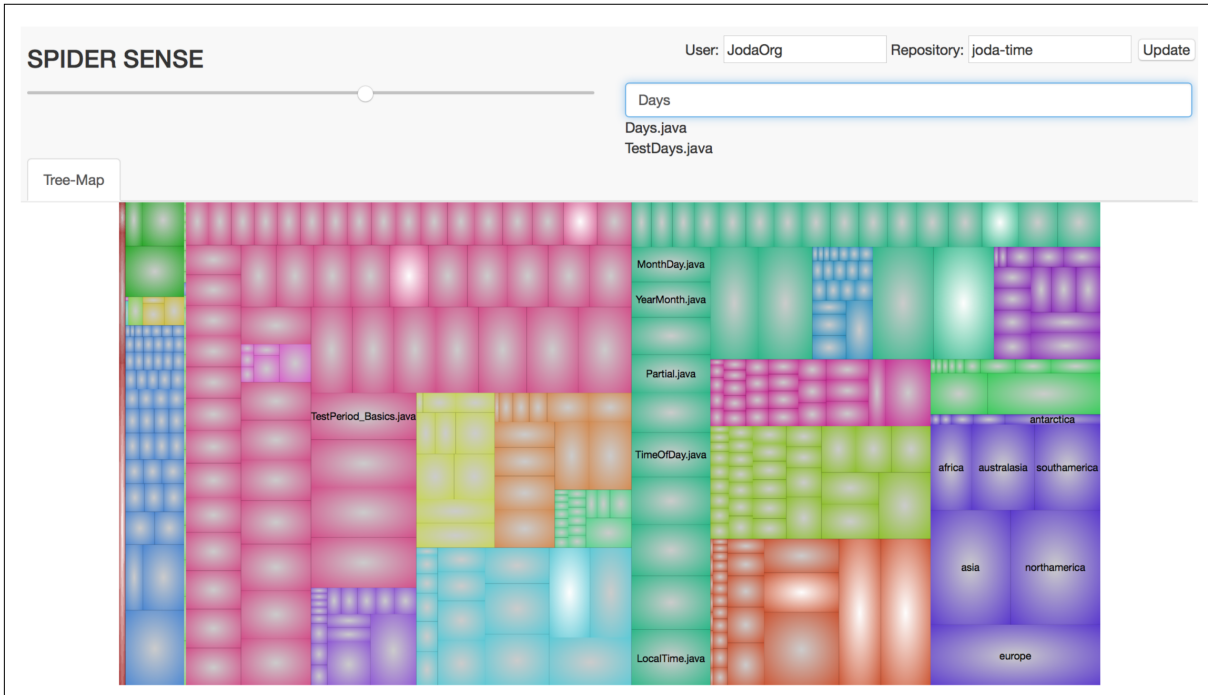


Figure 2.5: The Spider Sense treemap view for a set of Java files in a repository, displaying an overview of the entire folder hierarchy. [Image extracted from N. H. Reddy et al. [2015] and used under §42f.(1) of Austrian copyright law.]

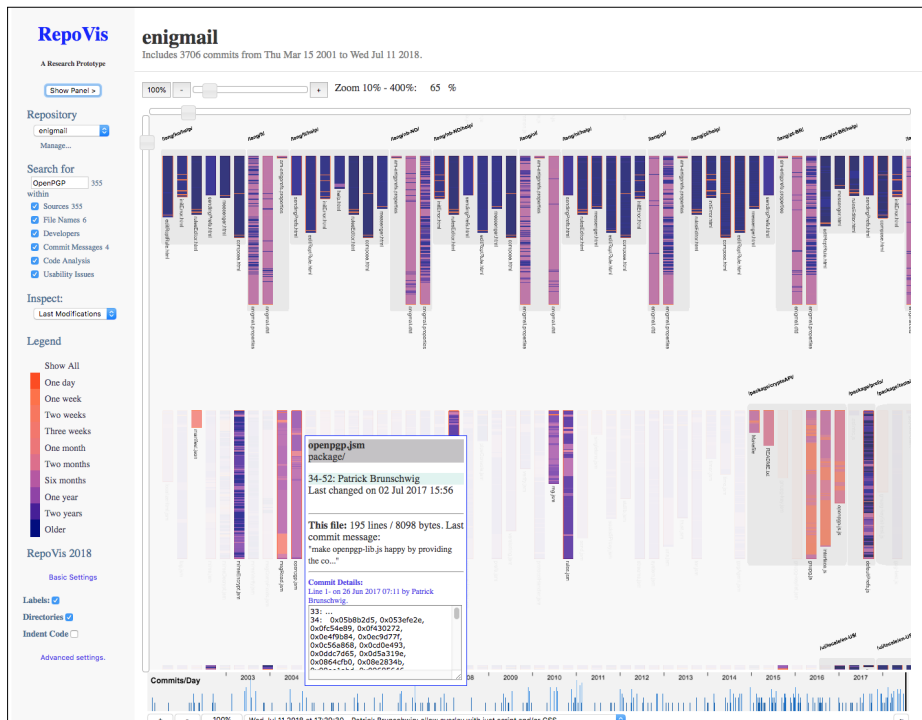


Figure 2.6: RepoVis provided a visual overview of files in a Git repository, overlaid with full-text search results. [Image extracted from Feiner and Andrews [2018] and used under §42f.(1) of Austrian copyright law.]

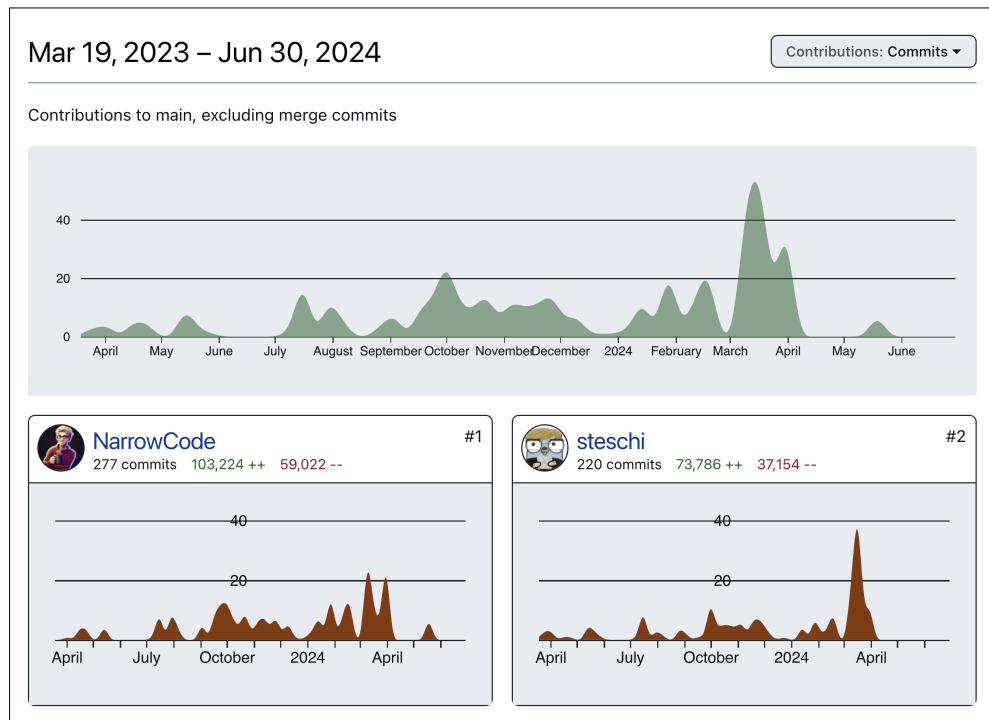


Figure 2.7: GitHub Contributors statistics for the Gizual repository, taken on 30 Jun 2024, providing an overview of commits and the total number of additions and deletions performed by each contributor. [Screenshot created by the author of this thesis.]

development needs. They found that often these visualisations tend to be solutions to problems which matter little to actual developers. Few tools cover the issues of implementation rationale and intent. Many visualisations instead focus on providing data about project dependencies and source code history, which may be of more interest to project managers than developers. According to this study, it would seem that focusing on visualisations that directly target software comprehension could provide engineers with a more immediate tangible benefit. However, models proposed for increased software comprehension, like Pacione et al. [2004] or Hawes et al. [2015] often fail to be adopted by software engineers [Duru et al. 2013], possibly due to the added complexity of their integration within the hectic development workflow.

Chapter 3

Frontend Web Design

Web-based tools have become increasingly popular in recent years. The web as a platform no longer just serves static web pages, it now contains a variety of content, ranging from typical media all the way to tools and applications for scientific or professional work. The line between web-based tools and native tools is blurring, as more and more native features become available in a web-based environment. Initiatives like the Web Standards Project (WaSP) [WaSP 2024] laid out the foundational work of standardising the web and the features users have come to expect. Additionally, official groups like the IETF [IETF 2024] and the World Wide Web Consortium [W3C 2024c] have been regulating the development and scope of new additions to the web for many decades. An impressive window into this standardisation process was provided during the introduction of the web for mobile devices, where standards had to be introduced quickly and efficiently [Ibrus 2013].

A web browser is the client software (frontend) used to access a web server (backend). Web servers can host both traditional websites and web applications. This chapter discusses frontend web development, with a focus on frameworks and developer tooling.

3.1 The Modern Web Browser

With the increasing prevalence of the web as an interactive application platform, web browsers have needed to incorporate more and more functionality into their repertoire. Nowadays, with WebAssembly [W3C 2024d] bringing low-level code execution to the browser, developers are able to access features from the operating system that were previously only available to developers of native applications.

Web browsers combine an optimised low-level rendering engine with a graphical user interface. The most frequently used web browser is Google Chrome with a market share of around 65%, followed by Safari (~18%), Edge (~5%), and Firefox (~3%) [StatCounter 2024], with many browsers often based on the same underlying browser engines. The browser's rendering engine is responsible for interpreting the source code and assets of a website and transforming them into visual and interactive experiences for users. This process is quite complex and usually consists of multiple steps. This thesis does not go into detail about individual rendering engines. Section 3.4 provides an overview of the rendering pipeline, but detailed discussion of low-level operating system code are omitted for brevity.

The modern web could not exist in its current form without the three pillars of web development: HTML defines the content and structure of the page, CSS is responsible for the visual appearance of page elements, and JavaScript adds behaviour and interactivity via code execution.

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="utf-8" />
5     <title>Title of the document</title>
6   </head>
7   <body>
8     <p>Content of the document.</p>
9   </body>
10 </html>
```

Listing 3.1: A minimal example of valid HTML5 syntax.

3.2 HTML

The Hypertext Markup Language (HTML) is the predominant language to structurally define documents and media elements for the use within a web browser. It is in continuous development through the World Wide Web Consortium (W3C) [W3C 2024c] and the Web Hypertext Application Technology Working Group (WHATWG) [WHATWG 2024b], and is now maintained as a “living standard” [WHATWG 2024a]. Listing 3.1 provides a minimal example of a valid HTML5 document. Structure is created by nesting elements within other elements. All elements have a starting tag and a corresponding closing tag, except for a small number of so-called void elements, which never have content or children. In such cases, the closing tag can be omitted, by prepending the end of the element tag with a forward slash. Some examples of valid HTML elements are the root `<html>` element, the `<head>` which usually contains document meta-data, and the `<body>`, which is usually filled with the content of the page, nested in smaller elements.

3.2.1 Semantic Markup

HTML markup can be written in a variety of ways. The most generic HTML elements are `<div>` and ``, which inherently do not convey any meaning beyond containing other elements. Creating a web page or application with only these types of elements is possible, but in the process of doing so, the benefits of the highly structured nature of HTML are lost. In order to create good and accessible web pages, developers are encouraged to use semantically correct elements for the types of content they are laying out on a page [MDN 2024n]. In total, there are currently 115 functional elements, with an additional 27 marked as deprecated or obsolete [MDN 2024i]. Listing 3.2 shows an example of a simple HTML5 page with no semantic meaning conveyed, whereas Listing 3.3 contains the same content, but follows the principle of good semantic markup.

In the example in Listing 3.3, the `<nav>` element is used to properly define the region which will contain the buttons to navigate from page to page. The `<button>` element is used instead of the generic `<div>` element, which automatically provides information to the user that the element can be interacted with. Finally, the `<section>` element creates a new block that can be properly parsed as a logically cohesive block, and the individual header and paragraph elements properly structure the content of the page. Whilst technically both of the provided examples can be styled to look exactly the same, the browser handles default styling automatically when provided with proper semantic markup. Section 3.3 goes into more detail about the styling of elements.

3.2.2 ARIA

An increasingly important topic for creating inclusive computing environments is the adherence to the Accessible Rich Internet Applications (ARIA) standard. ARIA defines rules and techniques to ensure

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="utf-8" />
5     <title>Example without semantic markup</title>
6   </head>
7   <body>
8     <div>
9       <div>Home</div>
10      <div>About</div>
11    </div>
12
13    <div>
14      <div>This is a heading.</div>
15      <div>This is a sub-heading.</div>
16      <div>This is the content of the page.</div>
17    </div>
18  </body>
19 </html>
```

Listing 3.2: A minimal example of valid HTML5 syntax with no semantic markup.

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="utf-8" />
5     <title>Example with semantic markup</title>
6   </head>
7   <body>
8     <nav>
9       <button>Home</button>
10      <button>About</button>
11    </nav>
12
13    <section>
14      <h1>This is a heading.</h1>
15      <h2>This is a sub-heading.</h2>
16      <p>This is the content of the page.</p>
17    </section>
18  </body>
19 </html>
```

Listing 3.3: A minimal example of valid HTML5 syntax with proper semantic markup. Elements like `<nav>`, `<button>`, `<section>`, and `<h1>` convey semantic meaning.

web pages and applications are equally accessible to people with various forms of disabilities, and is generally considered a supplement to writing semantically correct markup in HTML [MDN 2024a]. The World Wide Web Consortium (W3C) provides a detailed list of ARIA recommendations, ranging from very generic advice such as following colour contrast rules, all the way to specific elements that uniquely benefit people requiring screen readers [W3C 2024a]. For developers, it is often difficult to understand when to incorporate special ARIA properties in their markup, and to even know which HTML element or ARIA property to use in a given circumstance. Instead of providing additional HTML elements, the ARIA specification defines a set of additional HTML attributes and properties, which can be used to enhance existing HTML elements with additional accessibility information. Good web design with accessibility in mind can be more difficult to achieve initially, but will often also benefit people without any disabilities. The interested reader is referred to further literature, such as Pickering [2016].

Even years after the initial draft of the ARIA specifications, many websites still remain partially unusable for disabled people [Abuaddous et al. 2016]. Some websites even struggle with accessibility issues despite, or perhaps because they are, using ARIA attributes, but in a non-standard way [Matuzovic 2022; Martins and Duarte 2023]. Generally, it is preferable to only use ARIA attributes in situations where semantic HTML does not provide enough context for assistive technologies or people with disabilities to navigate the page properly.

3.3 CSS

HTML only provides content and semantic structure. It does not incorporate any style customisations for elements. In order to create unique and visually distinct user experiences, Cascading Style Sheets (CSS) are used to apply visual styling to semantic content on a page. To properly understand the intricacies of modern CSS, the concept of style application order must be introduced. This section gives an overview of basic CSS rules, the order in which they are typically applied, and scratches the surface of the underlying rendering pipeline to explain how these rules are transformed into pixels on a screen.

3.3.1 CSS Syntax

A typical CSS rule includes a target, which can be anything on the page, and a set of style rules to apply onto that target [MDN 2024g]. Targets are selected via CSS *selectors*, special instructions that allow developers to target one or many elements at once, and those selectors can be grouped together to select multiple targets at once. Listing 3.4 shows a typical way of styling an HTML `<button>` element, with three distinct ways to target the button. All three rules target the same button. However, they are not applied in document order (top to bottom), but rather in order of rule specificity.

Specificity is a required concept to enhance the selection of CSS targets. In short, the higher a CSS selector's specificity, the later it is applied in the rendering pipeline. This means it is applied after rules with a lower specificity, overwriting them. In the example in Listing 3.4, the selector `button` has the lowest specificity and is applied first, the `.button-class` selector is applied next, and the `#button-id` selector has the highest specificity and is applied last. Inline styles that are embedded directly into the element tag have an even higher specificity, and using the `!important` keyword creates the highest possible specificity in the chain. For each CSS rule, a specificity score is calculated by iterating through the selectors and assigning a numerical value based on the scores shown in Table 3.1.

3.3.2 CSS Origin Types

CSS can come from three distinct origin types: user-agent stylesheets, author stylesheets and user stylesheets [MDN 2024k]. User-agent stylesheets have the least priority and are generally provided by the browser for default styling. In some browsers, they can be customised or modified to achieve a different default behaviour that applies to all web pages viewed with that browser. In modern web design,

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <style>
5       button {
6         background-color: red;
7       }
8
9       #button-id {
10        background-color: green;
11      }
12
13      .button-class {
14        background-color: blue;
15      }
16    </style>
17  </head>
18  <body>
19    <button id="button-id" class="button-class">Click me!</button>
20  </body>
21 </html>

```

Listing 3.4: An example HTML page with a button and inline styles to apply to that button, in order to showcase both CSS rule syntax and target selector specificity. The rule with the highest specificity (in this case, the ID selector #button-id) wins.

Selector	Specificity Value
Universal selector (*)	0
Element or pseudo-element	1
Class, pseudo-class or attribute	10
Identifier	100
Inline style attribute	1,000
!important keyword	10,000

Table 3.1: The specificity of CSS selectors.

however, it is quite rare to see web applications which solely rely on user-agent stylesheets, since there are visual differences between different browsers. Pages which use proper semantic markup, as detailed in Section 3.2.1, generally benefit from the included user-agent stylesheets, because they provide a suitable default for these elements.

Author stylesheets come next in the priority hierarchy, and they are the most common source of modern CSS, written by web developers and designers to accompany their HTML markup. A commonly used practice by developers is to normalise the default CSS behaviour to a consistent default and then to apply custom styling on top of that.

User stylesheets are custom overrides that apply on a per-page basis and allow the end user of a web page to modify the style of elements on the page. User stylesheets are often used by web developers to test out design ideas before they are included in author stylesheets. The user stylesheets of a page can be modified with developer tooling, such as the built-in developer console in a standard web browser.

These three origin types define the basis of CSS rule application order. Stylesheets are normally parsed from top to bottom, and later styles overwrite the rules that came before, whilst regarding the specificity of the selectors, as explained in the previous section. Unfortunately, sometimes this application order

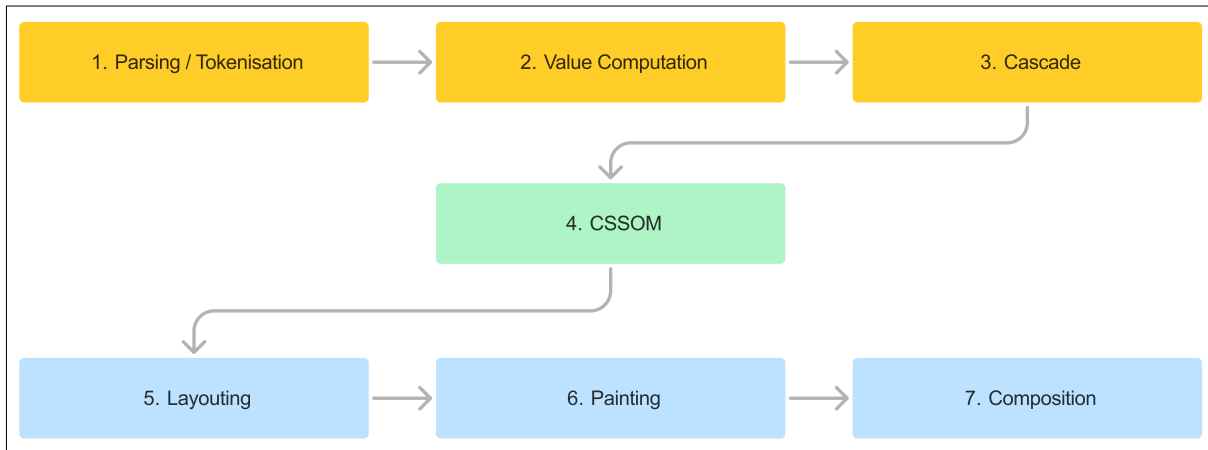


Figure 3.1: Simplified flow chart of the rendering pipeline in a modern web browser. The pipeline consists of seven steps, starting with the parsing and tokenisation of CSS, followed by value computation, cascade, CSSOM creation, layout calculations, painting, and finally composition. [Diagram created by the author of this thesis.]

is undesired or not sufficient to define the target style of an element. The `!important` modifier can be added to any valid CSS rule to hoist the application of that rule further down the pipeline, breaking out of the general cascade. Recently, modern browsers all gained support for a new feature in CSS called cascade layers [MDN 2024b], which allows for more fine-grained control of CSS application by providing user-defined layers which can be ordered in any way.

To properly understand the concept of origin types and specificity, the following section provides a brief introduction to the rendering pipeline and the way in which a web browser transforms CSS rules into applicable styles.

3.4 The Rendering Pipeline

Browsers have become very sophisticated and efficient at parsing and applying CSS through a streamlined rendering pipeline [Irish and Garsiel 2011; Whitworth 2018; Andrew 2020]. This rendering pipeline provides the functionality of converting code, usually HTML, CSS, and JavaScript into rendered pixels on a screen. Figure 3.1 illustrates the typical sequence of seven steps within the rendering pipeline in a simplified and condensed flow-chart.

3.4.1 Parsing and Tokenisation

Before the CSS can be properly interpreted, it is parsed and tokenised. Throughout this process, an internal representation of all individual CSS rules is created. During this step, all shorthand CSS properties that internally map to one or many properties at once are resolved. The CSS in Listing 3.5 is transformed to the internal representation displayed in Table 3.2.

3.4.2 Value Computation

The next step in the pipeline deals with value computation, and transforms all values into their standardised unit equivalent. Considering the example provided in Table 3.2, the value for the `font-size` property will be transformed from `2em` to the standardised value of `32px`. Additionally, all values containing CSS computations are resolved, floating-point values are rounded, and viewport units are translated into pixel values. Table 3.3 shows a few example values and their computed output after this step.


```
1 button {  
2   background: green;  
3   border: 1px solid red;  
4   font-size: 2em;  
5 }  
6  
7 .my-button {  
8   background-color: blue;  
9 }
```

Listing 3.5: Example CSS syntax for a simple button.

Selector	Property	Value
button	background-color	rgb(0,255,0)
button	border-width	1px
button	border-style	solid
button	border-color	rgb(255,0,0)
button	font-size	2em
.my-button	background-color	rgb(0,0,255)

Table 3.2: The parsed and tokenised CSS from the rules in Listing 3.5.

3.4.3 Cascade

The tokenised and parsed values are now fed into the cascade step, where the rules outlined in Sections 3.3.1 and 3.3.2 are applied to extend the internal representation with information about rule specificity and document order. After this step, the internal representation of the example defined in Listing 3.5 is shown in Table 3.4. In a real example, the styles defined in the user-agent stylesheet would need to be considered as well, but for simplicity, they are not shown in the table.

3.4.4 CSS Object Model (CSSOM)

After the cascade step, the browser knows exactly which styles apply to which element on the page in what order. All of this information is now stored in the CSS Object Model (CSSOM) [MDN 2024f]. This representation keeps track of the current values and allows developers to later manipulate each of these rules with JavaScript.

3.4.5 Layouting

After every element has an assigned style, the next step is layouting. For each element, boxes are created. Every element within a web page is placed in a box, and all boxes support common properties, such as margins, paddings, or content-fit properties. During the creation of the layout boxes, the CSS properties that apply to positioning are considered, and the final size of each box is reported back to the parent node in the DOM tree. As soon as floating elements are introduced, an additional measurement step is required to report minimum and maximum dimensions to the parent nodes as well.

In order to properly deal with blocks of content which can vary in size, the browser performs a special layout calculation to measure the size of all children in infinite space, and then calculate the minimum and maximum size from there [Whitworth 2018]. The flexible children are then fitted according to their respective properties, and the layout calculation can proceed with a positioned block.

Specified CSS Value	Computed Value
font-size: 1em	font-size: 16px
width: 50%	width: 50%
width: 123.456789px	width: 123.46px
height: calc(8px + 1rem)	height: 24px
height: 50vh	height: 800px

Table 3.3: Some examples of initial CSS values and their computed sizes after value computation.

Selector	Property	Value	Specificity	Origin	Document Order
button	background-color	rgb(0,255,0)	1	1	0
button	border-width	1px	1	1	1
button	border-style	solid	1	1	2
button	border-color	rgb(255,0,0)	1	1	3
button	font-size	2em	1	1	4
.my-button	background-color	rgb(0,0,255)	10	1	5

Table 3.4: The resulting internal representation from the CSS outlined in Listing 3.5 after the cascade step.

3.4.5.1 CSS Flexible Box Layout (Flexbox)

Some layouts are quite difficult to create with the traditional box method, which is why the CSS Flexible Box Layout with the `display: flex` property was introduced in 2018 [W3C 2018]. Its aim is to introduce a more flexible model for positioning elements of variable size. It facilitates this by providing simple ways of distributing size among children at the cost of omitting some document-centric properties that are present in `display: block` mode. The defining feature of this layout is its ability to define an item as flexible, which means it alters its width and height to fit the remaining space in the main dimension.

In this context, the main dimension is the axis in which the items are laid out, and it can either be `row` or `column`. The property that controls this behaviour is `flex-direction`. Additionally, the `flex-wrap` property controls if there should be any wrapping of elements, if the size of the container in the main dimension would overflow. Both the `flex-direction` and `flex-wrap` properties can be assigned simultaneously with the `flex-flow` shorthand. The `order` property controls the order in which flex items appear in the container. It defaults to document order, meaning that items that are defined first in the HTML are positioned first, but by providing an integer value to this property, the default ordering can be overwritten.

To control the flexibility of items, the `flex-grow` property determines how much the item will grow in relation to the other flex items. Similarly, the `flex-shrink` property determines how much the item will shrink in relation to the other flex items. Finally, the `flex-basis` property determines the main size of a flex item before free space distribution according to the other properties. The behaviour of this property is similar to setting width or height in the main dimension of the item.

3.4.5.2 CSS Grid Layout

CSS Grid layout was first proposed in 2011 [W3C 2011], as a better alternative to position elements without having to rely on HTML tables, but was only fully implemented by modern web browsers recently. At its core, CSS grid still resembles HTML tables, as it also works with columns and rows for managing items. By defining `grid-template-columns`, an element set to `display: grid` is assigned a fixed set of columns. Row behaviour can be adjusted by specifying `grid-template-rows`. Column or row information can also be automatically determined by the browser, if the `grid-auto-columns`, or

```
1 function updateDOM() {
2   var content = document.querySelector("#content");
3   if (!content) return;
4
5   content.setAttribute("style", "overflow: scroll");
6   content.innerHTML = "Hello World! I have been replaced by JavaScript content";
7 }
8
9 updateDOM();
```

Listing 3.6: Example JavaScript function which sets the content within a HTML div element, and applies a new style.

grid-auto-rows properties are used instead. Descendant elements can target a specific column or row inside the grid with the grid-column or grid-row properties. Both CSS Grid and CSS Flexbox provide developers with powerful tools to create responsive web applications that can dynamically adjust, based on the available viewport dimensions.

3.4.6 Painting

After layouting, every DOM node has a fixed size and can be positioned appropriately, but nothing has been painted on the screen yet. In the painting step, each element is rendered to the screen in the following order: background > border > content. After this step, each element is converted into a bitmap, which is ultimately the representation that the browser paints to the screen. The stacking context which defines object culling is determined through the DOM hierarchy and the additional CSS property z-index.

3.4.7 Composition

In the final composition step, layers are created and then rendered to the screen. Each layer contains a number of bitmaps that are already evaluated, positioned, and painted. Composition is commonly done on a separate compositor thread, which allows for high-performance animations, since entire layers can be shifted around with minimal computational effort before being rendered to the screen.

3.5 JavaScript

JavaScript (JS) is a scripting language for the web, used on billions of pages around the world, and is one of the most widely used programming languages. Its initial release version was created within a few days by Brendan Eich, in order to include dynamic code into otherwise static web pages, and was released in 1995 [Wirfs-Brock and Eich 2020]. Whilst the fundamentals of the language still remain partially intact, modern JavaScript has evolved over the past decades to incorporate more robust and error-safe methods to write dynamic code for web applications. Similar to HTML and CSS, features and aspects of the language were quickly standardised, with the driving force being the Ecma International organisation [Ecma 2024a]. Their standardised version of JavaScript is called ECMAScript [Ecma 2024b], and the two terms are commonly used synonymously.

The most common way to interface with a web page using JavaScript is by using the Document Object Model (DOM) API [MDN 2024h; W3C 2024e]. The DOM API provides an in-memory structural representation of the web page, which enables read and write access to the elements within it. The API is maintained by the World Wide Web Consortium (W3C) [W3C 2024c]. Listing 3.6 shows an example JavaScript function to manipulate an element of the DOM by modifying attributes and setting its inner HTML to custom content.

With the rise of web frameworks, JavaScript was often used quite heavily for general page layout or

```
1 interface User {
2   name: string;
3   age: number;
4 }
5
6 function printUserTyped(user: User) {
7   console.log("Name:", user.name);
8   console.log("Age:", user.age);
9   console.log("Address:", user.address); // Static type error
10 }
11
12 function printUserUntyped(user) {
13   console.log("Name:", user.name);
14   console.log("Age:", user.age);
15   console.log("Address:", user.address); // Runtime error, no type information
16 }
```

Listing 3.7: Comparison of a typed function in TypeScript to an untyped function in plain JavaScript. The typed function throws a static error at compile time, whereas the untyped function will throw an error at runtime.

interaction. This often lead to applications that were either slow to load or unusable without JavaScript. A common metric to measure the computational load on a given web page is to measure the time it takes the browser to do the Largest Contentful Paint (LCP) [W3C 2024b]. Roughly speaking, it measures the time it takes for the browser to finish painting a large portion of the page, which most likely corresponds to a meaningful document change for the user. Optimising for LCP requires minimising initial JavaScript loading times, which can be done by splitting large chunks of code into more manageable snippets that load on demand, or by reducing the computational complexity of the project. It is often said that the best way to handle JavaScript on a web page, is to only use it for things that absolutely require it, and handle all other interactions with HTML and CSS, if possible. This technique is commonly referred to as *progressive enhancement* [MDN 2024m], and it revolves around designing functional sites which still work when JavaScript is disabled.

3.6 TypeScript

TypeScript (TS) is an extension of the ECMAScript language, with the goal of adding static typing to improve the general developer experience and minimise errors [Microsoft 2024b]. It is developed and maintained by Microsoft and version 1.0 was first released in 2014 [Turner 2014]. The main benefit of using TypeScript as a web developer comes from the fact that many mistakes can be caught before runtime, which can speed up development significantly and create better code understanding [Bogner and Merkel 2022]. Additionally, it can help reduce general code smells, a metric to indicate negative software quality attributes [Alkharabsheh et al. 2018]. It also enables code editors and IDEs to implement sophisticated code completion.

Listing 3.7 highlights the key difference between typed and untyped functions that print data to the console. In the typed version of the function, accessing an undefined element within the user object is not allowed and results in a static compile-time error. In the untyped version, the browser will happily execute the code and then throw a runtime error instead.

According to a recent paper by Reid et al. [2023], 73.7% of the code snippets in NPM documentation contain errors, a number that can be reduced drastically to 25.1% after running the snippets through the TypeScript compiler. This demonstrates the power of static code analysis quite impressively, and is consistent with other literature on the topic [Louridas 2006; Bardas 2010].

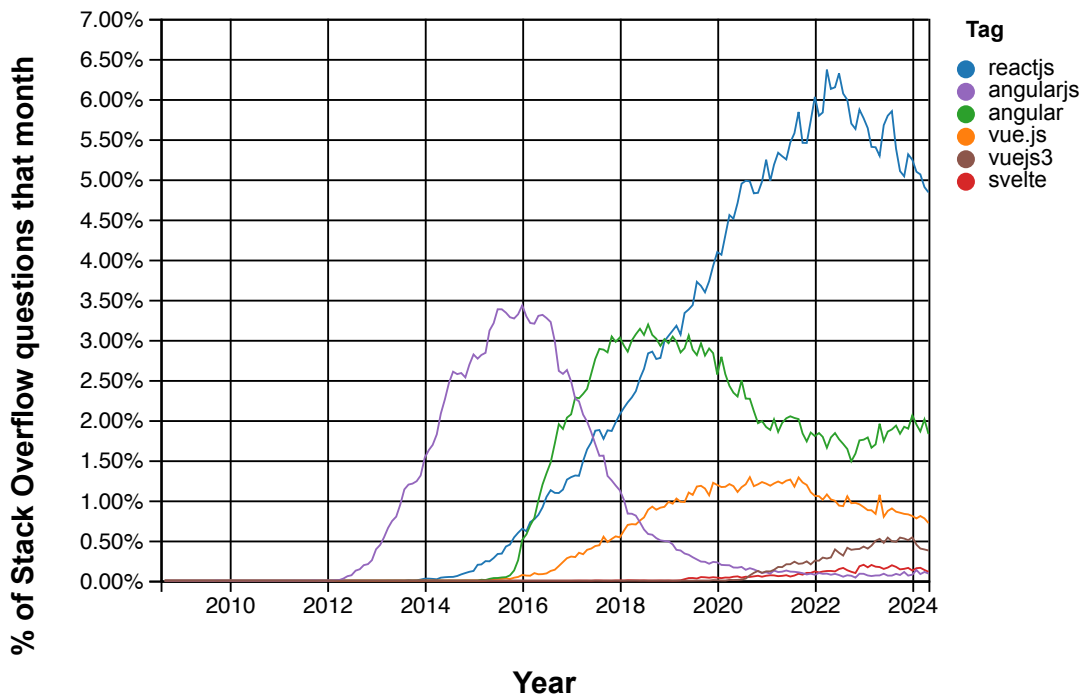


Figure 3.2: Comparison of keyword occurrence of the top frontend frameworks on Stack Overflow. [Image obtained from Stack Overflow [2024b] and used under §42f.(1) of Austrian copyright law.]

On the flip-side, whilst adding TypeScript to a project can significantly reduce code smells [Bogner and Merkel 2022], it adds complexity to the development process. Whilst JavaScript files can be embedded into an HTML document without further processing, TypeScript files require an additional transpilation step, which performs the static analysis and produces JavaScript files that can be embedded normally. Since this process is usually done on a per-file basis, it is common practice to use package managers such as npm [npm 2024], yarn [yarn 2024a], pnpm [pnpm Contributors 2024] or similar tools to manage project dependencies and handle static task execution. The process of transforming loosely coupled source code files into usable web pages will be explained in Section 3.8.1.

3.7 Frontend Frameworks

Frontend frameworks have markedly changed the typical flow of web development. They provide structural assistance to speed up the development of feature-rich applications using HTML, CSS, and JavaScript. According to a simple keyword trend search on StackOverflow [Stack Overflow 2024a], shown in Figure 3.2, React [React 2024a] appears to be the most popular frontend framework currently, followed by Angular [Angular 2024a], Vue [Vue 2024a], and Svelte [Svelte 2024] [Stack Overflow 2024b].

Generally, frameworks solve the problem of bidirectional state synchronisation of elements in the DOM, so that JavaScript can be used seamlessly to inject dynamic behaviour into otherwise static pages. All frameworks listed above follow different approaches in the way they are architected, and they each have unique benefits or drawbacks. In web development, choosing one framework over the another often comes down to personal preference, the fit into the existing technology stack, and the maintenance status and potential future releases. The following will introduce each framework individually, but will focus mostly on the intricacies of React, since it was the framework of chosen for implementing Gizual.

3.7.1 React

ReactJS [React 2024a] is a JavaScript framework developed by Meta [Meta 2024a], formerly Facebook. It was initially developed as a tool for creating the Facebook and Instagram news feeds, and was open-sourced in 2013. React was initially built as a way to simplify reactivity in the UI of web applications [Meta 2024f].

3.7.1.1 The Virtual DOM

At the core of the React architecture is the virtual DOM. It encapsulates manipulations on real DOM elements, allowing the scheduler to batch them together and update only necessary parts of the real DOM by using a diffing algorithm to compare differences [Aggarwal 2018; Meta 2024e]. In essence, this allows developers to just define the state they want the UI to be in, and React will optimise the updating of the actual underlying DOM structure. This process of synchronising states is called reconciliation [Meta 2024d]. During this step, the algorithm will parse through the tree of React elements after executing their render function. DOM elements that do not need to be discarded just have their attributes updated, instead of being replaced entirely. A custom key attribute is often used to assist the reconciliation process when dealing with dynamically created child elements. Assigning the same key attribute to a React element tells the reconciler to perform an attribute update instead of an expensive re-render.

3.7.1.2 Design Principles

The React framework is built around a set of core design principles that dictate the ways in which different problems can be solved within the framework [Meta 2024c]. Knowing these principles can help avert problematic or non-idiomatic usage of React's built-in utility functions and help with general understanding of the complex architecture. In brief, the seven core principles are:

- *Composition*: It is encouraged to rely on composition for components as much as possible. This means ensuring that adding state or lifecycle methods to a component does not create a destructive ripple effect within the entire codebase. In general, React developers try to encapsulate as much functionality as possible into separate and independent component blocks, which can be re-used within the entire codebase. This idea of composition is not inherently isolated to React developers though, and can be observed in many programming languages as a general paradigm [Alam and Kienzle 2012].
- *Common Abstraction*: The general approach of the React team is to provide as few utility functions as possible to avoid bloating the library with potentially irrelevant code. Exceptions to this rule are usually functions that would need to be re-written for every codebase, such as state and lifecycle methods.
- *Escape Hatches*: The React team diligently investigates new features in regard to their feasibility and compatibility with existing core concepts before committing to their integration. Still, sometimes there may be patterns that can become obsolete and deprecated. Usually, it is advised for developers to keep an eye on official blog posts and release documentation to gain an overview of upcoming features or breaking changes.
- *Interoperability*: One of the main concepts and strengths of React is its interoperability with non-React code in the same codebase. It is feasible, and even encouraged, to start by implementing some features as separate React components and gradually transition to React, rather than re-writing the entire codebase from scratch. React provides enough escape hatches to incorporate different kinds of models and support for other UI libraries to facilitate gradual adoption.
- *Scheduling*: Although the current idiomatic way of defining reusable components within the React framework is by expressing them as self-contained functions, a conceptual distinction to common

JavaScript functions needs to be made. The function that defines a functional component is not called directly on usage, but rather stored inside the internal model of the Virtual DOM. In practice, this allows the framework to delay function execution until it is necessary, while maintaining full control over the entire call stack. Delayed function execution can be used to optimise rendering for components that are not currently in view, or to add prioritisation to the rendering process. Whilst this technique is not currently used in the framework, the functional component approach allows for these gradual improvements to be implemented over time.

- *Developer Experience*: The React team maintains a strong connection to its core user-base and maintainers. Tools like the popular React DevTools extension help with easy adoption by providing easy-to-use insights into an otherwise rather complex rendering pipeline. Since other frameworks are emerging quite frequently, this connection to developers and users alike maintains a strong market position for the framework, which is crucial for attracting new users and maintainers.
- *Debugging*: Debugging, the process of isolating an error down to its origin, is simplified by the aforementioned React DevTools extension. Since it exposes the render tree, it is possible to traverse from the erroneous point up the tree to find the source of faulty components, component properties, or general mistakes in the rendering flow.

3.7.2 Angular

Angular [Angular 2024a] is a JavaScript framework developed by Google and a large community of open-source contributors. Its first stable release was in 2010, called AngularJS [GitHub 2024a]. As of January 2022, AngularJS is no longer maintained [Angular 2024b], and was superseded by Angular, with the latest current version being Angular v18, released in May 2024 [Gechev 2024]. In contrast to React, Angular aims to provide an out-of-the-box full stack framework experience, which supports two-way data binding, dependency injection, and a Model-View-Controller model, without the need for additional libraries or packages [Raval 2024].

Instead of relying on a Virtual DOM model like React does, Angular relies on an Incremental DOM [Angular 2024c]. The key difference between the Virtual DOM and the Incremental DOM models are the number of iterations each diffing pass requires and the space complexity of the implementation. In the Virtual DOM model, a virtual DOM tree is generated and then compared against the physical DOM, with changes being patched if necessary. In the Incremental DOM model, these virtual DOM nodes are parsed iteratively, and changes are applied as they are found [Ubl 2015]. This means that there is no need to store any data if there are no changes to a specific part of the DOM. This greatly reduces the memory usage of the implementation in comparison to strictly virtual DOM diffing algorithms.

3.7.3 Vue

Vue [Vue 2024a] is a JavaScript framework, created by Evan You and first released in 2014 [You 2014]. Its initial design goals were to extract the minimum required features from Angular and to create a lightweight alternative framework with them. Similar to React, Vue also relies on a Virtual DOM for rendering [Vue 2024b]. However, in contrast to React's runtime implementation of the Virtual DOM, Vue implements a hybrid-based Compiler-Informed Virtual DOM algorithm. It improves the performance of a runtime implementation by hoisting blocks of static markup out of the virtual DOM rendering pipeline, and specifically encoding rendering information into elements through a concept called Patch Flags.

3.7.4 Svelte

Svelte [Svelte 2024] is a JavaScript framework, created by Rich Harris and first released in 2016 [Harris 2016]. It was developed based on the idea that modern web applications, especially those that use giant frameworks like Angular or React, are inherently too bloated. Svelte aims to be a framework that provides

just the bare minimum needed functionality to create JavaScript modules that act as typical progressive enhancement to otherwise standard HTML and CSS. Svelte deliberately does not use a Virtual DOM implementation for performance reasons [Harris 2018].

3.7.5 Choosing React

With all of these frameworks explained briefly, the choice to use React for this project boils down to its long history of being interoperable with different JavaScript blocks in the same repository, and the fact that the framework has proven its stability over a number of years. For this thesis, React was deemed to be a safe choice, because countless libraries and utilities have been explicitly written to support the current version of React, and there is a large ecosystem of active web developers maintaining the framework and its related tooling.

When optimising for performance and developer experience in a large-scale web application, using the React built-in state management functionality is often not sufficient. There are numerous libraries and tools that integrate seamlessly with React and provide a more sophisticated developer experience and better tools to manage complex state derivations. The Gizual codebase relies on MobX [MobX 2024] as its state management library, which will be described in Section 4.3.

For completeness, it must be noted that, at the time of writing, the React team is working diligently to release React Version 19 [React 2024b], which will include a new compiler which can statically optimise React component memoisation [React 2024c]. This new compiler could solve many pain points within the React ecosystem, by providing an easier onboarding experience to new developers and reducing the mental complexity of optimising re-renders for all developers.

Nonetheless, the choice of React does have negative performance implications for some long-running tasks within Gizual’s rendering pipeline. In order to circumvent these issues, higher-order components within Gizual generally rely on MobX to handle state, and operate outside the typical idiomatic React lifecycle.

3.8 Building Web Applications

Traditionally, creating web applications did not require much additional software and dependencies. As the ecosystem grew larger, more sophisticated tooling was needed, and build tools and package managers emerged to simplify development and streamline the process of building web applications. Without these tools, the process of building web applications is much more cumbersome and time-consuming, which is why these tools are now considered standard [MDN 2024c].

3.8.1 Build Tools and Bundlers

Build tools are used to transform plain JavaScript or TypeScript code scattered across a multitude of folders into optimised chunks of minified code. This transformation step is necessary to allow developers to keep a clean directory structure during development, and still benefit of from optimised output at the end. The industry standard tool to use at the time of writing this thesis is Vite [Vite 2024a], which slowly took over the market share of the previously used tools Webpack [Webpack 2024], Parcel [Parcel 2024], and Rollup [Rollup 2024]. A general sense of the industry trend to adopt Vite can be observed in Figure 3.3, generated by npm trends [Potter 2024]. The data obtained from a simple download trend analysis is not very representative of real-world adoption of a tool, but the general trend can still be observed.

Vite uses esbuild [esbuild 2024a] to pre-bundle dependencies in an attempt to drastically improve hot module reload times. Instead of using JavaScript bundlers, esbuild is written in Go and is typically around 38 times faster than the quickest JavaScript alternative, according to benchmarks released on their own

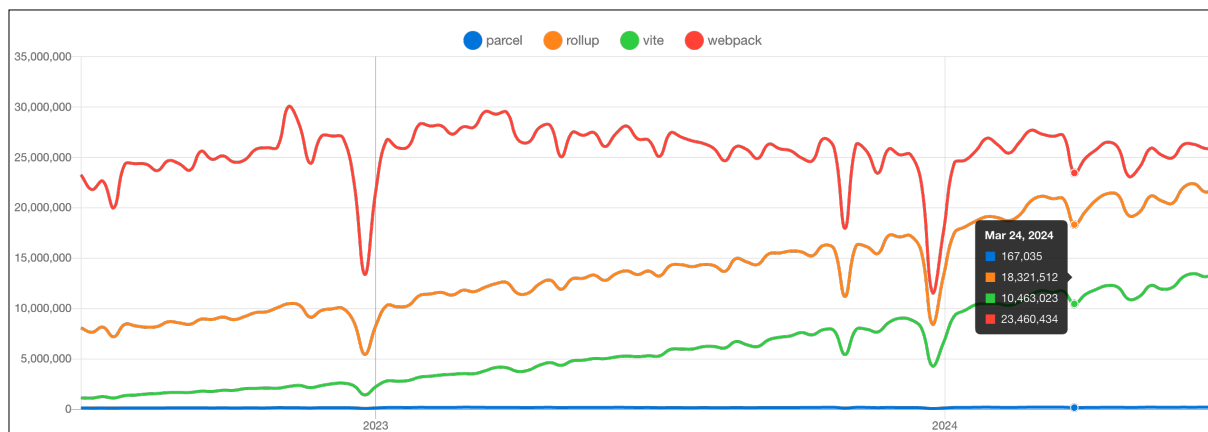


Figure 3.3: Graph comparing the weekly downloads of the most common web bundlers (Vite, Rollup, Parcel, and Webpack). [Image obtained from Potter [2024] and used under the terms of Creative Commons Attribution 4.0.]

website [esbuild 2024b]. Source code files are not pre-bundled, and instead are served as native ESM modules, which lets the browser take over the heavy computational load of bundling the files together. Whilst a detailed explanation of the historic development and current usage of JavaScript modules is beyond the scope of this thesis, detailed resources are available online for interested readers, such as [MDN 2024].

3.8.2 Package Managers

Usually, complex web applications consist of a multitude of packages and libraries. Package managers were developed to simplify the process of managing installed dependencies, scheduling updates, and interacting with the project as a whole. The basic functionality of a package manager is to download dependencies, check them for potential security vulnerabilities, place them in the correct directory within a project, add code that includes the package in the project, and handle package upgrade and deletion requests. The most popular package manager to date appears to be npm [npm 2024], but obtaining an accurate percentage-based comparison between package managers may not be feasible due to pre-bundled inclusions or different versions that are quite difficult to track [Wojciech 2023].

Generally, package managers manage versions of dependencies by creating a lock file, which contains the name of the package, its origin, the version of the package, and its location on the disk. The lock file must be in sync with the `package.json` file of a project, which is usually edited by the developer by hand, or by interfacing with the command-line interface of the package manager. Listing 3.8 shows a typical `package.json` file for a small React project, with Vite and its starter template being used as a build tool, and React with MobX included as dependencies.

In the case of Gizual, yarn [yarn 2024a] is used instead of npm, because a feature called yarn workspaces [yarn 2024b] helps with project structure in large mono-repositories. This feature allows developers to create multiple subprojects that share the same lock file, which in turn allows the package manager to download each dependency only once, whilst still keeping all subprojects separate.

```

1 {
2   "name": "snippets",
3   "private": true,
4   "version": "0.0.0",
5   "type": "module",
6   "scripts": {
7     "dev": "vite",
8     "build": "tsc -b && vite build",
9     "lint": "eslint . --ext ts,tsx --report-unused-disable-directives --max-warnings
10      0",
11     "preview": "vite preview"
12   },
13   "dependencies": {
14     "mobx": "^6.12.4",
15     "mobx-react-lite": "^4.0.7",
16     "react": "^18.3.1",
17     "react-dom": "^18.3.1"
18   },
19   "devDependencies": {
20     "@types/react": "^18.3.3",
21     "@types/react-dom": "^18.3.0",
22     "@typescript-eslint/eslint-plugin": "^7.13.1",
23     "@typescript-eslint/parser": "^7.13.1",
24     "@vitejs/plugin-react-swc": "^3.5.0",
25     "eslint": "^8.57.0",
26     "eslint-plugin-react-hooks": "^4.6.2",
27     "eslint-plugin-react-refresh": "^0.4.7",
28     "typescript": "^5.2.2",
29     "vite": "^5.3.1"
30   }
31 }

```

Listing 3.8: A simple `package.json` file used in a small React project. Vite is used as a build tool, and npm is used as a package manager.

3.9 Responsive Web Design

In a world where applications often need to scale to a plethora of differently sized screens, and be used on a multitude of devices, Responsive Web Design (RWD) emerged as a strategy to develop applications which behave consistently and expectedly in these circumstances. Users often use their smartphone instead of their desktop computers to browse the web [Adepu and Adler 2016], further increasing the importance of designing for multiple device sizes and capabilities at once. Unfortunately, devices like smartphones and tablets can come with hefty limitations regarding screen size, interactivity, and computing power [Buring et al. 2006; Müller et al. 2019].

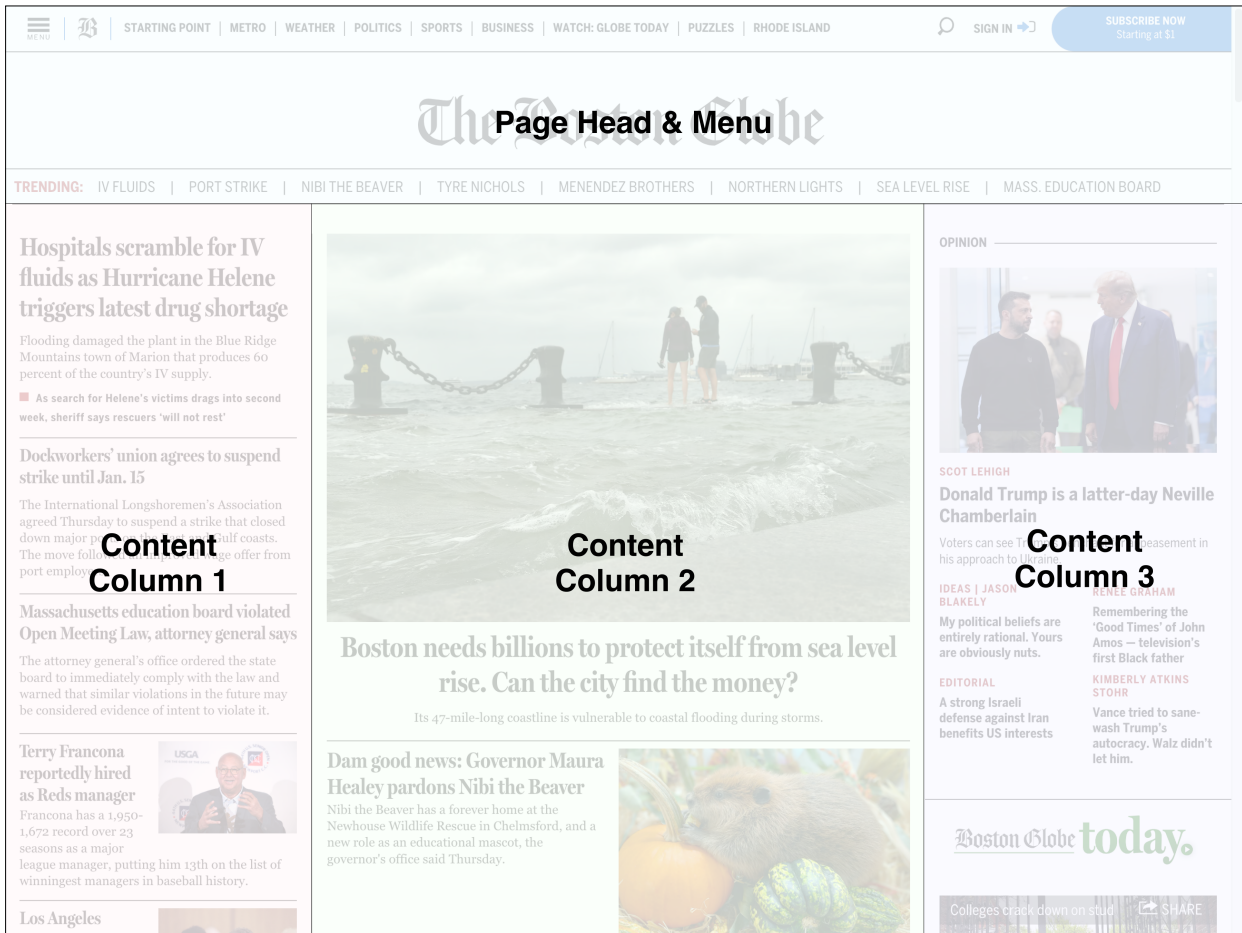
Utilising the unique benefits of touchscreen devices, for example, often requires additional thought regarding the implementation of hover states, which would be common with traditional mouse and keyboard navigation [Lee et al. 2012]. These differences not only create difficulties for web applications, but also for more general computing [Punchoojit and Hongwarittorn 2017]. Additionally, assumptions about specific device usage or interaction patterns can disregard certain accessibility concerns, so it is paramount to design modern web applications with a focus on responsive and adaptable user interfaces which can scale to any device and screen size.

Ideally, web applications should be designed in a way to natively support scaling up or down to the size of the device viewport. In practice, however, this often proves to be a challenge, since menu bars, icons, and buttons often take up considerable screen real estate. In the case of web applications that need to scale to mobile and desktop equally, it is often necessary to create two or three different versions of

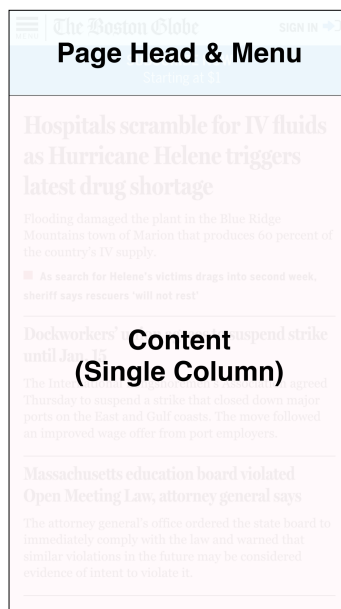
the user interface, and choose the most appropriate for the current viewport size. Optimising for mobile devices after implementing the desktop user interface often requires a radical approach of re-imagining an application, which is why the “mobile first” approach of developing web applications has gained popularity [Kim 2013]. The general trend of user interfaces is currently to reduce the user interface for phones to an absolute minimum, whilst maximising the amount of content that can be shown. On larger interfaces, more options are frequently exposed and easily accessible, while being tucked away in a sub-menu on a mobile device. Figure 3.4 shows the website of the Boston Globe on a wider screen (iPad Air in landscape orientation) and a narrower screen (iPhone SE). The layout adapts to the different screen widths.

Designing applications for multiple different screens and devices is not a trivial task, often requiring a plethora of different test devices or sophisticated tools. The most simple way to develop an application for different screen sizes is generally to use the browsers’ built-in developer tools to constrain the viewport size to the dimensions of the target device. With this approach, the general scaling and positioning of elements on the screen can be tested and confirmed. Tools like Responsively [ResponsivelyApp 2024] aim to solve the problem of constantly testing with differently sized viewports by providing an application that contains multiple viewport sizes, and syncs the UI state across them. This allows for easy testing across multiple differently sized viewports at once, without the overhead of manually adjusting the settings every time. Figure 3.5 shows the use of Responsively to test the Boston Globe website.

Virtual testing for different viewport sizes is great for observing responsive behaviour of simple web applications, but to test more device-specific features, or quirks with different browser on different devices, a more sophisticated approach must be used. Platforms like BrowserStack [BrowserStack 2024] and LambdaTest [LambdaTest 2024] provide powerful device farms, where real device testing can be done on a wide variety of different physical hardware.



(a) Wider screen.



(b) Narrower screen.

Figure 3.4: A responsive website adapts itself to the available display space. The website of the Boston Globe uses a three-column layout on wider screens where more space is available, and shrinks down to a one-column layout on narrower screens. [Both screenshots created by the author of this thesis.]

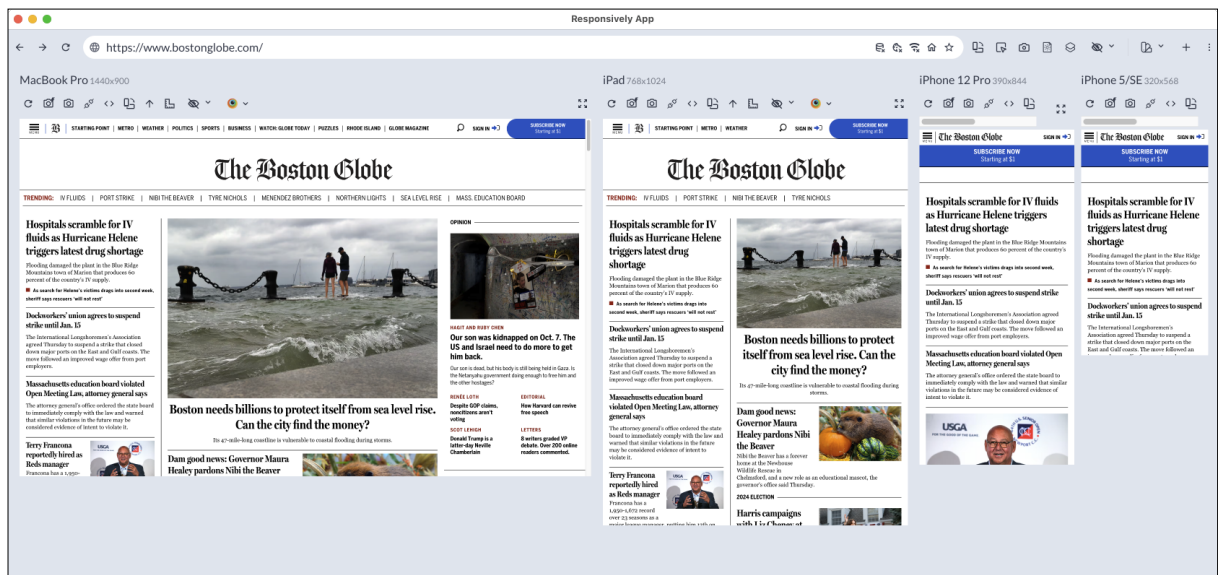


Figure 3.5: The Responsively App being used to simultaneously test the website of the Boston Globe across several different viewport sizes. [Screenshot created by the author of this thesis.]

Chapter 4

Gizual Architecture

“ I feel that it’s lovely when, as a user, you’re not aware of the complexity. ”

[Jonathan Ive; British and American designer, former Chief Design Officer at Apple.]

Gizual is a browser-based web application for visualising Git repositories. It is implemented with modern web technologies including HTML, SCSS, React, TypeScript, MobX, WebAssembly, Rust, web workers, and Node.js. All repository data processing is done in the browser, no external servers or databases are required. To bring all of these technologies together, a sophisticated software architecture was evolved.

This chapter was written jointly by Stefan Schintler and Andreas Steinkellner.

4.1 Architectural Requirements

Being a web application, Gizual can harness some unique benefits of the web as a platform. From a development standpoint, web libraries are available in abundance, reducing the complexity of building an application from scratch. For end users, the web generally provides a simple and easy-to-use platform for utilities that they can trust, since browser security policies are generally strict enough to protect from malicious actors. Building applications for the web does, however, come with a unique set of challenges, many of which are irrelevant for desktop applications. During the initial project ideation phase of Gizual, the team agreed upon some non-negotiable architectural requirements that would ensure that the application would match the perceived performance of a native tool.

4.1.1 Non-Blocking

By default, all JavaScript code within a web application is processed on the main thread. In addition to being responsible for layout and reflow operations, custom JavaScript execution is also part of the main thread’s responsibilities. For simple websites, this behaviour is usually not problematic, since JavaScript execution engines have become increasingly performant over the years. Users’ device performance is also steadily increasing, making performance optimisations less relevant for a typical web development workflow. Gizual, however, needs to process large quantities of data rapidly, usually in bursts that occur when users zoom or pan around on the visualisation canvas. Batching file manipulation and calculation operations into the same thread handling user interactions creates a performance choke point, leading to an unresponsive user interface or jittery animations. One of the agreed upon architectural requirements was to implement a solution which would reduce the computational complexity on the main thread by assigning pools of web workers [Surma 2019; WHATWG 2024c] to handle time-consuming tasks such as rendering and git exploration. With less work on the main thread, user interactions and animations would not be impeded by rendering tasks, and the application would always feel responsive.

4.1.2 Asynchronicity

In synchronous contexts, instructions are processed sequentially. In the context of web applications, one JavaScript thread runs through its set of instructions and processes them in order. Unfortunately, this can lead to long and inefficient wait times, drastically reducing the user experience and slowing down the application. Asynchronicity solves this problem by providing developers with functionality to let their code run outside the boundaries of the synchronous execution context. Writing asynchronous functions creates a chain of asynchronicity within the application, since any function which calls an asynchronous function must itself be asynchronous. This is why asynchronous functions are often referred to as being contagious [Haverbeke 2024b, Chapter 11]. For Gizual, doing all calculations sequentially in a synchronous manner would not be feasible. Asynchronicity was deemed necessary, despite the additional complexities this concept invariably introduces.

4.1.3 Parallel Execution

Complex functions within Gizual are generally expected to be executed in parallel. In order to achieve this, the project heavily relies on the use of asynchronous code and web workers. Without parallel execution of simultaneous file operations, the Git exploration performance would be too slow at providing repository analytics, slowing down the performance of the entire application. Additionally, parallel execution unlocks the power of modern multicore processor architecture, with different tasks being scheduled across many threads or even processor cores. However, the process of spawning and coordinating multiple threads (web workers) creates a slight computational overhead. Gizual relies on a defined limit of concurrent web workers, adhering to best practice recommendations in order to achieve maximum performance.

4.1.4 Separation of Concerns

The Gizual architecture consists of many modules of varying complexity, linked together via common bridges and interfaces. This loose coupling of segregated functions adheres to the general software engineering concept of *separation of concerns*. The term was initially used in a blog post by famous computer scientist Edsger W. Dijkstra [Dijkstra 1974], who used it to describe the process of clustering and ordering thoughts. In software engineering, the term is often used in conjunction with programming concepts like *model-driven development* [Kulkarni and S. Reddy 2003]. It describes the strategic separation of self-contained functions into smaller modules, which can be used across the entire application. This module-based approach invites the use of abstract interfaces, which greatly enhance modularity and simplify code maintenance.

Separation of concerns is a prevalent concept in web development too, where the three distinct technologies HTML, CSS, and JavaScript are used in tandem to create a single application. Modern frameworks, however, enable developers to forego this distinction, if desired. The Gizual architecture required more separation of concerns than a typical web application, since the architectural requirements defined in the previous three subsections already necessitate additional layers of abstraction and workload distribution.

4.2 Architectural Overview

To fulfil the architectural requirements, the project is split into six packages. A single, central management component, called Maestro, is responsible for managing the application, including global application state, user input, and data processing. Maestro controls the explorer and renderer pools of web workers, manages global state for the three UI Controllers and various higher-order UI Components, and populates an SQLite database with indexable metadata such as the names of authors (developers) for efficient querying. Figure 4.1 provides a structural overview of the interconnected packages and their communication pathways.

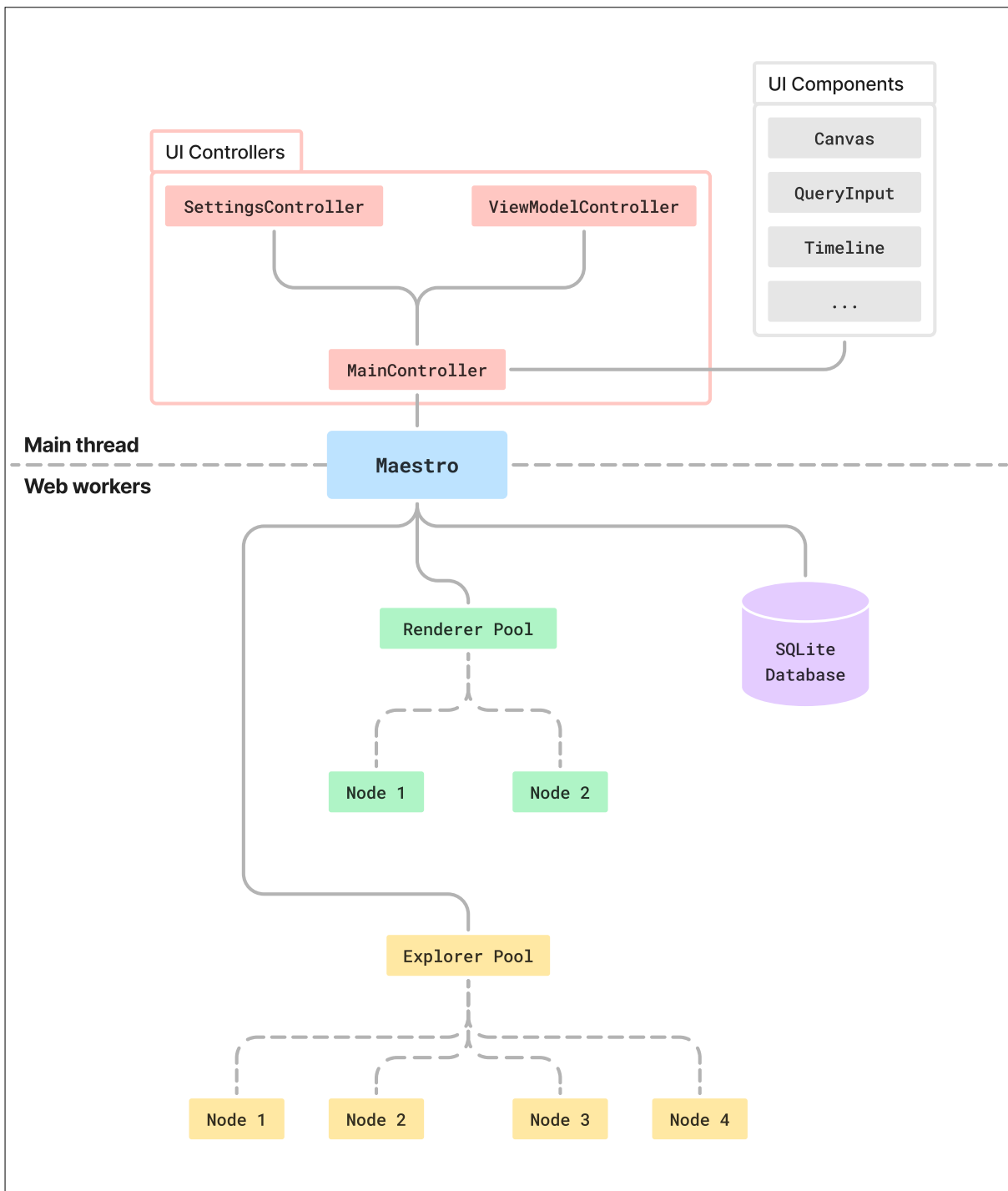


Figure 4.1: The software architecture of Gizual. UI controllers and UI components are instantiated in the main thread. The explorer pool, renderer pool, and SQLite database are executed asynchronously in web workers inside the browser. The Maestro provides a unified interface across the different realms. [Diagram created by the authors of this chapter.]

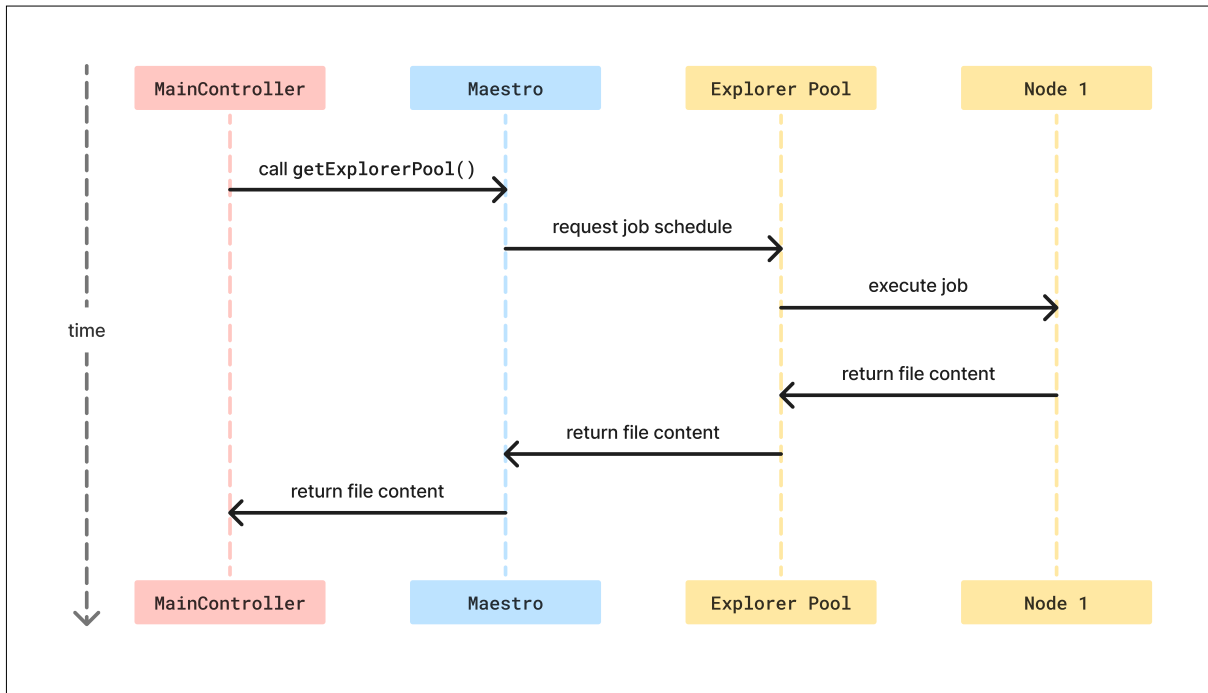


Figure 4.2: Sequence diagram for the `getFileContent` function. [Diagram created by the authors of this chapter.]

4.2.1 Explorer Pool

The explorer pool is a managed pool of web workers, responsible for handling data analysis and interactions with Git repositories. A set number of individual self-contained explorer workers are spawned by a central pool master. Maestro provides new workloads, which are distributed evenly across the worker pool, as illustrated in Figure 4.2. Each worker node is a continuously running process, awaiting a new job when idle. Every node is written in WebAssembly [WACG 2024] and leverages the open-source library `libgit2` [libgit2 2024] to interact with the Git repository chosen by the user.

4.2.2 Renderer Pool

The renderer pool is a collection of web workers which generate bitmaps for visualisation tiles. Distribution is handled through a central pool master, which schedules jobs and assigns them to idle workers. Each job results in a single bitmap image (tile) representing one file in the repository, which is used in the masonry grid on the main visualisation canvas.

4.2.3 SQLite Database

For performance reasons, an SQLite [SQLite 2024] database is used to store certain indexable metadata such as the names of authors (developers), so they can be efficiently queried. The SQLite database runs inside the web browser in its own web worker through WebAssembly. It is not persisted between sessions.

4.2.4 UI Controllers

Higher-Order UI Components use view models based on MobX [MobX 2024] to store state. These view models are centrally managed through the `ViewModelController`, which handles their assignment and keeps a reference for garbage collection. This controller is exposed to the Maestro via the `MainController`, responsible for grouping common UI functionality, variables, and controllers. User settings are handled

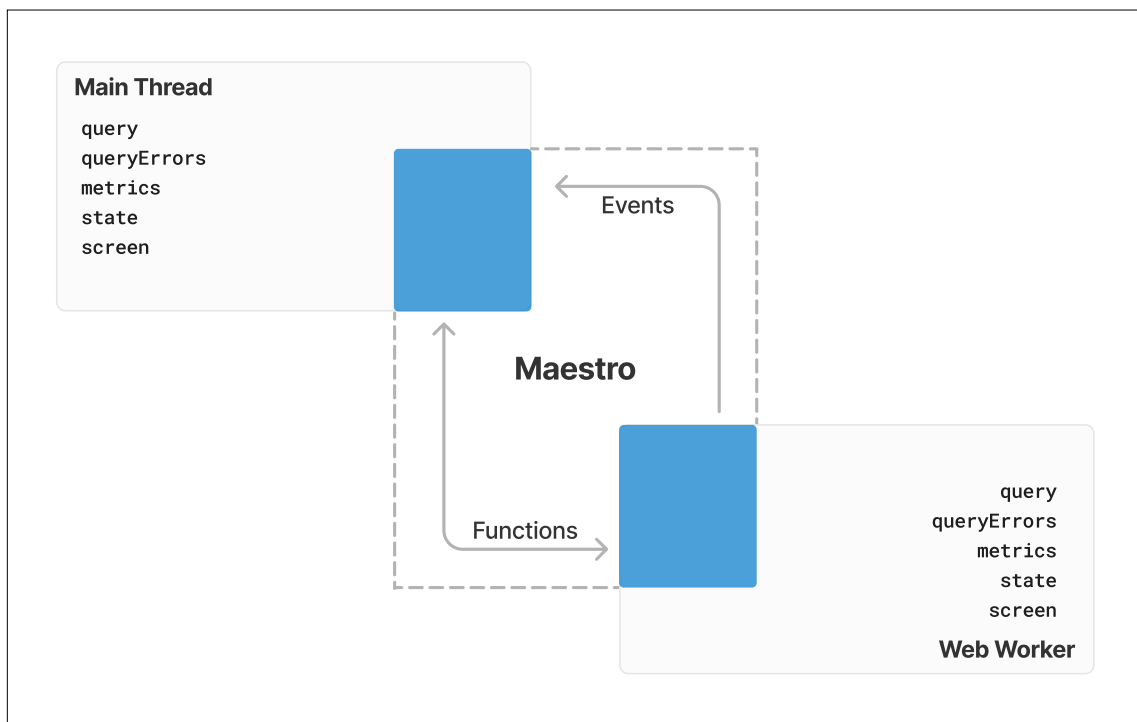


Figure 4.3: The Maestro controller has two parts: MaestroMain and MaestroWorker. MaestroMain is created in the main thread. MaestroWorker is invoked within a worker thread to manage longer-running asynchronous tasks. [Diagram created by the authors of this chapter.]

through the SettingsController, which exposes all settings as a single JavaScript object, and manages their state and persistence.

4.2.5 UI Components

The project has two distinct types of UI component: Primitive UI Components, which keep track of state internally in typical React fashion, and Higher-Order UI Components, which have a view model assigned to them to access global state. The three most notable Higher-Order UI Components are:

- Canvas: The Canvas component provides an interactive way to navigate the visualisation. Tiles are arranged in a masonry grid, and users can interact with the canvas through mouse and touch.
- QueryInput: The QueryInput component aggregates all supported query modules in a single component. It is directly attached to the Maestro and the query interface.
- Timeline: The Timeline component aggregates functionality to navigate time within a repository. It displays commits and the selected start and end dates in an interactive way.

4.2.6 Maestro

Gizual’s architecture is based upon a single main controller called Maestro. Maestro is responsible for managing the overall state of the application, including loading a repository, selection of files, and the visualisation chosen by the user. To accomplish this while reducing the chance of blocking the main thread, Maestro actually consists of two entities, MaestroMain and MaestroWorker, as shown in Figure 4.3.

The MaestroMain instance is created within the main thread and is responsible for handling user input and maintaining global state for the user interface in a MobX store, to allow UI components to subscribe to changes and update themselves accordingly. This allows the user interface to remain responsive, while

the application works in the background. The `MaestroWorker` instance is invoked within a worker thread and is responsible for handling longer-running asynchronous tasks. Global application state is synchronised between `Maestro` and `MaestroWorker` using a custom, event-based protocol.

4.3 State Management

Managing state in Gizual across different realms requires additional tooling compared to a standard single-threaded web application. In order to synchronise state globally, Gizual uses a set of functions and events across different threads. The underlying state management between realms is handled by `Maestro`. It uses `MobX` to store a reactive representation of the global application state in the main thread. UI controllers and components subscribe to this state representation and update accordingly.

4.3.1 Introduction to MobX

`MobX` [MobX 2024] is a state management library that aims to simplify the process of managing UI state. It does so by providing an easy way for components to automatically derive their state from the general application state. In the context of `React`, it ensures that only the components that are affected by a state update are re-rendered, providing a substantial boost to perceived speed and performance. `MobX` solves state management issues through *state*, *actions*, and *reactions*, which are all part of the library.

Actions in `MobX` are functions that modify state. Executing an action will task `MobX` with investigating which state variable was changed during this action, such that it can propagate a state update to all functions that “observe” that piece of state. Through that interface, components can all share a common similar state, stored in a `MobX` class, and re-render events will only propagate to those components that require them.

4.3.2 State Management in React

Typically, the proposed way of managing state within `React` is to propagate state updates down the rendering hierarchy. This goes hand in hand with the paradigm of “lifting state up”, a term used to describe the process of moving the assignment of state to the parent component if a sibling component might also need access. For smaller applications, this process is quite intuitive and easy to grasp. As applications grow in scale, business requirements often create a need to extend component state to encapsulate more design variants or general functionality. At some point, managing state manually can become quite cumbersome and error-prone.

The traditional way of managing `React` state uses the following syntax:

```
[state, setState] = React.useState(undefined)
```

As soon as a variable is stateful in the context of a `React` component, every usage of this state variable is tracked by `React` automatically, and changing the state by calling the `setState` function re-renders the component. Sometimes, state updates need to be conditional and based on other variables, like function properties or calls to an API. This is usually handled with the `useEffect` function, which allows developers to specify an optional set of tracked variables that trigger a function execution. One of the more common problems that `React` developers frequently face is unnecessary component re-rendering, usually triggered by non-idiomatic or erroneous usage of the `useEffect` function, which trigger unnecessary state updates.

To illustrate this concept, the example app in Listing 4.1 uses two implementations of a simple `ToDo` list, called in lines 21 and 22. The first function component, `ToDoReact`, shown in Listing 4.2 uses plain `React` code and state management logic, while the second implementation, `ToDoMobX` shown in Listing 4.3, uses `MobX` to manage state. The button that adds a `ToDo` to the list is deliberately placed as a sibling to the components that display the items. With simple `React` state management code, it would be an expected

```
1 import React from "react";
2 import "./App.css";
3 import { TodoMobX, TodoStore } from "./TodoMobx";
4 import { TodoReact } from "./TodoReact";
5
6 const store = new TodoStore();
7
8 function App() {
9   const [todos, setTodos] = React.useState([
10     "Example Todo 1",
11     "Example Todo 2",
12   ]);
13
14   const addTodo = () => {
15     store.addTodo();
16     setTodos([...todos, `Example Todo ${todos.length + 1}`]);
17   };
18
19   return (
20     <main>
21       <TodoReact todos={todos} />
22       <TodoMobX store={store} />
23       <button onClick={addTodo}>Add Todo</button>
24     </main>
25   );
26 }
27
28 export default App;
```

Listing 4.1: A simple React application with two alternative components to manage a ToDo list. The first component, `TodoReact` called in line 21, uses vanilla React to store state. It is shown in Listing 4.2. The second component, `TodoMobX` called in line 22, uses MobX and is shown in Listing 4.3.

procedure to lift state up, which means transferring it to the `App` component. This would allow for both the list and the button to access the state value and update it accordingly. Doing so works perfectly fine, but adds an unnecessary re-render for all components within the `App` component as soon as the state changes. The MobX implementation, on the other hand, only re-renders the components that are affected by the state change, which is only the `TodoList` component in this example.

The key strength of MobX lies in the automatic tracking of state changes by observing access to the values of *observable* properties within *tracked functions*. Instead of tracking the specific *values* of observable objects, MobX tracks the *property access*. In principle, this means that once the proper decorator annotations are placed to differentiate *observable*, *action* and *computed* objects appropriately, MobX automatically handles state propagation. In practice, this is enough for most application code, but sometimes reactivity needs to be defined on a more granular basis.

4.3.3 Advanced Reactivity Within MobX

When automatic reactivity tracking is not sufficient for the specific needs of an application, MobX provides custom functions to attach reactions to specific observable values. With the `autorun` function, it is possible to define an encapsulated function block, in which all used *observable* values are tracked automatically. Additionally, an optional delay can be specified to introduce a wait time before the initial function execution.

For more granularity, a custom reaction function can be used to define the *observable* objects that

```

1 function TodoReact({ todos }: { todos: string[] }) {
2   return (
3     <div className="container">
4       {todos.map((t) => (
5         <div>{t}</div>
6       ))}
7     </div>
8   );
9 }
10
11 export { TodoReact };

```

Listing 4.2: A simple ToDo list implemented with basic React state management.

```

1 import { makeAutoObservable } from "mobx";
2 import { observer } from "mobx-react-lite";
3
4 class TodoStore {
5   _todos: string[]; // Internal representation of state.
6
7   constructor() {
8     this._todos = ["Example Todo 1", "Example Todo 2"];
9
10    /* Automatically let MobX track variables as state, functions as actions. */
11    makeAutoObservable(this, undefined, { autoBind: true });
12  }
13
14  addTodo() {
15    this._todos.push(`Example Todo ${this._todos.length + 1}`);
16  }
17
18  get todos() {
19    return this._todos;
20  }
21 }
22
23 const TodoMobX = observer(({ store }: { store: TodoStore }) => {
24   return (
25     <div className="container">
26       {store.todos.map((t) => (
27         <div>{t}</div>
28       ))}
29     </div>
30   );
31 });
32
33 export { TodoStore };
34 export { TodoMobX };

```

Listing 4.3: A simple ToDo list implemented with React and MobX. An additional store is introduced to efficiently manage state updates and actions.

MobX needs to track manually. A supplied callback function will then automatically be executed once any of the supplied dependencies update. This behaviour is similar to the functionality React provides with its `useEffect` hook.

Regardless of using `autorun` or `reaction`, both functions exist outside the scope of MobX's automatic garbage collection and cleanup pipeline, so they need to be disposed of manually when no longer needed. Within Gizual, both `autorun` and `reaction` are used across parts of the application to fine-tune the reactivity of components. The architectural split between View and ViewModel, explained in Section 4.2, often requires a granular approach to state management to reduce unnecessary re-renders whilst keeping state consistent and synchronous across the entire application.

4.3.4 MobX Usage in Gizual

In Gizual, Maestro encapsulates the application state into specific objects, called observable boxes in MobX terminology. UI components can individually subscribe to specific parts of state they depend on by observing these boxes. Listing 4.4 shows a truncated example of the observable boxes used by Maestro.

4.4 Query Interface

Users can customise the output of the visualisation through a defined set of input parameters. These parameters are grouped into a single query object. This object is a crucial part of the application state, and is encapsulated into a separate reactive box within Maestro.

The query interface allows communication between the user interface and the data layer. It consists of a strict data schema to represent the user's input and desired output. To make this data accessible to the main thread and other realms, the query object is JSON-serialisable. It satisfies the `QuerySchema` interface shown in Listing 4.5. The query is configured using three scopes: `commit-range`, `files`, and `visualisation`.

4.4.1 Scope `commit-range`

The number of files inside a repository usually varies over time. Each commit can add or remove files, thereby changing the available file list. To specify the desired range of commits, two options are available:

- `branch` and `rangeByDate`: Select commits from a specific branch within a given time range.
- `rangeByRev`: Select commits between two Git revisions (e.g. a commit id or a tag).

By specifying either of these options, a start commit and an end commit are selected. The end commit defines the available file list, which is then narrowed down by the `files` scope. The start commit saves time by limiting the backward extent of the Git blame operation on each file.

4.4.2 Scope `files`

The `files` scope defines the files the user would like to see. Gizual supports the following file selection types:

- `path:string`: Select files by their path using a glob pattern.
- `path:string[]`: Select files by their distinct paths using a file picker.
- `changedInRev:string`: Select files which were changed in the specified Git revision (e.g. a commit id or a tag).

```
1 import { observable } from "mobx";
2
3 class Maestro {
4   globalState = observable.box<State>(
5     {
6       screen: "welcome" as "welcome" | "initial-load" | "main",
7       queryValid: false,
8       repoLoaded: false,
9       authorsLoaded: false,
10      commitsIndexed: false,
11      filesIndexed: false,
12      error: undefined,
13      currentBranch: "",
14      tags: [],
15      branches: [],
16      remotes: [],
17      // ... truncated
18    },
19    { deep: false }
20  );
21
22  metrics = observable.box<Metrics>(
23    {
24      numExplorerJobs: 0,
25      numExplorerWorkersBusy: 0,
26      numExplorerWorkersTotal: 0,
27      numRendererJobs: 0,
28      numRendererWorkersBusy: 0,
29      numRendererWorkers: 0,
30      numSelectedFiles: 0,
31    },
32    { deep: false }
33  );
34
35  query = observable.box<Query>(
36    { branch: "", type: "file-lines" },
37    { deep: false }
38  );
39
40  queryErrors = observable.box<QueryError[] | undefined>(undefined, {
41    deep: false,
42  });
43
44  // ... truncated
45 }
```

Listing 4.4: Partial implementation of the global state management defined in Maestro. State is encapsulated into observable boxes, which leverage the MobX reactivity ecosystem.


```

1 export interface QuerySchema {
2   branch: string;
3   time:
4     | {
5       rangeByDate: [string, string] | string;
6     }
7     | {
8       rangeByRev: [string, string] | string;
9     };
10  files:
11    | {
12      path: string[] | string;
13    }
14    | {
15      lastEditedBy: string[] | string;
16    }
17    | {
18      changedInRev: string[] | string;
19    };
20  type:
21    | "file-lines"
22    | "file-lines-full"
23    | "file-mosaic"
24    | "author-mosaic" /* Future work */
25    | "author-contributions" /* Future work */
26    | "file-bar" /* Future work */
27    | "author-bar" /* Future work */;
28  preset:
29    | {
30      gradientByAge: [string, string];
31    }
32    | {
33      paletteByAuthor: [string, string][];
34    };
35 }

```

Listing 4.5: TypeScript interface for the query schema. The `files`, `time` and `branch` properties define the target input for the visualisation. The `type` and `preset` properties define the desired output shape.

4.4.3 Scope visualisation

The visualisation scope defines two properties of the resulting visualisation: the visualisation type and its visual encoding. Gizual supports three visualisation types, shown in Figure 4.4:

- `file-lines`: Displays file contents as a series of coloured lines. Each line represents one line of code.
- `file-lines-full`: Displays file contents as a series of full-width coloured lines. Each line represents one line of code.
- `file-mosaic`: Displays file contents as a mosaic of coloured tiles. Each tile represents one line of code.

Gizual currently implements two different visual encodings:

- `gradientByAge`: Colours are assigned based on the age of the line of code.
- `paletteByAuthor`: Colours are assigned based on the author of the line of code.

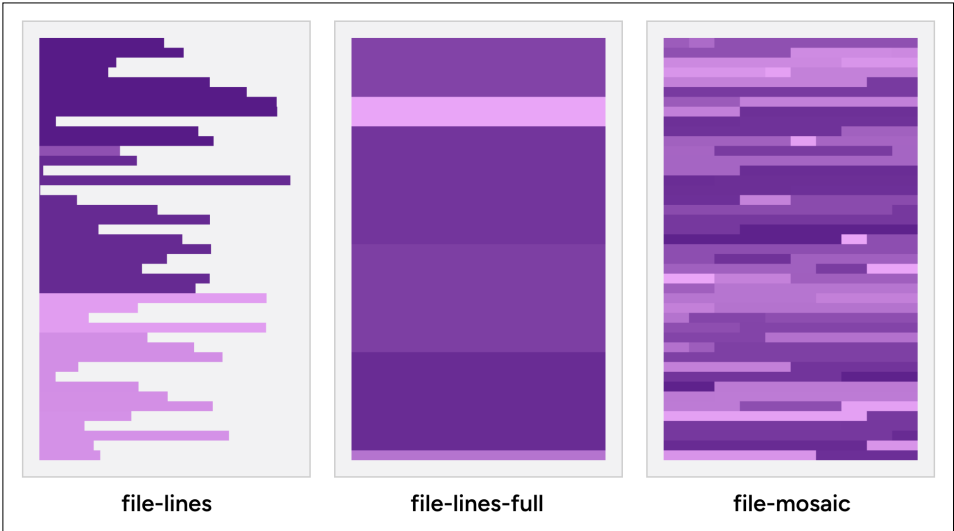


Figure 4.4: The three visualisation types provided by Gizual. [Image created by the authors of this chapter.]

Chapter 5

Gizual User Interface

“Design is the intermediary between information and understanding. ”

[Hans Hoffman; American painter, artist and teacher; 1880–1966]

To provide a good user experience, Gizual’s user interface design followed accepted design principles and best practice [Lidwell et al. 2023; Pereyra 2023; Shneiderman et al. 2017]. This chapter describes the evolution of the Gizual user interface, the current version of which is shown in Figure 5.1.

5.1 Design Principles

One extensively studied topic in the field of psychology is the dissonance between the impact of positive and negative experiences on the human psyche, often referred to as Negativity Bias [Vaish et al. 2008; Peeters 1991; Peeters and Czapinski 1990]. Aesthetically pleasing design is incredibly difficult to create, but can subconsciously invoke positive emotions. Software design is no different, and user interfaces should be crafted in a way that makes them visually appealing and easy to use [Setlur and Cogley 2022]. Some aspects of good software design are timeless. A focus on accessibility, minimising distractions, and maximising user attention should always be retained. Other principles, like a focus on certain input devices, should regularly be adapted to fit current industry trends and standards [Pereyra 2023].

Humans are very good at perceiving order and patterns in chaos. A group of German psychologists developed theories on how people perceive the world, and called them Gestalt Principles [Todorovic 2008], reasoning about vision and the bidirectional process of the mind informing the eye of what it sees and vice-versa. There are several overlapping Gestalt Principles, including:

- *Closure*: When humans see incomplete shapes, they automatically fill in the gaps to create a complete image.
- *Common Region*: Elements in containing regions are grouped automatically, inherently adding a sense of relatedness to items within the same visual group.
- *Proximity*: Elements that are closer together in physical space are assumed to be more related than those farther apart.
- *Similarity*: Similarly shaped, coloured, or spaced out elements are naturally grouped into categories.

These principles can not only be frequently found in the design of everyday objects and things, they can also provide insights in how to create visualisations that are easy to understand and follow. In case a chart or graph needs to break with one or more of these principles, it can be difficult to convey the proper meaning without additional context.

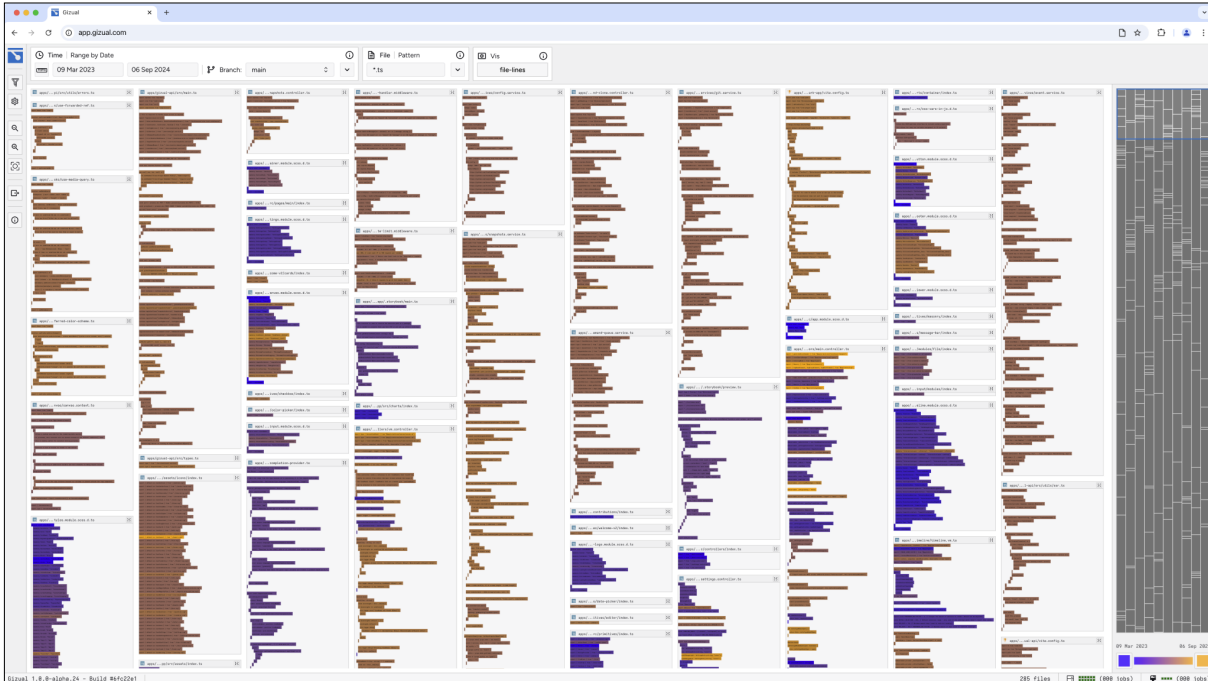


Figure 5.1: The user interface of the Gizual application, visualising the Gizual source code repository.
[Screenshot created by the author of this thesis.]

The Gizual user interface was crafted to adhere to the following general principles and guidelines:

- *Simplicity:* The user interface should never feel cluttered or overwhelming. Unnecessary labels, buttons, or other forms of visual disorder are to be avoided.
- *Similarity and Cohesion:* Similar HTML elements should look and feel consistent across all elements of the user interface.
- *Proximity:* Elements should be logically grouped alongside their most relevant siblings, making them obvious to find.

These core values often placed restrictions on what the user interface would allow in terms of general functionality. During the process of incrementally implementing features into the application, the design and the internal UI framework were extended accordingly.

5.2 Design Tooling

As in any developed field of expertise, specialised tooling can immensely speed up the web design and development processes. Prototyping tools and design systems help designers define and maintain consistent styles across components. User interface component libraries provide a set of reusable components with a cohesive look and feel for developers to use.

5.2.1 Figma

One of the most commonly used design tools in the software industry is Figma [Figma 2024]. It was initially created by Dylan Field and Evan Wallace in 2012 [Berg 2012], and has since become a pillar of modern web development. According to a recent survey, [Geoco et al. 2023], it captures more than 80% of the design tool market. Many online resources provide guides on getting started, but a complete introduction can be found in Staiano [2023]. One of the core strengths of Figma is its prototyping tool,

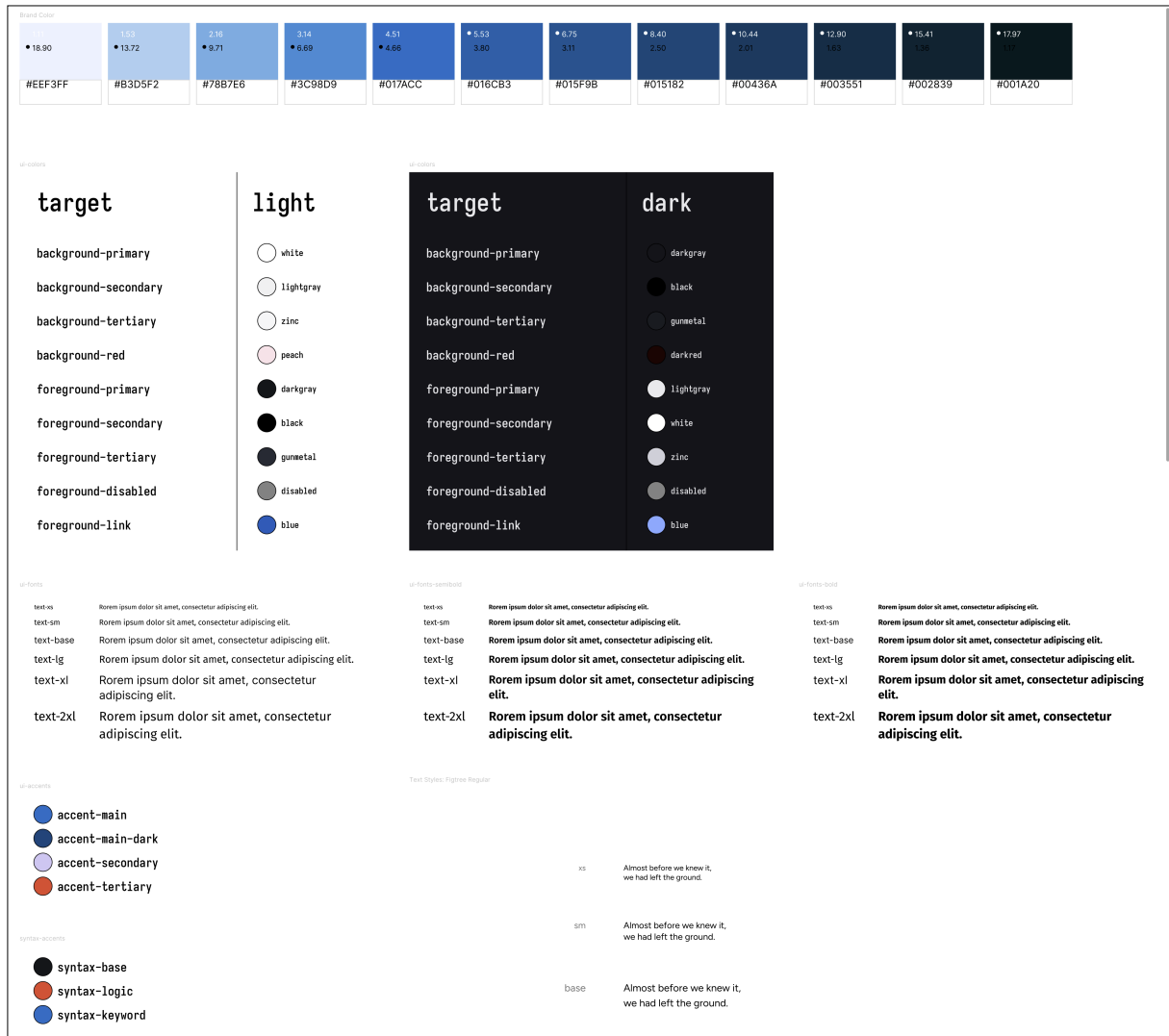


Figure 5.2: The design theme for Gizual, created in Figma. Colours and text styles are laid out in a clear and structured manner, so that they can be reused from anywhere in the design file. [Screenshot created by the author of this thesis.]

allowing designers to test out user interactions without the need to write any source code. Simple flow charts represent the flow of the UI from one interactive prototype to the next. With the recent introduction of variables, even complex behaviours like adding items to a shopping cart can be simulated within the prototype.

Many of the interactions in Gizual were not possible to recreate in Figma due to their complexity, but the tool was still heavily used to draft isolated components of the user interface before implementing them. Figure 5.2 shows the Figma design theme for Gizual, where all text styles and colours are grouped, so that they can be reused across the entire application design.

5.2.2 User Interface Component Libraries

Creating user interface components with all modern standards of interactivity and accessibility in mind can be a challenging endeavour. User interface frameworks help designers and developers by providing basic components that can be reused across the entire codebase. Using a well-established component library is often the quickest way to create a coherent user interface, without the need to consciously think about all possible variants of components. During the implementation of Gizual, two component libraries

were used extensively: first Ant [Ant Design 2024] and then later Mantine [Mantine 2024a].

At the start of the project, every component for Gizual was manually crafted using regular HTML and CSS. Best practices for user interface components had to be checked manually, and additional variants for component hover states or interactive feedback were made by hand. During the later stages of the first prototype, it became important to increase the speed of development of the user interface, so that new feature ideas could be rapidly tried out.

5.2.2.1 Ant Design

Ant Design [Ant Design 2024] is a popular user interface component library for React. It is open-source and maintained by the Chinese technology company Ant Group. It features a wide array of customisable React components for building feature-rich web applications.

The Ant Design library was added to the project for its file tree and input field components. It promised to be very extensible, and its design was highly compatible with the design language of Gizual. Significant efforts were dedicated to optimising the rendering of Ant components, especially the file tree, to ensure that large repositories could still be visualised without any major performance issues. Section 9.2 gives a detailed explanation of the rendering optimisations that were necessary to create a custom high-performance file tree component in the later stages of the project. Unfortunately, the Ant components that were used within Gizual proved to be quite difficult to style with the standard CSS techniques used by the user interface components within the project. This led to increasingly hard to maintain CSS across a line of components. Eventually, Ant was removed from the project entirely, since the maintenance issues started to outweigh the benefits.

5.2.2.2 Mantine

Mantine [Mantine 2024a] is a component library for React. It focuses on usability, accessibility and developer experience. Adoption has steadily climbed since its first stable release in 2021.

The core components from Ant were replaced with implementations from Mantine [Mantine 2024a], which were much easier to customise. In order to align the design of the Mantine components with Gizual, a `MantineProvider` is wrapped around the application entry point, and it provides default colours, fonts, and spacing. In addition to the core of Mantine, Gizual also uses the following community packages:

- *Mantine-Datatable*: This package is used to create responsive tables with pagination for listing the authors within a repository.
- *Mantine-Contextmenu*: The core of Mantine only provides a menu component attached to a button. This package provides menu functionality wrapped inside a context menu.
- *Mantine-Notifications*: This package provides a notification system that can be used globally across the application to show important notifications.

```

gizual-app/
├── src/
│   ├── assets/
│   ├── controllers/
│   ├── primitives/
│   │   ├── animated-logo/
│   │   ├── author-panel/
│   │   ├── button/
│   │   │   ├── button.module.scss
│   │   │   ├── button.module.scss.d.ts
│   │   │   ├── button.tsx
│   │   │   └── index.ts
│   │   ├── canvas/
│   │   ├── checkbox/
│   │   ├── color-picker/
│   │   ├── css/
│   │   ├── date-picker/
│   │   ├── dialog-provider/
│   │   ├── editor/
│   │   ├── file/
│   │   ├── file-tree/
│   │   ├── ...
│   │   ├── toolbar/
│   │   ├── index.ts
│   │   ├── package.json
│   │   └── tsconfig.json
│   ├── utils/
│   ├── package.json
│   ├── ...
├── index.html
├── package.json
├── svg.d.ts
├── ...

```

Listing 5.1: The structure of Gizual’s source code repository, showing the `gizual-app/` workspace.

5.3 User Interface Components in Gizual

The user interface in Gizual is encapsulated into separate self-contained components. These components are stored in the `src/primitives/` directory within the `gizual-app` package. Listing 5.1 shows a truncated excerpt of the directory and file structure. Each component is located in a subdirectory, which usually consists of the following files:

- `<component>.module.scss`: An SCSS module with all required styles for this component.
- `<component>.module.scss.d.ts`: An automatically generated file with TypeScript types corresponding to the SCSS definitions in the `<component>.module.scss` file, allowing them to be accessed with type annotations.
- `<component>.tsx`: The main entry file for the component, which imports the styles defined within the `<component>.module.scss` file and exports a React function component with the corresponding component name.
- `index.ts`: An index file which exports the main entry file, allowing imports to target `@app/primitives/<component>` instead of `@app/primitives/<component>/<component>`.

The SCSS and TypeScript files for the `Button` component within Gizual are shown in Listings 5.2 and 5.3. Every reusable part of the user interface is defined in this same way. The `Button` component uses

```

1 @use "@mixins" as *;
2 @use "@colors";
3
4 .ButtonBase {
5   @include transition-all;
6   padding: 0.25rem 1rem;
7   display: inline-flex;
8   flex-direction: row;
9   gap: 0.5rem;
10  align-items: center;
11  justify-content: center;
12  text-align: center;
13  border: none;
14 }
15
16 .Button {
17   border-radius: 4px;
18   font-weight: 500;
19   border: 1px solid var(--border-tertiary);
20
21   &:active {
22     @include button-click-animation;
23   }
24   &:hover {
25     cursor: pointer;
26   }
27   &:disabled {
28     border-color: var(--border-primary);
29     color: var(--foreground-disabled);
30     & > svg {
31       & > path {
32         fill: var(--foreground-disabled);
33       }
34     }
35     background-color: var(--background-tertiary);
36     cursor: not-allowed;
37   }
38   & > svg {
39     margin: 0;
40   }
41 }
42
43 .ButtonUnstyled {
44   &:hover {
45     cursor: pointer;
46   }
47
48   &:disabled {
49     cursor: default;
50   }
51 }
52 // Content truncated

```

Listing 5.2: The file `button.module.scss` contains styling for Gizual's `Button` component. Hover and active states are defined to ensure consistent behaviour.


```
1 import clsx from "clsx";
2 import React from "react";
3 import style from "../button.module.scss";
4
5 type ButtonVariant =
6   | "filled"
7   | "outline"
8   | "gray"
9   | "dangerous"
10  | "unstyled"
11  | "primary"
12  | "secondary";
13 type ButtonSize = "small" | "regular" | "large";
14 const buttonVariantCSSMapping: Record<ButtonVariant, string> = {
15   filled: style.ButtonFilled,
16   dangerous: style.ButtonDangerous,
17   outline: style.ButtonOutline,
18   gray: style.ButtonGray,
19   unstyled: style.ButtonUnstyled,
20   primary: style.ButtonFilled,
21   secondary: style.ButtonGray,
22 };
23 const buttonSizeCSSMapping: Record<ButtonSize, string> = {
24   small: style.ButtonSmall,
25   regular: style.ButtonRegular,
26   large: style.ButtonLarge,
27 };
28 type ButtonProps = React.ButtonHTMLAttributes<HTMLButtonElement> & {
29   children: React.ReactNode;
30   variant?: ButtonVariant;
31   size?: ButtonSize;
32 };
33 export const Button = React.forwardRef<HTMLButtonElement, ButtonProps>(
34   ({ className, children, variant = "filled", ...props }, ref) => {
35     return (
36       <button
37         className={clsx(
38           className,
39           style.ButtonBase,
40           variant === "unstyled" ? undefined : style.Button,
41           `${buttonVariantCSSMapping[variant]}`,
42           `${buttonSizeCSSMapping[props.size ?? "regular"]}`
43         )}
44         type="button"
45         {...props}
46         ref={ref}
47       >
48         {children}
49       </button>
50     );
51   }
52 );
```

Listing 5.3: The file `button.tsx` serves as the main source file for the `Button` component.

```

1 import { DatePickerInput } from "@mantine/dates";
2
3 import { DATE_DISPLAY_FORMAT } from "@giz/utils/gizdate";
4
5 import style from "../date-picker.module.scss";
6
7 type DatePickerProps = {} & React.ComponentProps<typeof DatePickerInput>;
8
9 /**
10  * Wrapper around the Mantine DatePickerInput component.
11  * @param styles - Styles appended to the default overrides.
12  * @param props - The props for the DatePicker component.
13  */
14 function DatePicker({ styles, ...props }: DatePickerProps) {
15   const height = 30;
16   return (
17     <DatePickerInput
18       className={style.DatePicker}
19       styles={{
20         input: {
21           height: height,
22           minHeight: height,
23           maxHeight: height,
24           padding: "0 0.5rem",
25           minWidth: 150,
26           width: "100%",
27         },
28         label: {
29           fontWeight: 500,
30         },
31         ...styles,
32       }}
33       {...props}
34       valueFormat={DATE_DISPLAY_FORMAT}
35     />
36   );
37 }
38
39 export { DatePicker };

```

Listing 5.4: The `date-picker.tsx` file serves as the main entry file for the `DatePicker` component, and wraps around Mantine's built-in `DatePickerInput` component.

the basic HTML `<button>` element internally. Other components often rely on a component base from the Mantine library, but the directory setup for the UI component in Gizual is the same. This ensures a consistent look and feel across all parts of the user interface that use this component. If changes need to be made, they propagate through the entire user interface seamlessly. Listing 5.4 demonstrates this setup for the `DatePicker` component. The default styles of the input element within the component are replaced with defaults that better suit the design of Gizual, and the `DATE_DISPLAY_FORMAT` variable is used to format all date values consistently.

With all user interface elements grouped in the single `primitives/` directory, exposing them for import in other modules is quite trivial, using basic functionality provided through the yarn package manager. All user interface elements are registered in the `@app/primitives` package, and can be used from anywhere in the application by including `@app/primitive/<component>`.

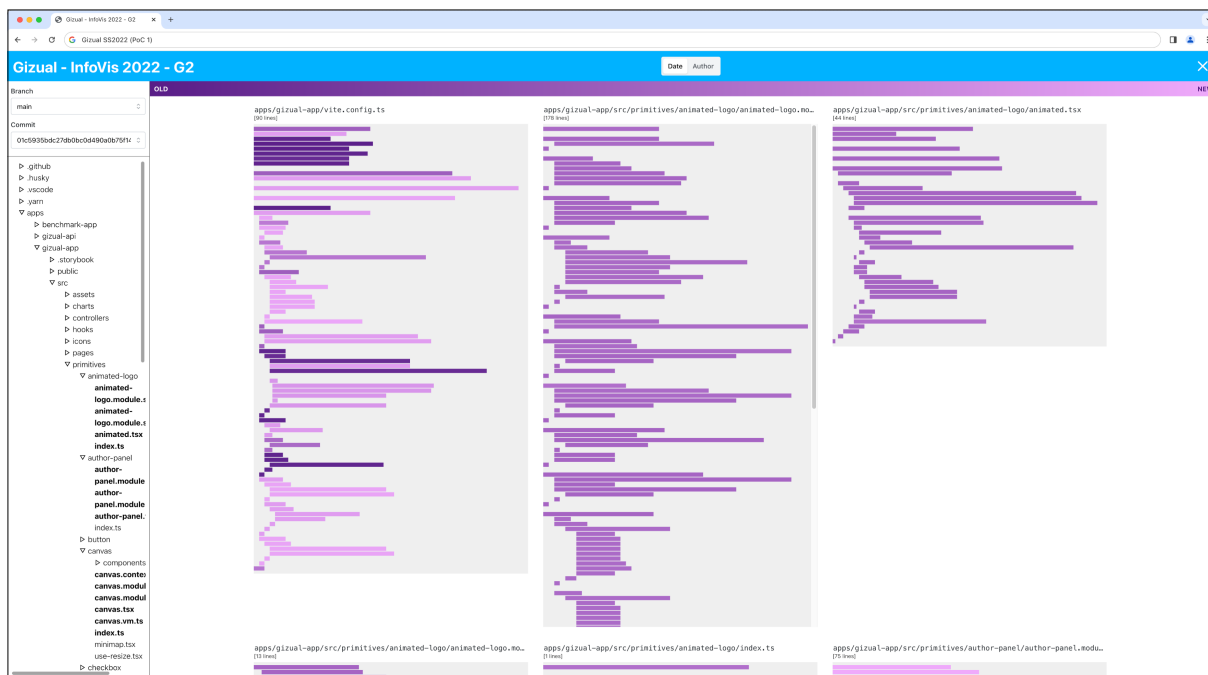


Figure 5.3: The user interface of Gizual POC1, created by Korduba et al. [2022] in the summer term of 2022. The interface features a visualisation canvas in the centre, a file tree on the left, and a horizontally aligned legend component at the top. [Screenshot created by the author of this thesis.]

5.4 Previous Design Iterations

The version of Gizual presented in this thesis was built on top of learnings from two previous seminar projects on the same topic [Korduba et al. 2022; Pinheiro de Souza et al. 2023], which will be referred to as Proof of Concept 1 (POC1) and Proof of Concept 2 (POC2) respectively. In contrast to the design principles laid out in Section 5.1, the first two versions of Gizual were mostly research experiments. As such, they did not follow any strict guidelines on user interface design, and were generally crafted in a way to include as much functionality as was possible to create within the span of a few weeks. As such, functionality was often cumbersome to reach, and the general user experience was subpar, especially on non-standard desktop viewpoints. Additionally, devices smaller than tablets were disregarded completely, and the tool was unusable on them.

Figure 5.3 shows the user interface of POC1, as described by Korduba et al. [2022]. It features a central canvas to display files, a file tree and settings panel on the left, and a horizontal legend component at the top. Users were able to scroll within the central canvas component, and hovering on a line within the visualisation would reveal additional information. As a proof of concept, this interface was sufficient to demonstrate the feasibility of the general project architecture. File selection was quite cumbersome, as each file had to be selected manually, and the application itself only worked well on screens with the viewport size of a typical desktop monitor. Additional problems included general sluggishness during scroll interactions, lack of overview within the main canvas, and lack of customisation of the visualisation output.

In POC2, described by Pinheiro de Souza et al. [2023], some of these problems were addressed. The result of this iteration can be seen in Figure 5.4. A new canvas implementation allowed for more files to be visible at one time, whilst still keeping a familiar look and feel. The settings panel on the left was extended to feature additional toggles and options to customise the visualisation colour. The file tree was enhanced, and the option to filter files via matching expressions was added. This interface solved issues regarding general user interface usability. Proximity grouping in a single settings panel on the left ensured that all relevant toggles were located next to each other. Unfortunately, the new canvas layout

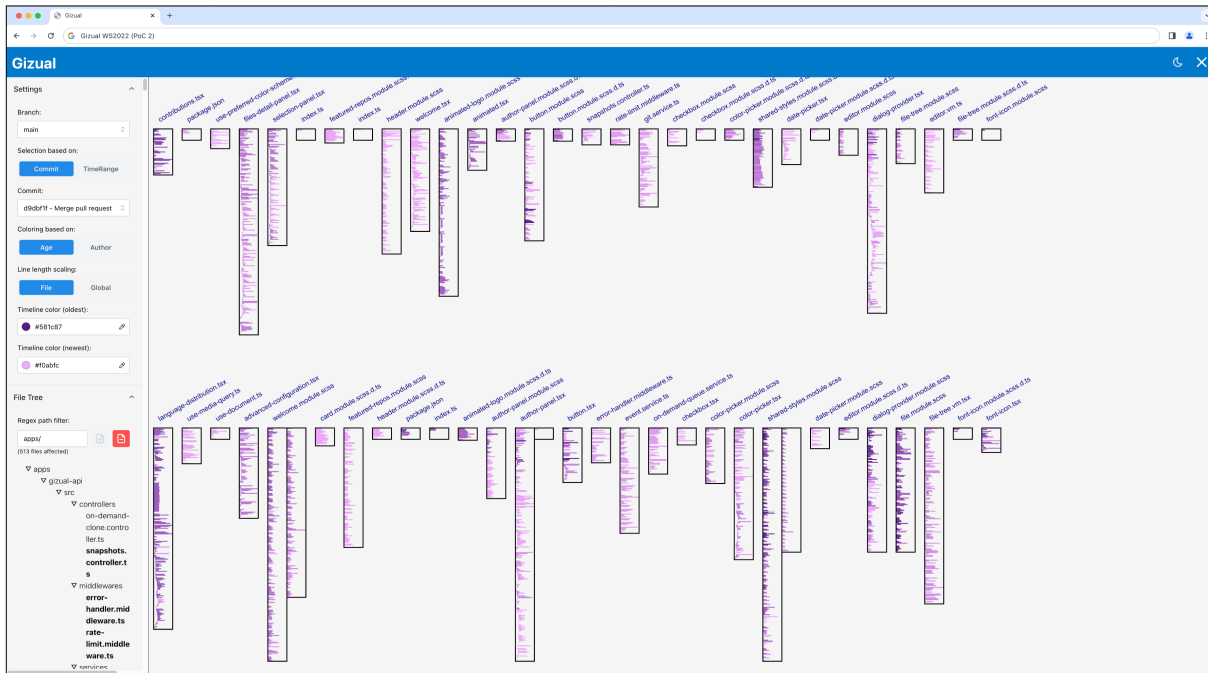


Figure 5.4: The user interface of Gizual POC2, created by Pinheiro de Souza et al. [2023] in the winter term of 2022/23. The interface features a visualisation canvas in the centre and a file tree and settings panel on the left. [Screenshot created by the author of this thesis.]

re-introduced some inconveniences when file names needed to be truncated, and general sluggishness of scrolling within the visualisation still remained an issue.

5.5 Current User Interface

The issues that were prevalent in POC1 and POC2 provided valuable insights for the current user interface of Gizual. The user interface is split into five main regions, as shown in Figure 5.5:

- **Canvas:** A central canvas acts as the heart of the visualisation. Individual tiles represent files, and coloured strips within the tiles represent lines of code. The colour-coding is based either on the age or the author of each line of code. Tiles are laid out in a masonry grid, and the entire area is interactive. Users can navigate around by zooming, panning, or pinching.
- **Query Bar:** The Query Bar above the Canvas provides all functionality relating to file selection and customisation of the visualisation.
- **Toolbar:** The Toolbar on the left provides quick access to navigation functionality, so that users without a mouse-wheel or people with special needs can navigate within the Canvas.
- **Sidebar:** The Sidebar on the right provides a custom Minimap, which mirrors the content of the main canvas in an abstracted, global overview. Both components are synchronised, so that any movement within either one of them is immediately replicated in the other. Additionally, a legend component at the bottom of the Sidebar provides an overview of the visualisation colours and quick toggles to change them directly.
- **Status Bar:** The Status Bar provides feedback about ongoing operations and current resource usage. It displays the number of selected files, the current usage of explorer workers, and the current usage of renderer workers, as explained in Chapter 4.

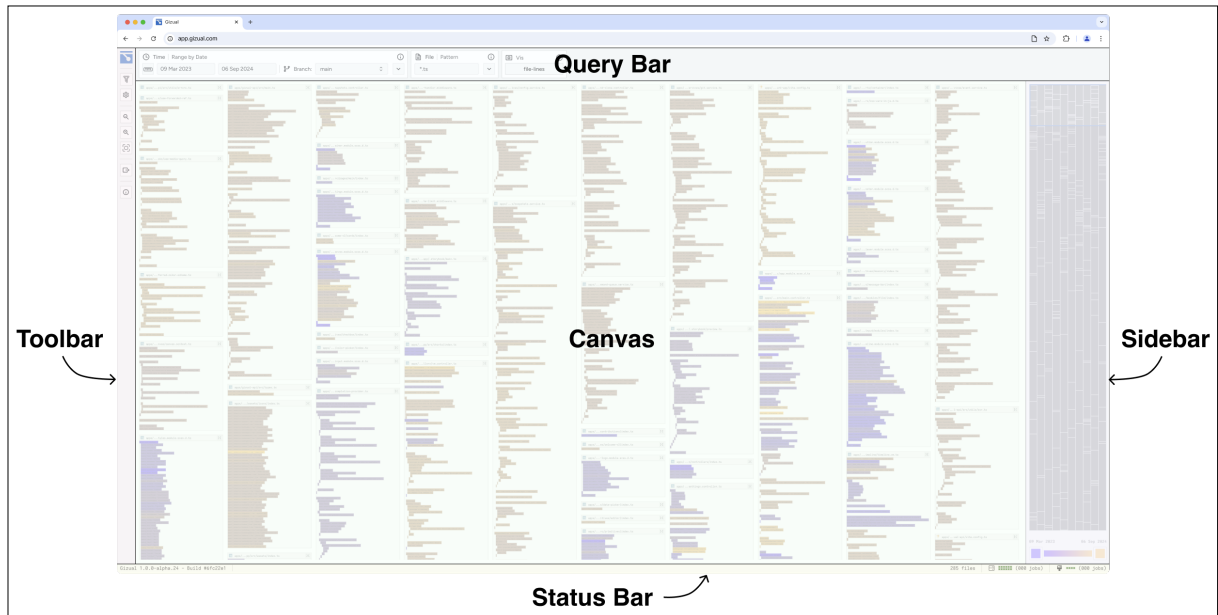


Figure 5.5: The user interface of the current release of Gizual. The interface is separated into five main regions. The central Canvas provides the visualisation of files in the repository. The Query Bar at the top customises file selection. The Toolbar on the left provides quick access to navigation functionality. The Sidebar to the right provides an abstracted, global overview of the Canvas. Finally, the Status Bar at the bottom provides feedback about ongoing operations and current resource usage. [Screenshot created by the author of this thesis.]

Each distinct region provides users with access to closely related functionality. The most feature-packed region of the user interface is the Query Bar, which will be described in detail in Chapter 8. During the project phase of this thesis, many iterations were necessary to reach the final design. Feature requests always had to be considered carefully, in order to keep the general design language and the look and feel of the application consistent.

The Gizual user interface is optimised for a variety of viewport sizes, devices, and web browsers. On devices with narrow viewports, sections of the user interface like the *Query Bar* are hidden or behave differently. The main canvas is always the most prominent UI component on display, and it can be navigated equally well with mouse, touch, or touchpad devices.

5.6 Designing for Interactivity

Interactivity in software design is a well-defined field, with notable contributions made in the gaming and virtual reality sectors by Ryan [2003], and in the advertising sector by Li [2011]. Interactive web design shares some similarities with video games and advertising, but requires tailored solutions for different kinds of problems. The field of interactive data visualisation, as popularised by modern newspapers, often pushes the boundaries of the media by creating novel and unique approaches to visualise data. An example of an infinitely zoomable visualisation is the Scale of the Universe project by Huang et al. [2024], which uses interactivity to help users grasp the vastness of space. An interactive slider can be used to adjust the camera position within the visualisation, giving users full agency about their desired exploration speed.

Using interactivity to put the user in a central role within a virtual space or a representation of data has become a common practice in modern web design. The user interface of Gizual aims to provide the minimum required amount of interactivity, so that things are sufficiently interactive, but the additional complexity is not too overwhelming. Designing web experiences with interactivity in mind often requires

a slight mental shift. Instead of exposing functionality in always visible elements of the user interface, clever use of interactivity may allow for features to be hidden, reducing the visual complexity at first glance. Doing so does come at the cost of discoverability, though. Decluttering the user interface too much and hiding features in difficult to navigate sub-menus can also lead to user exhaustion and frustration. Within Gizual, the following user interactions were deemed natural enough to require no additional explanations or hints, primarily concerning the Canvas and Timeline:

- *Zoom*: Both the Canvas and Timeline allow users to zoom by scrolling with the mouse-wheel.
- *Pan*: The Canvas allows users to pan with a single finger or mouse drag. The Timeline supports the same motion, but requires a three-finger gesture on touch devices, since a drag with a single finger moves the selected time range. This behaviour is deliberate and is described in Section 9.1. Users with a mouse and keyboard can pan the timeline by scrolling while holding down the `Shift` key.
- *Pinch*: Both the Canvas and the Timeline support pinch-zooming for touch screens.

The Canvas is described in Chapter 6. Usage of the Timeline is explained in Chapter 8. Section 9.1 provides a detailed explanation of the interactive elements in the Timeline.

Chapter 6

Gizual Canvas

The main visualisation canvas in Gizual was inspired by the design and feature set of Seesoft [Eick et al. 1992], using the metaphor of listings of source code hanging on a wall far away. It can also be regarded as a kind of information mural [Jerding and Stasko 1998], providing a condensed, abstract two-dimensional representation of an information space, with the ability to selectively focus in on a specific subset of data within the space.

Seesoft used a rectangular box for each file in the visualisation. Inside the box, individual lines of code were represented as coloured strips. If the number of lines of code exceeded a certain threshold, the box wrapped into a second column. The colour-coding of a strip reflected the particular metric of interest, such as age of the line of code or last author of the line of code. Multiple levels of zoom window were provided to zoom down to the level of individual lines of code.

In Gizual, the visualisation canvas is a mural of tiles, organised in a masonry layout [MDN 2024e]. Each tile represents one file in the repository, typically of source code. Within a tile, each line of code is represented by a coloured strip. This representation is implemented in the Canvas component, which is shown in Figure 6.1. On wide viewports, the Sidebar is displayed to the right of the Canvas. The Minimap component in the Sidebar shows the currently visible viewbox, in relation to the entire canvas. Interactive navigation supports zooming and panning across the visualisation, so a user can both obtain an overview and zoom in on a particular region or an individual file. For the purpose of this discussion, the “visualisation canvas” represents the virtual arrangement of files in infinite space. The Canvas represents the UI component that displays a section of the visualisation canvas within its viewbox and provides support for interactivity. The Sidebar and Author Panel are placed to the right of the Canvas.

6.1 File Tiles

Each rectangular tile on Gizual’s visualisation canvas represents one file in the repository. Within a tile, each line of code is represented by a coloured strip. Binary files such as images are represented by special tiles. Technically, each tile is considered to be a block within the Gizual architecture. Tiles can be arranged in arbitrary orders and are designed to be self-contained, so that they can be printed or exported in isolation.

Unlike Seesoft, Gizual does not wrap long files into multiple columns, since that would interfere with the masonry layout algorithm of the Canvas. Instead, each tile displays up to 400 lines of code per file at once. If a file contains more lines of code it is truncated, and an additional text at the bottom of the tile informs the user about the truncation. The defined number of lines per file is exposed as a setting and users can freely modify it, or disable truncation entirely. When fully zoomed out, the characters may be too small to read, but the user can easily zoom in. This is in contrast to the original implementation in Seesoft, which used a separate zooming window to show individual lines of code. To avoid issues

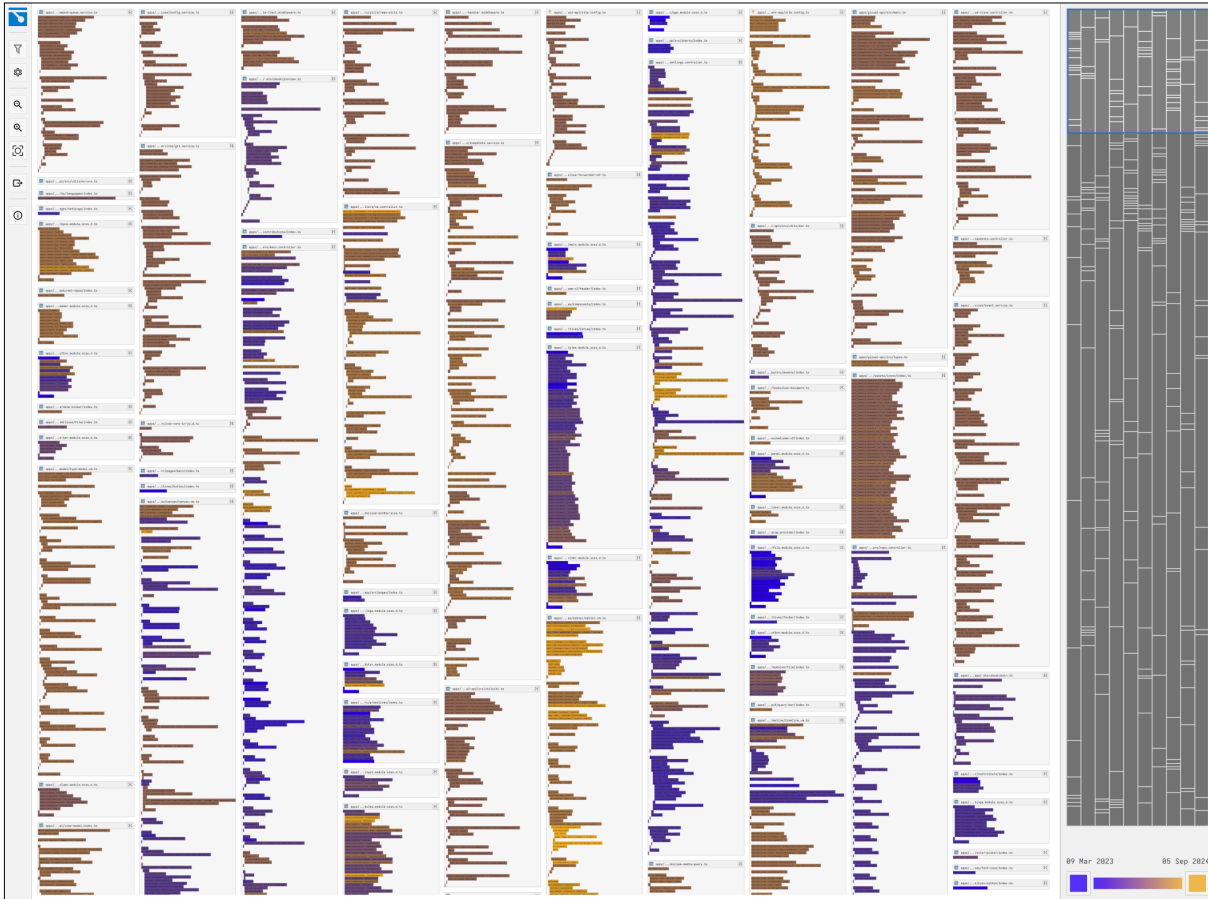


Figure 6.1: Gizual’s Canvas with Gradient by Age visual encoding. Each tile represents a file of source code. Tiles are arranged in a masonry layout. The Sidebar is displayed on the right, and contains the Minimap and Legend. [Image created by the author of this thesis.]

with files containing very long lines of code, Gizual trims each line at a fixed maximum length of 120 characters. This ensures a consistent visual layout and improves the readability of the visualisation.

Inside each tile, each line of code is represented by a coloured strip. The horizontal extent of each strip corresponds to the number of characters in the associated line of code (or spans the entire width of the container, depending on user preference). Gizual renders the content of each line of code as text inside the coloured strip. Figure 6.2 shows a side-by-side comparison of traditional Git blame output and the Gizual equivalent for part of a `package.json` file.

For large repositories with many files, screen real-estate can be conserved at the expense of seeing lines of source code, by rendering lines in mosaic mode. Instead of rendering each line of code as a strip in the tile, each line is rendered as a coloured box, and 10 boxes are rendered from left to right. The colouring of each box is identical to the colouring of the standard strip. Figure 6.3 shows Gizual’s line mode and mosaic mode side by side.

Source code repositories often contain many thousands of individual files. To rapidly generate visual representations for all relevant source code files, the rendering of each tile is performed by a web worker from a shared pool. Gizual supports three different types of renderer:

- `CanvasRenderer`: Renders into an offscreen canvas.
- `SvgRenderer`: Renders into an SVG string.
- `AnnotationRenderer`: Renders into an HTML string.

The image shows two side-by-side views of the same file, `package.json`. The left view is the output of a standard `git blame` command, showing a list of commit hashes, author names, dates, and line numbers for each line of code. The right view is the same file rendered in the Gizual application, where each line is highlighted with a color gradient representing its age, with older lines being darker purple and newer lines being lighter purple. The Gizual view also includes a file icon and a search icon in the top right corner.

Figure 6.2: Comparison of output from a traditional Git blame command on the left, and the equivalent in Gizual on the right (coloured by age of line of code). [Image created by the author of this thesis.]

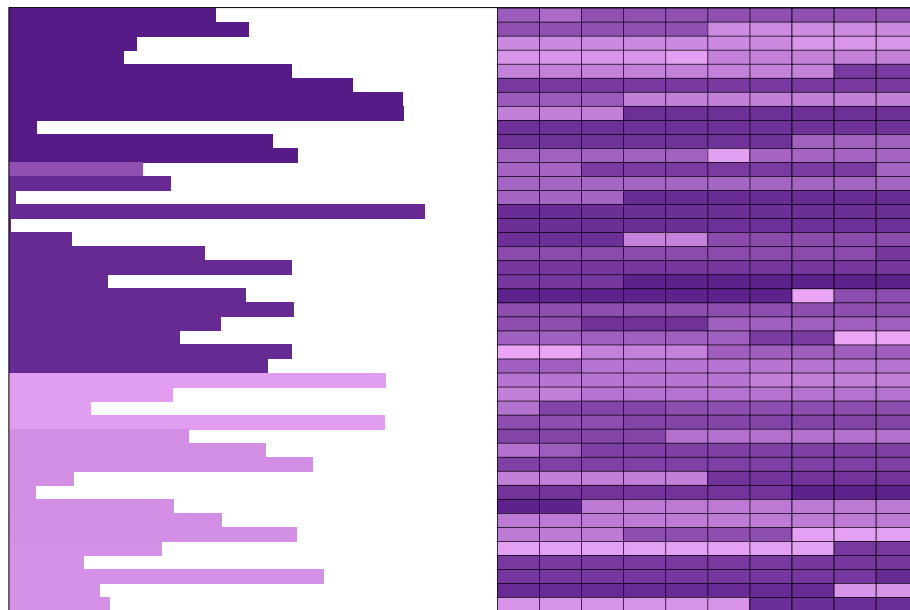


Figure 6.3: Comparison of visualisation tiles in standard line mode (left) and mosaic mode (right). In line mode, each coloured strip represents one line of code. In mosaic mode, each coloured box represents one line of code, and ten boxes are drawn in each row. [Image created by the author of this thesis.]

All three renderers are based on the `BaseRenderer` interface, which enforces the structure of their implementation. Listing 6.1 shows the interface definition of the `BaseRenderer`.

This modular approach of composing different renderers based on a shared interface ensures that the renderer worker can use any of the three renderers to complete a given rendering job. The shared interface enforces the structure of input and output. In the source code, the choice of renderer is passed to the web worker through a `mode` variable. Based on this variable, the web worker then instantiates the corresponding renderer, and assigns the drawing context to it. Once the context has been assigned, the web worker executes a rendering function based on the selected visualisation type. The implementation of this rendering function for the `file-lines` and `file-lines-full` visualisation types is shown in Listing 6.2. After executing a rendering function, the result from the rendering worker is handed over to Maestro, which propagates it back to the main thread, where the browser takes over and renders the image in the Canvas.

```

1 import { SvgAttributes, SvgElement } from "@app/utils/svg";
2
3 export type AnnotationContext = Object;
4
5 export type RectAnnotation = {
6   width: number;
7   height: number;
8   x: number;
9   y: number;
10  color: string;
11  ctx: AnnotationContext;
12 };
13
14 export type TextAnnotation = {
15   text: string;
16   x: number;
17   y: number;
18   fontSize: number;
19   ctx: AnnotationContext;
20 };
21
22 export type AnnotationObject = TextAnnotation | RectAnnotation;
23
24 export type ValidContext = OffscreenCanvas | SvgElement | AnnotationObject[];
25
26 export interface BaseRenderer {
27   prepareContext(width: number, height: number, dpr?: number): void;
28   assignContext(ctx: ValidContext): void;
29   getContext(): ValidContext | undefined;
30   getReturnValue(): Promise<string | string[]>;
31
32   applyTransform(x: number, y: number): void;
33   drawRect(attr: SvgAttributes, annotationCtx?: AnnotationContext): void;
34   drawText(
35     text: string,
36     attr: SvgAttributes,
37     annotationCtx?: AnnotationContext
38   ): void;
39 }
40
41 export function evaluateTransform(
42   x: number,
43   y: number,
44   transform: { x: number; y: number }
45 ): { x: number; y: number } {
46   return { x: x + transform.x, y: y + transform.y };
47 }

```

Listing 6.1: The `BaseRenderer` enforces function implementations and types across all three supported renderers.

```
1  async drawFileLines(ctx: FileLinesContext, renderer: BaseRenderer) {
2    const colors: string[] = [];
3    const { width } = calculateDimensions(ctx.dpr, ctx.rect);
4    const lineHeight = 10 * ctx.dpr;
5    this.colorManager.init(ctx.colorDefinition);
6    let currentY = 0;
7    const widthPerCharacter = width / ctx.lineLengthMax;
8
9    for (const line of ctx.fileContent) {
10     const lineLength = line.content.length;
11     let rectWidth = width;
12     let lineOffsetScaled = 0;
13
14     if (ctx.visualizationConfig.style.lineLength === "lineLength") {
15       lineOffsetScaled = (line.content.length -
16         line.content.trimStart().length) * widthPerCharacter;
17       rectWidth = Math.min(
18         lineLength * widthPerCharacter - lineOffsetScaled,
19         width - lineOffsetScaled,
20       );
21     }
22
23     const color =
24       line.commit && !ctx.isPreview
25         ? this.colorManager.interpolateColor(ctx, line)
26         : "transparent";
27     line.color = color;
28     colors.push(line.color ?? "#000");
29
30     renderer.drawRect({
31       x: lineOffsetScaled,
32       y: currentY,
33       width: rectWidth,
34       height: lineHeight,
35       fill: color,
36     });
37
38     if (ctx.showContent)
39       renderer.drawText(line.content, {
40         x: 0,
41         y: currentY + Math.round(lineHeight / 1.5),
42         fontSize: "4.1",
43         fill: ctx.visualizationConfig.preferredColorScheme
44           === "dark" ? "white" : "black",
45       });
46     currentY += lineHeight + VisualizationDefaults.lineSpacing;
47   }
48   const result = await renderer.getReturnValue();
49   return { result, colors };
50 }
```

Listing 6.2: The `drawFileLines` function is called inside the renderer web worker to generate visual output for the `file-lines` and `file-lines-full` visualisation types.

6.2 Masonry Canvas

All rendered file tiles are placed on a two-dimensional canvas. Pan, zoom, and pinch operations are supported within this Canvas component, using the `react-zoom-pan-pinch` [BetterTyped 2024] library.

Tile positioning on the Canvas is determined by a masonry layout algorithm [MDN 2024e]. In essence, this algorithm creates $1..N$ columns, based on user preference, with a default of 10. All tiles are sorted based on their calculated height. The highest tile is inserted into the currently shortest column in the layout. This step is repeated until all tiles have been assigned to a column. Afterwards, tiles are sorted within their respective columns based on their ID, which typically corresponds to the name of the file. Finally, all columns are sorted left to right based on their height, with the shortest on the left. This algorithm produces a stable masonry layout for an arbitrary number of tiles, and is used for the masonry canvas and the SVG export. Figure 6.4 shows an SVG export of a masonry grid created with Gizual.

6.2.1 Canvas Interactivity

The Canvas can be navigated with the mouse or by touch. Clicking and dragging with the mouse and touching and dragging with a finger trigger a pan operation, which moves the viewbox of the visualisation. Scrolling on the mouse wheel or pinching with two fingers are converted into a zooming interaction. To maintain a smooth user interface during the animation of these transform operations, tiles can be rendered in varying degrees of detail. Zooming out reduces the required resolution for each tile in the visualisation. The Intersection Observer API [MDN 2024j] is used to track tiles which are currently in the viewbox. Tiles outside the viewbox are disregarded. Zooming into the visualisation uses the same logic to determine which tiles to render at a higher resolution. Once a tile has been rendered in a different resolution, the image in the visualisation is replaced with the new result. This process ensures smooth interactivity within the Canvas by reducing the resolution of items the browser needs to render.

6.2.2 Canvas Minimap

The Canvas can display many tiles simultaneously, and grows infinitely in height. User navigation can become increasingly more disorienting with increased canvas height. For this reason, a Minimap component was added to the Sidebar, as shown in Figure 6.5. The Minimap displays the entire canvas of tiles at a much lower scale, and uses grey rectangles to reduce information density. The current viewbox of the Canvas is indicated with a blue rectangle.

The `react-zoom-pan-pinch` library provides an implementation of a minimap out of the box, but it had to be adapted to include interactivity. In Gizual, the Minimap syncs bidirectionally with the Canvas. Panning or zooming in the Canvas updates the Minimap, and dragging the blue viewbox rectangle in the Minimap updates the Canvas.

6.2.3 Canvas Legend

The Sidebar also features a Legend, displaying the selected colour coding and commit range for the Gradient by Age visual encoding, as shown in Figure 6.6. The component features two colour pickers, which provide quick access to change the start and end colour of the colour gradient.

6.2.4 Author Panel

When the Palette by Author visual encoding is selected, the Author Panel displays the authors of the selected commits. Figure 6.7 shows the Canvas for the Palette by Author visual encoding, with the Author Panel visible on the right. Author names and avatars are anonymised for this example.

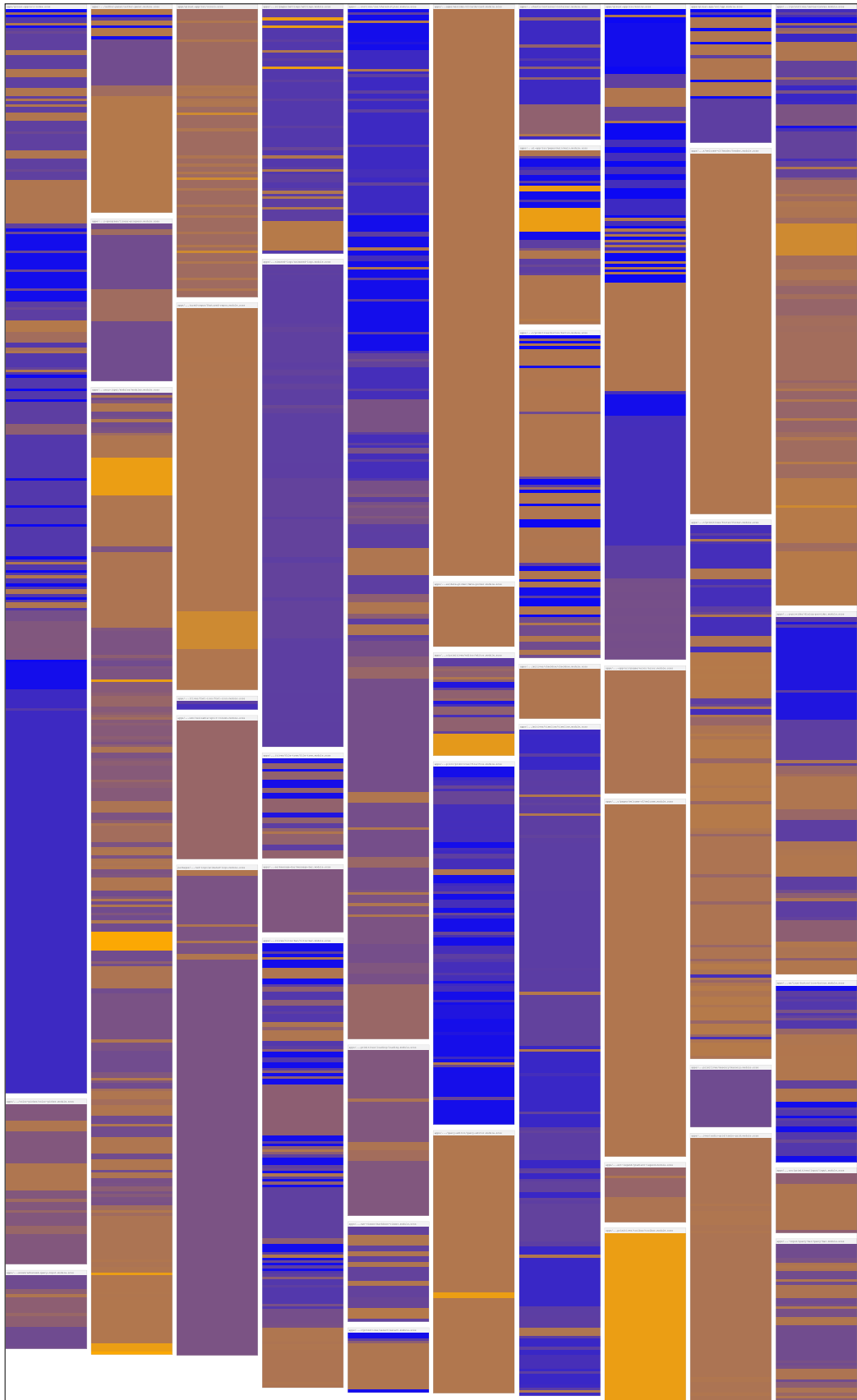


Figure 6.4: SVG export of a masonry grid created in Gizual. The resulting layout of the algorithm is stable. Columns are sorted left to right by their height. Within columns, tiles are sorted based on their ID. The text of lines of code is not exported. [Image created by the author of this thesis.]

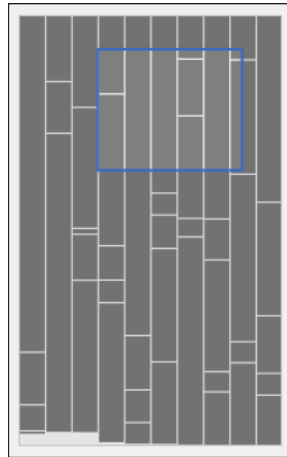


Figure 6.5: The Minimap component in Gizual, for a set of tiles where the user has zoomed in on the Canvas. All tiles in the Minimap are displayed as simple gray rectangles, retaining only their relative height. The blue rectangle indicates the current viewbox of the Canvas. [Image created by the author of this thesis.]



Figure 6.6: The Legend component in the Gizual Canvas for the Gradient by Age visual encoding. Two colour pickers provide quick access to customise the start and end colours of the colour gradient. The selected commit range is shown in text. [Image created by the author of this thesis.]

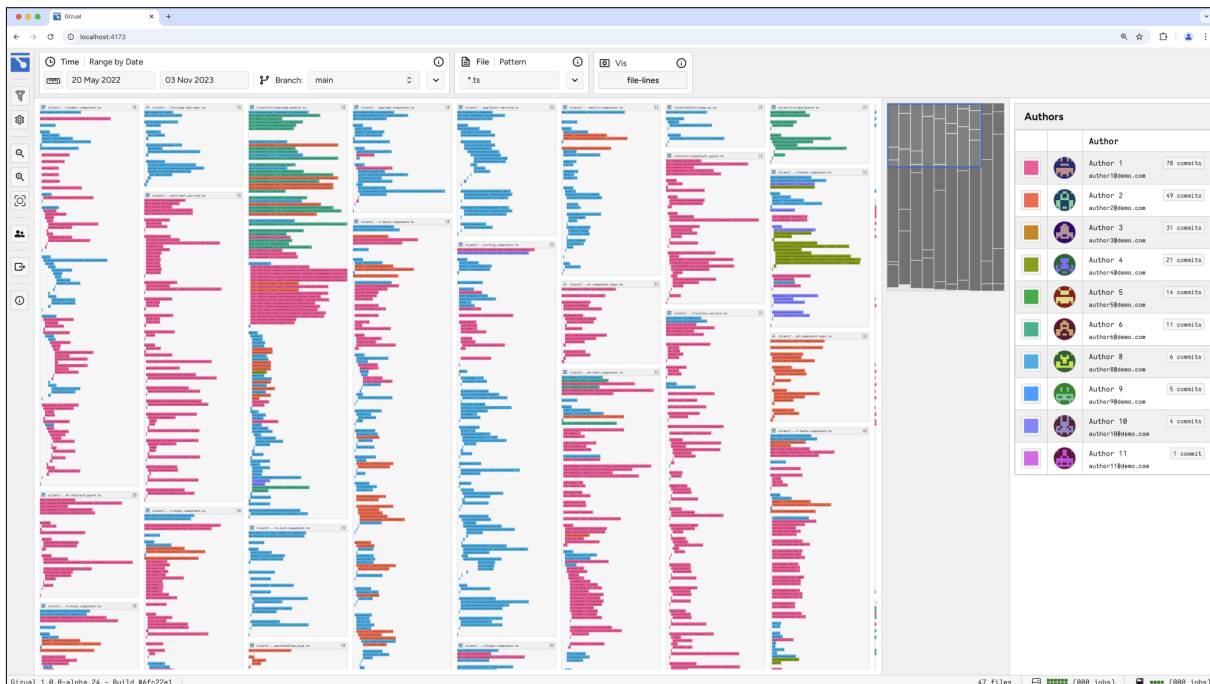


Figure 6.7: A Gizual visualisation using the Palette by Author visual encoding. The Author Panel on the right shows the colour palette used to identify the author of each line of code. Avatars are anonymised in this example. [Image created by the author of this thesis.]

Chapter 7

Visual Encoding in Gizual

In Gizual’s visualisation canvas, files are drawn as rectangular tiles and lines of code are drawn as coloured horizontal strips within them. The length of the horizontal strip typically depends on the length of the line of code (number of characters). The horizontal strips are colour-coded according to one of two visual encodings: Gradient by Age or Palette by Author. In Gradient by Age encoding, the colour of each horizontal strip is calculated according to the timestamp of the last modification of the line of code. Users can freely choose the colour of the start and end of the time range, and the colour value for a particular line of code is interpolated between those colours based on the timestamp. In Palette by Author encoding, each author within the selected time range is assigned a colour value from a palette. Each horizontal strip is assigned the colour of the author who last modified the line of code. Users can freely customise the assigned colour values of each author, but sensible defaults ensure that there is clear visual distinction between the first 30 authors.

7.1 Colour Spaces

Traditionally, computer displays have relied on the RGB colour coding to persuade the human brain that it is looking at a vivid, colourful image, when, in reality, it is just looking at a set of millions of tiny red, green and blue lights. To represent the colour red, an RGB value of `rgb(255,0,0)` would be used, whilst the colour blue would be represented as `rgb(0,0,255)`. This approach of encoding colour makes sense based on its physical origin. However, colours coded with RGB are usually hard to understand for humans, because the mixture between colour channels is not linear. Instead, the human brain can perceive slight variations in RGB values as huge changes in colour, making it difficult to create proper colour gradients within this colour space.

A more modern alternative for representing colour, better-suited for the purpose of data visualisation, is the HCL colour space [Zeileis et al. 2009], often also referred to as CIELCh_{uv}. Ihaka [2003] popularised its use for information graphics by applying the principles of colour from Munsell [1919] to the colour space, proposing a set of uniformly distributed colours. For visualisation, the HCL colour space has benefits over RGB, because it eliminates the bias of varying saturation of colours, rooted within the human visual system. An excellent write-up on the historic developments of the HCL colour space and comparisons to other colour spaces was published by Rhyne [2021]. The hclwizard.org website [hclwizard 2024] shows an overview of hue, chroma, and luminance colour maps and serves as a great introduction.

7.2 Gizual Colour Manager

Within the Gizual project, colour distribution is managed through instances of a `ColorManager` class. This class is responsible for assigning colour to domain values, and for transforming values from one colour space to another. Through this abstraction layer, all web workers within the Gizual architecture can access the ground truth mapping of domain value to colour by instantiating an instance of the `ColorManager` with the same `ColorSetDefinition`. Listing 7.1 showcases a version of the `ColorManager` with the function implementations removed for brevity.

For low-level colour manipulation, the `ColorManager` uses scale and colour transformation functions provided by the D3 library [Bostock et al. 2011; Bostock 2024a], a well-established project for creating data-driven documents and interactive visualisations. D3 provides excellent functions for managing colour bands and colour space conversion.

The colour gradient for the *Gradient by Age* visual encoding is calculated by assigning a user-defined value to the chosen start and end timestamp. If the timestamp value of a line of code is within the specified range of commits, it is passed into a D3 colour range function, otherwise a semi-transparent colour is assigned. Users can freely modify the colour values of the two extreme values, and the colour of timestamps that are out of the selected time range. The D3 colour range uses a linear scale [Bostock 2024b] to transform the timestamp value and the two user-defined start and end values into the final colour value for a given line of source code.

Colouring for the *Palette by Author* visual encoding works by first assigning a fixed set of colours to a colour palette. The colour palette is initialised upon creation of the `ColorManager` instance, and defaults to 30 distinct colours. Algorithmically, each colour is generated within the HCL colour space by evenly distributing the hue across all values of the band, with chroma and luminance kept consistent for all colours. This colour scale is fed into a `ScaleOrdinal` [Bostock 2024c], which maps the 30 colour values to values within the target domain. If a value for a specific domain entry is user-defined, it is fed into the `init` function of the `ColorManager`, which assigns it directly to the associated domain value. Figure 7.1 shows a design prototype of the *Author Panel*, which displays author colour next to author avatar, name, email address, and number of commits.


```

1 import { hcl, HCLColor } from "d3-color";
2 import { ScaleLinear, scaleLinear, ScaleOrdinal, scaleOrdinal } from "d3-scale";
3
4 export type ColorSetDefinition = {
5   excludedColors?: string[];
6   assignedColors?: [string, string][];
7   domain?: string[];
8   bandLength?: number;
9 };
10
11 export class ColorManager {
12   // The band of all colors that are available for default use.
13   colorBand: string[] = [];
14
15   // Ordinal scale that maps identifiers to colors.
16   colorScale?: ScaleOrdinal<string, string, never>;
17
18   // The target domain of the color band.
19   domain: string[] = [];
20
21   // Colors the user has explicitly excluded from the color band.
22   excludedColors: HCLColor[] = [];
23
24   // Colors the user has explicitly assigned to a specific identifier.
25   assignedColors: Map<string, string> = new Map();
26
27   // The length of the color band.
28   bandLength = 8;
29
30   constructor(ctx?: ColorSetDefinition) {}
31   init(csd: ColorSetDefinition) {}
32   get state(): ColorSetDefinition {}
33   assignColor(identifier: string, color: string) {}
34   excludeColor(color: string) {}
35   get isInitialized() {}
36   initializeColorBand() {}
37   getBandColor(identifier: string): string {}
38
39   // Conversion functions.
40   static stringToHcl(color: string): HCLColor {}
41   static hclToRgb(color: HCLColor): string {}
42   static stringToHex(color: string): string {}
43
44   // Interpolation functions based on Renderer Context.
45   interpolateColor(ctx: AuthorContributionsContext, value: GizDate): string;
46   interpolateColor(ctx: FileLinesContext, value: Line): string;
47   interpolateColor(ctx: FileMosaicContext, value: Line): string;
48   interpolateColor(ctx: AuthorMosaicContext, value: number): string;
49   interpolateColor(ctx: RendererContext, value: any): string {}
50 }

```

Listing 7.1: A stubbed version of the `ColorManager` class in Gizual, responsible for assigning and distributing evenly spaced HCL colours for all possible domain values based on a `ColorSetDefinition`.



















Author				
		Author One author.one@example.com		123 commits
		Author Two author.two@example.com		120 commits
		Author Three author.three@example.com		99 commits
		Author Four author.four@example.com		95 commits
		Author Five author.five@example.com		90 commits
		Author Six author.six@example.com		80 commits
		Author Seven author.six@example.com		70 commits
		Author Eight author.eight@example.com		60 commits
		Author Nine author.nine@example.com		59 commits

Figure 7.1: Design prototype of the Author Panel for Gizual. It displays the visual encoding of authors based on the HCL colour palette. From left to right, it displays the current colour, avatar, name, email address, and number of commits for each author. [Screenshot created by the author of this thesis.]

Chapter 8

Query Bar


Gizual’s visualisation output can be restricted to a specific selection of files at specific points in the history of a repository. Section 4.4 introduced the query interface and the three different query scopes `commit-range`, `files`, and `visualisation`, which make up the single query object that represents state in Gizual. The Gizual Query Bar, located immediately above the Canvas, provides the UI component to customise the input for these scopes. Figure 8.1 shows the Query Bar in isolation. Figure 5.5 shows it in relation to the full Gizual interface. Each query scope has a corresponding query module in the Query Bar. Internally, JSON [Ecma 2024c] is used to represent queries within Gizual, as described in Section 4.4 and illustrated in Listing 4.5.

8.1 Query Modules

All query scopes are adjustable by the user in the Query Bar. Three query modules, the Time Module, File Module, and Vis Module correspond to the three query scopes `commit-range`, `files`, and `visualisation`, respectively. The shared interface implemented in Maestro ensures that both the Query Bar and the underlying query interface remain synchronised, and both can be extended separately. Figure 8.2 shows the Query Bar with annotated modules.

8.1.1 The Time Module

The Time Module controls the input for the `commit-range` scope. The scope supports two different input combinations:

- Range by Date: A select box provides functionality to select a branch. Two date picker components facilitate the selection of a start date and end date. Additionally, a Timeline Button  toggles the visibility of the Timeline component. This configuration can be seen in Figure 8.3.
- Range by Revision: Two input fields allow the input of two valid Git references, for example a branch name, Git revision, or commit id. This configuration can be seen in Figure 8.4.

Gizual features a dedicated Timeline component to aid in the selection of a valid time range for the `commit-range` scope. Figure 8.5 shows the Query Bar with the Timeline visible. In the Timeline, commits are represented as circles. Selecting a start date and end date is done by dragging a rectangular selection box over the desired commits. Scrolling the mouse wheel zooms the timeline. Dragging while holding the `Shift` key pans the visible section horizontally. A detailed description of this component and a discussion of animation and interaction is provided in Section 9.1.

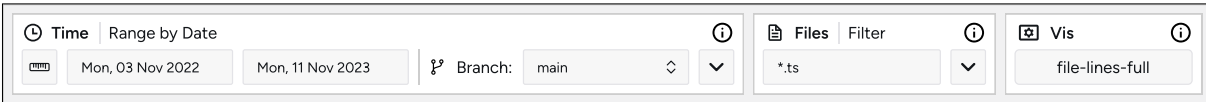


Figure 8.1: The Query Bar allows a user to interact with Gizual’s query interface. [Image created by the author of this thesis.]

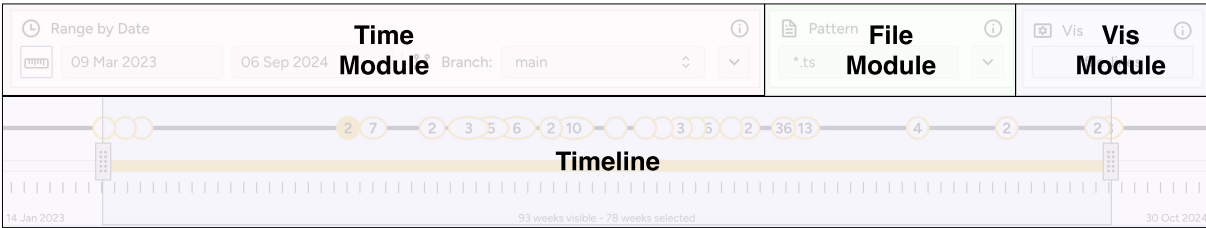


Figure 8.2: The Query Bar comprises three modules. The Time Module controls the selected commit range and includes access to the Timeline component. The File Module controls the selected files. The Vis Module controls the style of the visualisation. [Image created by the author of this thesis.]



Figure 8.3: The Time Module of the Query Bar facilitates the selection of a branch, a start date, and an end date. The user input in this component is used for the commit-range scope. [Image created by the author of this thesis.]



Figure 8.4: The Range by Revision Module of the Query Bar facilitates the selection of two Git revisions (pasted in as free text). [Image created by the author of this thesis.]

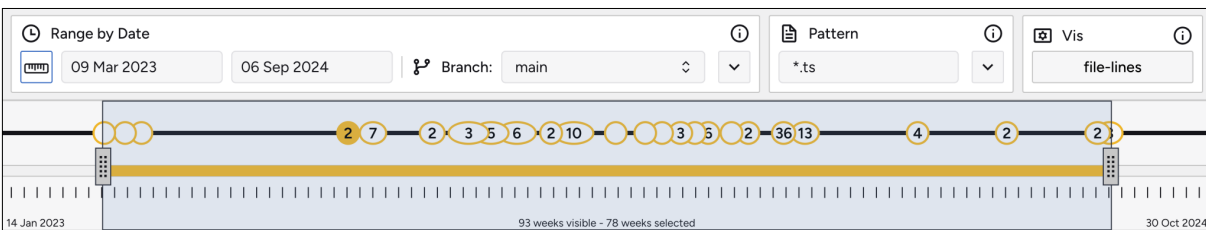


Figure 8.5: The Timeline associated with the Time Module supports the selection of a start date and an end date. Commits are represented as circles, and each tick in the ruler represents a time range of one week. The user input from this component is used for the commit-range scope. [Image created by the author of this thesis.]

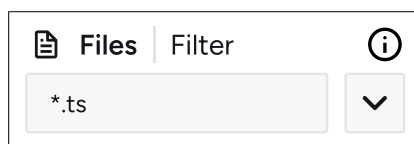


Figure 8.6: The File Module of the Query Bar facilitates the selection of files with a glob pattern. The user input from this component is used for the files scope. [Image created by the author of this thesis.]



Figure 8.7: The `Vis` Module of the Query Bar facilitates the selection of a visualisation type. The user input from this component is used for the `visualisation scope`. [Image created by the author of this thesis.]

8.1.2 The File Module

The File Module controls the input for the `files` scope. This scope supports three different inputs:

- **Pattern:** A path can be specified in a text input field using a Unix-style glob pattern [Kerrisk 2024]. A glob pattern to select all files with the TypeScript extension `.ts` would be: `*.ts`. This configuration uses a regular text input field, as shown in Figure 8.6.
- **File Picker:** Multiple paths can be specified by selecting them with a custom file picker component. The file picker is implemented as a tree view and opens in a separate modal dialogue. Nodes can be individually collapsed and expanded. Entire directories can be selected, and the tree supports partial selection. A detailed explanation of this component is provided in Section 9.2.
- **Revision:** A text input field can be used to enter a valid Git revision (pasted in as free text). All files that were changed in the corresponding Git commit will be selected and loaded.

8.1.3 The Vis Module

The `Vis` Module provides functionality to change the output visualisation type. It is shown in Figure 8.7. Three types of visualisation are currently available: `file-lines`, `file-lines-full`, and `file-mosaic`. Each of these types supports the selection of a visual encoding. The supported encodings are: `Gradient by Age`, and `Palette by Author`. A detailed description of their behaviour can be found in Chapter 7. The selection of type and visual encoding is handled in a separate modal dialogue, shown in Figure 8.8.

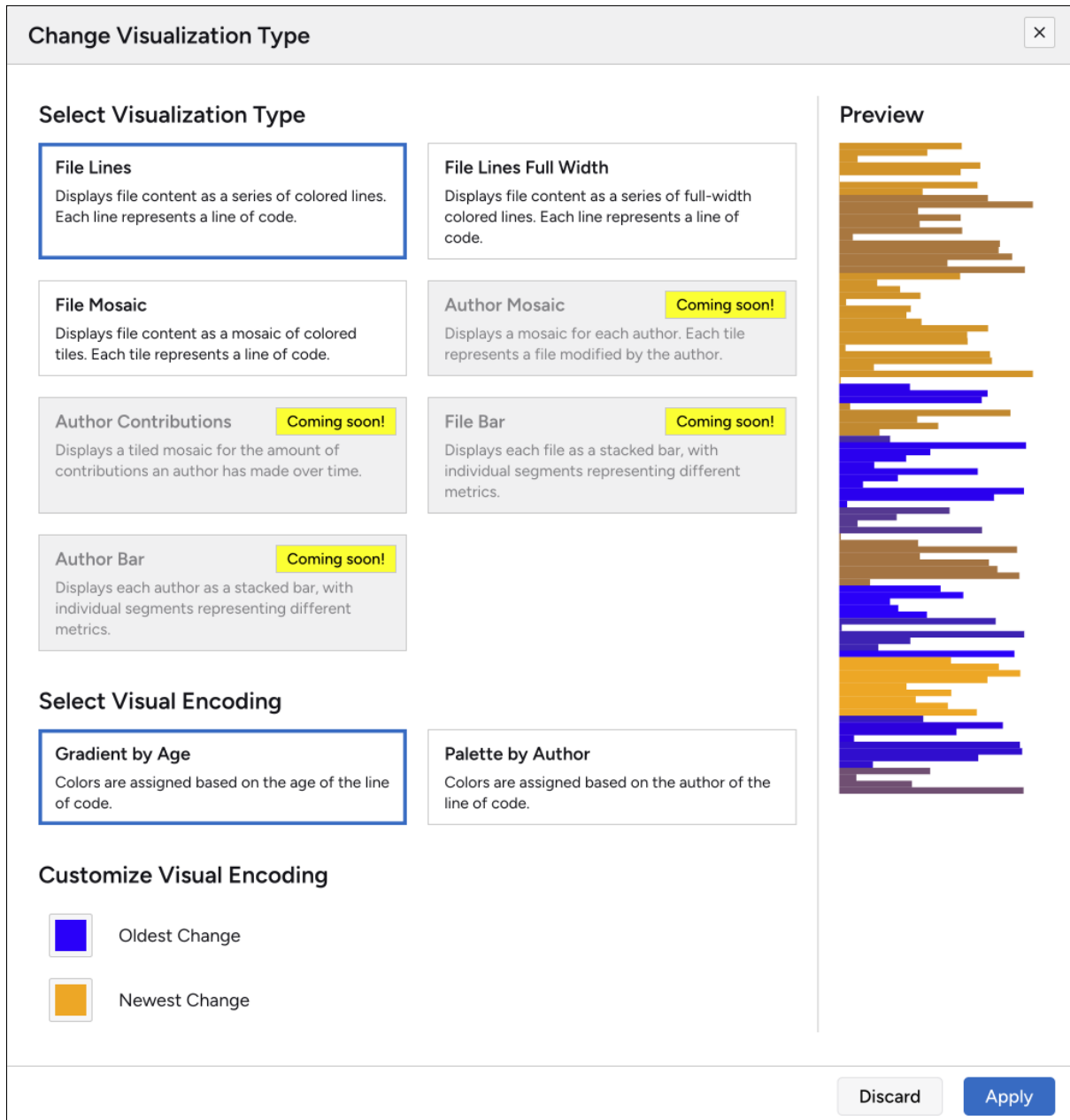


Figure 8.8: The Vis Type Dialogue facilitates the selection of the visualisation type and visual encoding. The visual encoding can be customised with custom colours. A preview section on the right displays randomised demo output for the current settings. [Image created by the author of this thesis.]

```
query-input/  
├── modules/  
│   ├── file/  
│   └── time/  
│       ├── time-base-module.tsx  
│       ├── time-menu.tsx  
│       ├── time-placeholder-module.tsx  
│       ├── time-range-by-date-module.tsx  
│       └── time-range-by-ref-module.tsx  
│   └── type/  
│       ├── base-query-module.tsx  
│       ├── index.ts  
│       ├── modules.module.scss  
│       └── modules.module.scss.d.ts  
├── query-bar/  
│   ├── index.ts  
│   ├── query-bar.module.scss  
│   ├── query-bar.module.scss.d.ts  
│   └── query-bar.tsx  
├── index.ts  
├── module-provider.tsx  
├── query.vm.ts  
└── shared.ts
```

Listing 8.1: The file and directory structure of the Query Bar UI component in Gizual’s source code.

8.2 Implementation Details

The source code for the three query scopes in the Query Bar and their associated modules is structured similarly in terms of files and directories, as can be seen in Listing 8.1. In the source code, the module implementations for the Time Module are contained in the `modules/time/` directory of the Query Bar. The `modules/file/` directory contains the source code for the File Module implementations. Finally, the `modules/type/` directory contains the source code for the Vis Type Module and the corresponding Vis Type Dialog.

The basic shape and interactive elements of each module are derived from the `BaseQueryModule` component, shown in Listing 8.2. All modules are then placed in a `ModuleProvider` component, shown in Listing 8.3. The `ModuleProvider` is responsible for displaying the module that corresponds to the currently selected option within each scope. All modules feature a Swap Button with a downward facing arrow on the right. Clicking the button opens a menu, with which the user can choose a different module for that particular scope. All modules also feature optional tooltips for additional information, and each module can provide as many input elements as necessary for its corresponding scope. This modular setup of components allows for easy customisation of existing query modules, and provides a streamlined way to add new modules in the future.

8.2.1 Query Editor

The horizontal layout of modules works great for devices with wide viewports, but does not scale down well to smaller, narrower devices. To accommodate a wide variety of different viewports, the Query Bar is replaced by the Query Editor on devices with viewport widths of less than 64rem. The Query Editor provides the most commonly used input variants for each scope in a modal dialogue window, shown in Figure 8.9, which is easier to navigate on smaller devices.

```

1 // Imports and types truncated
2 export function BaseQueryModule(props: BaseQueryModuleProps) {
3   const { icon, title, children, hasSwapButton, onSwap, menuItems,
4     containsErrors, hasHelpTooltip, helpContent, hasEditButton,
5     onEdit, editButtonComponent } = props;
6
7   return (
8     <div
9       className={clsx(
10        style.BaseQueryModule,
11        containsErrors && style.ContainsErrors
12      )}
13     aria-expanded={swapMenuOpen}
14   >
15     <div className={style.ColumnContainer}>
16       <div className={style.QueryModuleHeader}>
17         <div className={style.QueryModuleIconWithText}>
18           {icon && <div className={style.QueryModuleIcon}>{icon}</div>}
19           {title && <div className={style.QueryModuleTitle}>{title}</div>}
20         </div>
21         {hasHelpTooltip && (
22           <Tooltip label={helpContent} withArrow>
23             <div>
24               <IconInfo className={style.QueryModuleIcon} />
25             </div>
26           </Tooltip>
27         )}
28       </div>
29       <div className={style.RowContainer}>
30         {children}
31         {hasSwapButton && (
32           <Menu /* Props truncated */>
33             <Menu.Target>
34               <IconButton
35                 className={style.SwapButton}
36                 aria-expanded={swapMenuOpen}
37               >
38                 <IconChevronDown className={style.CloseIcon} />
39               </IconButton>
40             </Menu.Target>
41             {menuItems}
42           </Menu>
43         )}
44         {hasEditButton &&
45           (editButtonComponent ?? (
46             <IconButton onClick={onEdit}>
47               <IconEdit className={style.CloseIcon} />
48             </IconButton>
49           ))}
50       </div>
51     </div>
52   </div>
53 );
54 }

```

Listing 8.2: The `BaseQueryModule` component serves as a base for all derived query modules. It implements the basic layout and provides interactive elements for swapping modules.


```

1 // Imports truncated
2 type ModuleProviderProps = {
3   viewMode: ViewMode;
4 };
5 const ModuleProvider = observer(({ viewMode }: ModuleProviderProps) => {
6   return (
7     <>
8       <TimeModuleProvider viewMode={viewMode} />
9       <FilesModuleProvider viewMode={viewMode} />
10      <PresetModuleProvider viewMode={viewMode} />
11    </>
12  );
13 });
14
15 const TimeModuleProvider = observer(({ viewMode }: ModuleProviderProps) => {
16   const { query } = useQuery();
17   const timeMatch = match(query)
18     .with({ time: P.select() }, (time) => {
19     return match(time)
20       .with({ rangeByDate: P.select() }, () => {
21         return <TimeRangeByDateModule key="time-range-by-date-module" />;
22       })
23       .with({ rangeByRef: P.select() }, () => {
24         return <TimeRangeByRefModule key="time-range-by-ref-module" />;
25       })
26       .with({ sinceFirstCommitBy: P.select() }, () => {
27         return (
28           <TimeSinceFirstCommitByModule key="time-since-first-commit-by-module" />
29         );
30       })
31       .otherwise(() => {
32         return <TimePlaceholderModule key="time-placeholder-module" />;
33       });
34     })
35     .otherwise(() => {
36       return <TimePlaceholderModule key="time-placeholder-module" />;
37     });
38
39   return React.cloneElement(timeMatch, { viewMode });
40 });
41 export {
42   // FilesModuleProvider, (truncated)
43   // PresetModuleProvider, (truncated)
44   TimeModuleProvider,
45   ModuleProvider,
46 };

```

Listing 8.3: The `ModuleProvider` component provides the logic for swapping modules. It is responsible for displaying the correct user input module to the corresponding query state, and contains logic to display a placeholder instead of a module in case of a state mismatch.

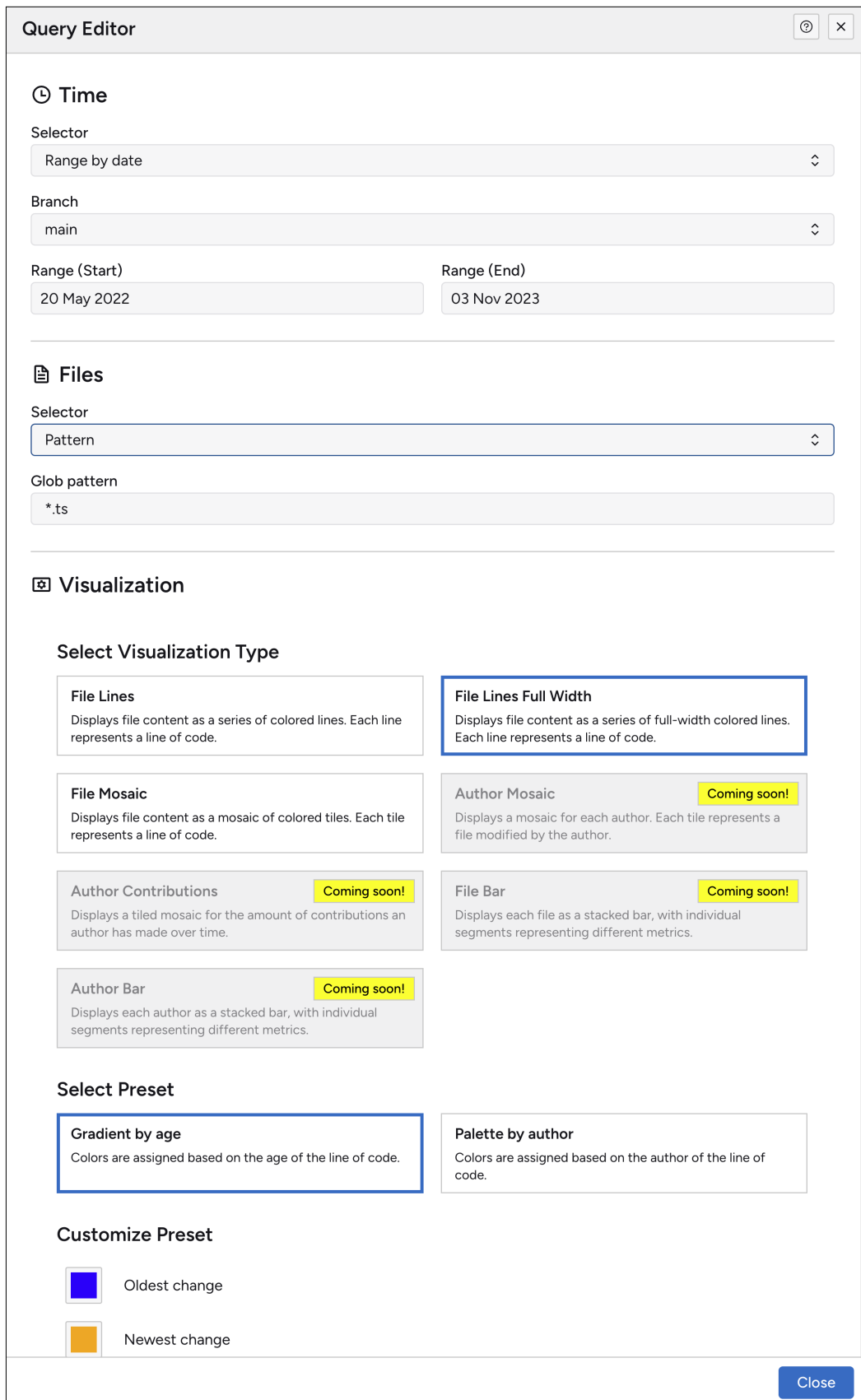


Figure 8.9: The Gizual Query Editor is a modal dialogue window. It replaces the Query Bar on devices with a viewport width smaller than 64rem. [Image created by the author of this thesis.]

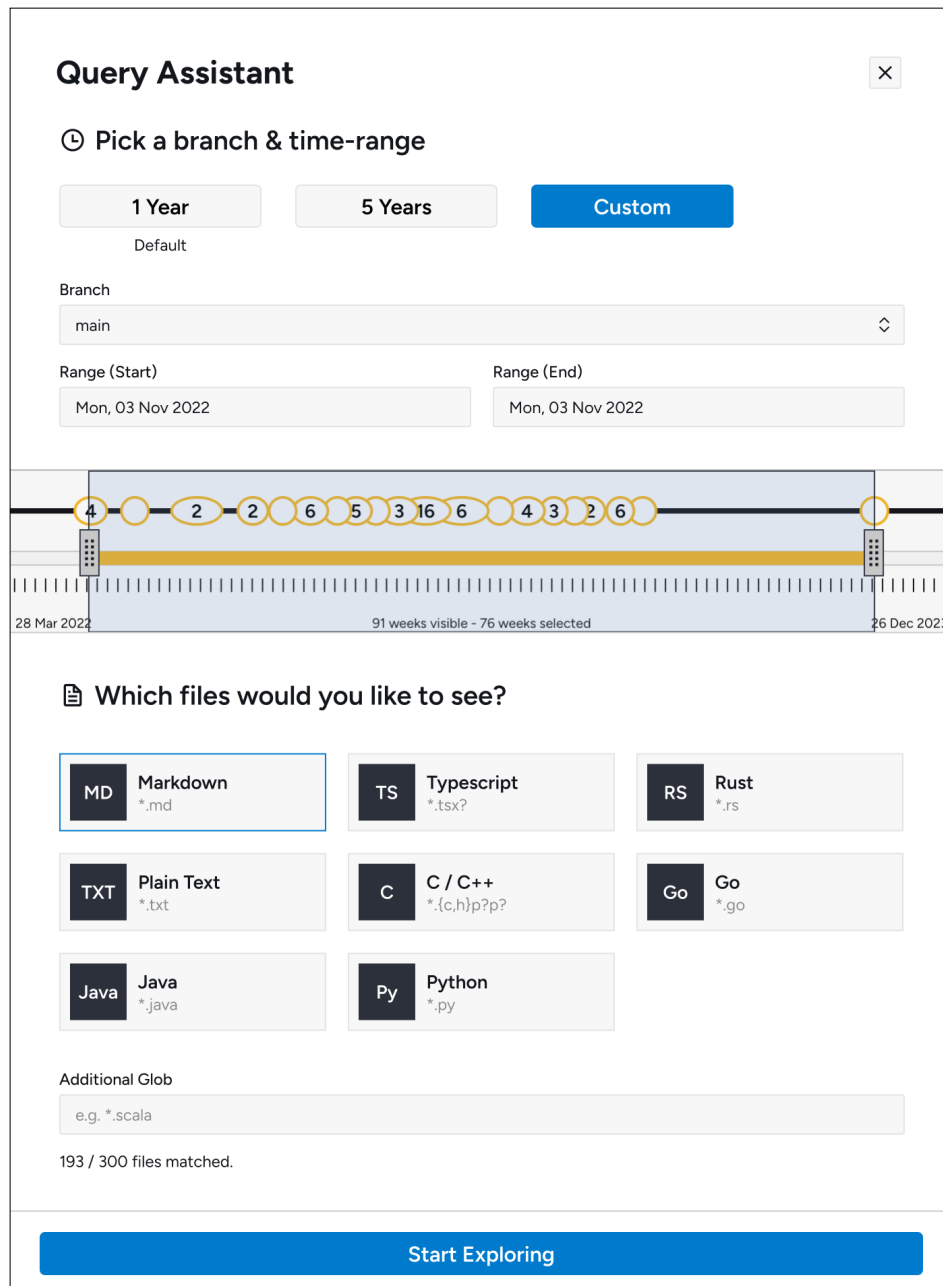


Figure 8.10: Draft of the Query Assistant, which provides a simpler introduction to Gizual’s query interface for new users. [Image created by the author of this thesis.]

8.2.2 Query Assistant

The Query Bar is most useful to experienced users, who already have an idea of what to expect when they launch Gizual. These users can easily customise their desired set of input files and output style with a few clicks. For new users, some modules can be overwhelming. For these users, an additional Query Assistant is proposed, which shows fewer options in a more straight-forward layout. This should help new users obtain a good first visualisation in a matter of seconds. A draft for the proposed dialogue can be seen in Figure 8.10, but its implementation was declared out of scope for this thesis.

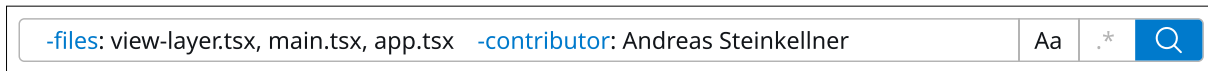


Figure 8.11: Early design prototype of QB1, a single search bar to customise the input query. Query scopes, such as `-files:`, always follow the same syntax. They start with a minus, followed by the name of the scope, and end with a colon. Custom syntax highlighting helps segregate query scopes. [Image created by the author of this thesis.]

8.3 Previous Iterations and Concepts

The Query Bar is a central UI component within Gizual. As such, its usability has been a major concern throughout development. Some earlier concepts, which were eventually discarded in favour of the current implementation, included: 1) a modular query editor, based on a single input element (QB1), 2) a text editor for JSON input called the *Advanced Query Editor* (QB2), and 3) a hybrid approach between modules and a text editor (QB3).

8.3.1 QB1: Single Input Field

This first iteration was largely inspired by the search bars used on sites like GitHub [GitHub 2024b] or Sourcegraph [Sourcegraph 2024]. It was implemented using the CodeMirror [Haverbeke 2024a] library and featured a custom search syntax for chaining commands together. The syntax looked like this:

```
-files:OR(path=["index.html","index.js"],lastEditedBy=["joe"])
```

Each query scope had a defined set of acceptable input patterns. Other inputs would be marked as erroneous and had to manually be fixed by the user. Custom syntax highlighting was introduced in order to visualise the different scopes to the user. Each query scope had to follow a specific input shape: starting with a minus, followed by the name of the scope, and ending with a colon. Figure 8.11 shows an early design prototype of this component.

Eventually, this iteration of the Query Bar was discarded, because the interplay of a shared focus state between the Query Bar and the Timeline proved to be problematic in terms of user experience. When users moved the selection on the Timeline, the Query Bar needed to reflect that change immediately, regardless of the current cursor position. Additionally, all input within the search bar had to be parsed continuously, and errors were hard to pinpoint and resolve for end users.

8.3.2 QB2: Advanced Query Editor

The second iteration of the Query Bar focused on the rigid JSON structure of the query interface and exposed it directly within a code editor. The *Advanced Query Editor* uses the Monaco [Microsoft 2024a] code editor. Besides great support for plugins and styling, Monaco provides advanced source code highlighting defaults and allows simple highlighting based on a predefined JSON schema. Since the query was already written to support a JSON schema, it could easily be integrated into the Monaco editor. The *Advanced Query Editor* is still used within the project for debugging purposes, but is currently only available when Gizual is started in a development environment. Figure 8.12 shows the *Advanced Query Editor* with the default query.

8.3.3 QB3: Hybrid Combination of Modules and JSON Input

QB3 segregated the individual scopes into distinct modules, which could be added or removed to build up the query. Advanced users were supposed to make edits in the *Advanced Query Editor* built in QB2, which was accessible via a button on the right of the Query Bar. A design prototype of QB3 can be seen in Figure 8.13.

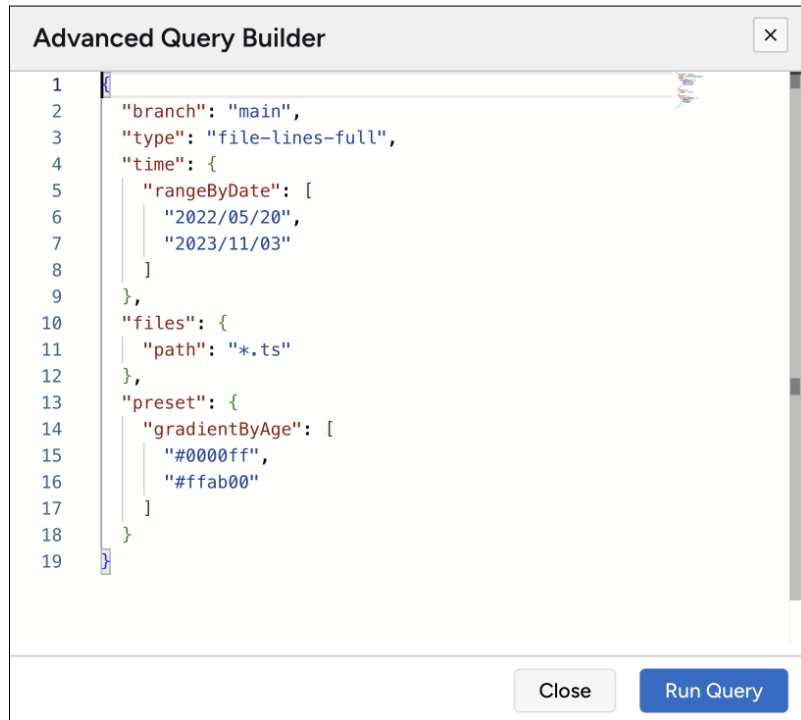


Figure 8.12: The Advanced Query Editor is a live implementation of a Monaco code editor, used to directly edit the JSON query. Initially implemented as QB2, it is now used for debugging and is only available when Gizual is started in a development environment. [Image created by the author of this thesis.]



Figure 8.13: Design prototype of QB3, a hybrid approach with modules and the Advanced Query Editor built in QB2. Modules can be added by clicking the green + button, and removed by clicking the red x button within each module. [Image created by the author of this thesis.]

This iteration was functionally similar to the final implementation of the Query Bar, except for the support of empty modules in QB3, which was removed in the final implementation. Additionally, the Advanced Query Editor was eventually removed from QB3 for regular users, since the implemented query modules became sophisticated enough to represent the entire JSON query appropriately.

Chapter 9

Selected Details of the Implementation

“Details matter, it’s worth waiting to get it right. ”

[Steve Jobs; Co-founder of Apple; 1955-2011]

Gizual’s user interface went through many iterations before arriving at its current state. This chapter provides selected details about some components that were featured, but not fully explained in previous chapters.

9.1 Interactive SVG Timeline

Navigating historical data within a repository requires a visually perceptible way of navigating time. None of the user interface libraries used within the project provides a functionally complete timeline that would fit the requirements of Gizual. Throughout this section, the words “time” and “date” are often used synonymously. Technically, all selections in Gizual are always time-based. From a user perspective, date selection is usually sufficient, and simpler to navigate. The Timeline in Gizual is a custom implementation of an interactive SVG element, transformable through interaction with the mouse or by touch. The component, shown in isolation in Figure 9.1, is split into the following three sections: The Commit Timeline located at the top, the Time Ruler at the bottom, and the Range Selector as an overlay.

9.1.1 Commit Timeline

The Commit Timeline contains all historically relevant commits, displayed as circles on a horizontal line representing infinite time. Some repository activities, such as merge commits, often lead to multiple commits in the same repository with little temporal spacing between them. If two commits are too close to each other, the Timeline visually groups them into an ellipse and displays the number of commits within the group as a label. The width of each ellipse is based on the number of commits it aggregates within a fixed region of influence, based on the current zoom level within the Timeline.

9.1.2 Time Ruler

The Time Ruler contains a visual legend to help users to orient themselves within the visible time range. The distinct visual look of a ruler was replicated to display the distance between days or weeks, depending on the zoom level. Each tick on the ruler represents the start of a new day or week. Additionally, the corners show the earliest and latest date currently visible. A central information text indicates how many days or weeks are currently in view, and how many of them are selected for the visualisation.

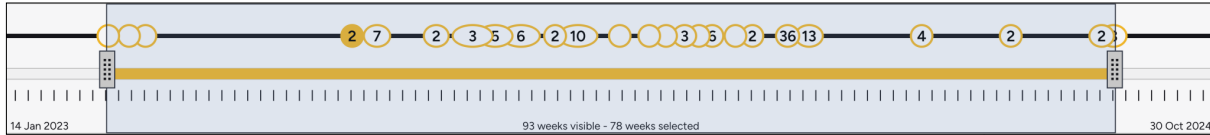


Figure 9.1: The Timeline component in Gizual, associated with the Time Module, supports the selection of a start date and an end date. Commits are represented as circles, and each tick in the ruler represents a time range of one week. The user input from this component is used for the `commit-range` SCOPE. [Image created by the author of this thesis.]

9.1.3 Range Selector

The Range Selector spans the entire Timeline and is positioned on top of it. It implements interactive controls which can be manipulated with the mouse or touch. A rectangular selection box, coloured in light blue, displays the current selection. Handles on the left and right edges can be grabbed to adjust the start or end date of the selection. The entire selection can be moved to a different region by dragging the selection rectangle. Finally, the Range Selector also displays an optional tooltip for each commit. Hovering over any ellipse within the timeline shows a tooltip with information about the contained commits.

The timeline is built as a set of three stacked layers: The base layer contains the logic for the ruler and legend. The commit layer contains the ellipses for the commits within the repository. Finally, the interaction layer contains the interactive elements for capturing user input and displaying the selection rectangle.

The base layer and commit layer are embedded into an SVG element. The SVG element is deliberately three times as wide as it needs to be to display the entire user-selected time range. A CSS transformation is used to transform the component, so that only the user-selected range is visible. The extra rendered width to the left and right are added for performance reasons. They prevent the need for costly re-renders when the user navigates back and forth in time. The ruler element, embedded into the base layer, adapts the spacing and positioning of its ticks depending on the selected start and end time. A visible section of less than 365 days is displayed with ticks on each day. If the visible section spans more than 365 days, the ruler displays a tick for each week.

User interaction is handled through the interaction layer, which is implemented as a regular HTML element outside the SVG. A custom `TimelineEventHandler` class receives all events from the interaction layer, and directly synchronises with the `TimelineViewModel`. The `TimelineViewModel` controls the state of the timeline and is implemented with MobX. The `TimelineEventHandler` defines custom functions for each user input event, and contains optimisation for mouse movements and touch gestures.

9.2 File Tree

Large source code repositories can contain hundreds or even thousands of individual files, nested in directories. None of the user interface component libraries investigated during the implementation could handle this large number of files without stuttering or overloading the main thread. This introduced a need for a custom, high-performance file tree component, which could handle the selection of thousands of files without performance issues.

Gizual's custom File Tree is shown in Figure 9.2. It uses a recursively rendered list of nodes to represent a flat array of file paths as a tree. The array of file paths is obtained from Gizual's data layer after an initial loading period. Algorithmically, the tree is constructed by slicing the set of input paths to create a nested structure. All tree items are then assigned one of three values to represent their state: unchecked, indeterminate (partial selection of children), and checked. The tree items are stored in a flat map, with their paths as a unique identifier. Each item contains a reference to its children and its parent. This

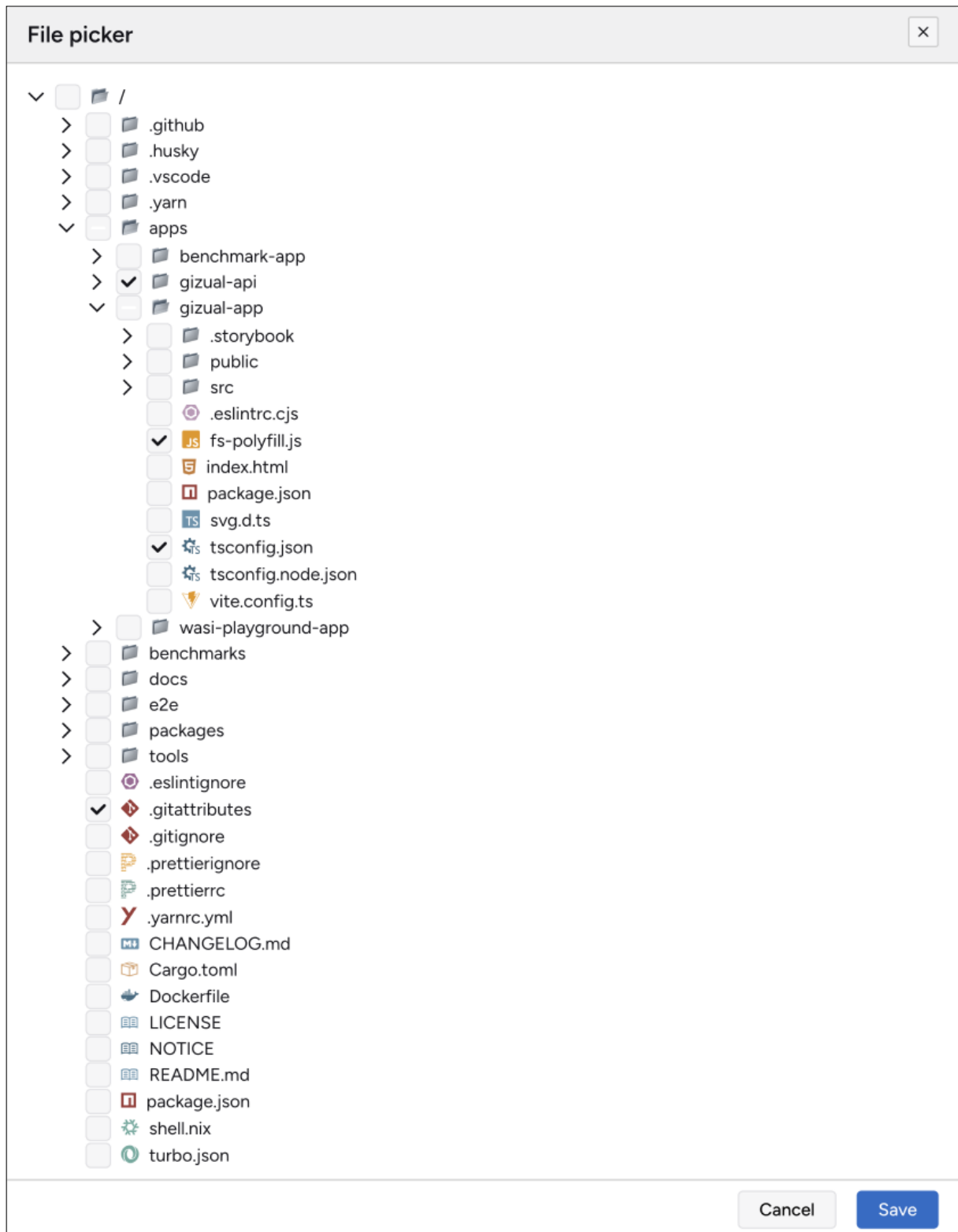


Figure 9.2: The File Tree component in Gizual, associated with the File Module, which supports unchecked, indeterminate (partial selection of children), and checked states for all tree items. [Image created by the author of this thesis.]

approach reduces the lookup time for a specific path, which is useful for propagating the selection state of an entire directory up and down the tree.

Each tree item is then recursively rendered in the corresponding React component, but only children within a defined maximum render depth are added to the DOM. This avoids the performance issues that all other investigated file tree implementations struggled with, by always minimising the number of items in the DOM. Child elements further down in the tree are deliberately only appended to the DOM, when their third ancestor is expanded.

Chapter 10

Outlook and Future Work

Gizual's capabilities still have room to grow. Some features, like the additional visualisation types, already have partial implementations, but were excluded from this thesis. Through the sophisticated split between main thread and web workers, Gizual can handle computationally expensive operations in a streamlined framework, allowing for easy future expansion into more nuanced visualisations or analytics.

The performance of the Canvas could be improved even further by eliminating the need for re-renders during zooming operations. A possible solution for this would be to use SVG elements instead of pre-rendered pixel-based images that require constant re-renders at different resolutions. Gizual already features an SVG rendering backend in the rendering worker, which should allow this feature to be implemented in a reasonable time-frame. This feature would also provide a simple solution to rendering accurate line of code tooltips in the visualisation, since every rendered object could be associated with mouse or touch events.

Similarly, panning in the Timeline could be improved further by replacing the current position update through coordinates with a CSS transform operation, reducing the number of re-renders React needs to compute for this component substantially. Additionally, the Timeline could feature more advanced interactivity when hovering and interacting with commits. A visual indication of commits originating from the same branch, or more advanced grouping and highlighting, come to mind.

Future versions of Gizual could also include more detailed analytics about repository statistics. At the moment, the analytics section only features a bar chart with file extensions, and it is currently disabled. Modern source code repository hosting providers, such as GitHub [GitHub 2024b] and Bitbucket [Bitbucket 2024], provide a variety of analytics about a given repository. These analytics often give insight in distinctive data about a project, such as the number of active contributors, or the general workload or language distribution. While somewhat limited by performance constraints inside the browser sandbox, Gizual could theoretically provide a number of interesting analytics, including:

- *Commit Activity Over Time (Line Chart)*: Displays the number of commits in the repository as a line chart. Data is aggregated over days, weeks, or months.
- *Commit Activity (Bar Chart)*: Displays the number of commits for the hour of day, or the day of the week, as a bar chart.
- *Code Additions and Deletions (Line Chart)*: Displays the additions and deletions for a chosen user as a line chart. Data is aggregated over days, weeks, or months.
- *Top Modified Files (Bar Chart)*: Displays the files which were most often modified, as a bar chart. The length of each bar corresponds to the number of modifications in the corresponding source file.

Other potential future improvements might include:

- With regard to visual encoding, a binning feature would allow users to select a specific range of time within their selection, and update the colours for that specific range separately.
- A brushing feature would also be highly useful. It would allow users to hover over a specific line of code in the Canvas, and all other lines of code that were changed within the same commit would also be highlighted automatically.
- A treemap view of the hierarchical structure of a repository, like the one in Spider Sense [N. H. Reddy et al. 2015], might be a useful addition to Gizual.
- The SVG export of the visualisation currently only renders the tiles and strips for each line of code. The text of each line of code is not included in order to save space. In the future, the user should be able to decide whether to include text or not.

Finally, web applications heavily depend upon the feature set provided by the browser engine. Major updates to the available feature set could eventually lead to a breaking change. Tools like Electron [Electron 2024] and Tauri [Tauri 2024] allow web applications to be bundled into an executable package, which can be installed and run natively. This would insulate Gizual from such breaking changes in the future.

Chapter 11

Concluding Remarks

“ In any series of elements to be controlled, a selected small fraction, in terms of numbers of elements, always accounts for a large fraction in terms of effect. ”

[Vilfredo Pareto; Italian engineer, economist and teacher; 1848–1923]

Crafting a data visualisation application that is both functionally complete and aesthetically pleasing to use is incredibly challenging. Many iterations were required to arrive at the final result, as presented in this thesis. Very often, it felt like parts of the application were almost complete, but the final polish required to make functionality seamless to use was very difficult to achieve.

The project goal was to find a novel solution to combine the ease-of-use of a web application with the performance and capabilities of native code and harness it for data visualisation. During implementation, it often became apparent that some implementation details require strategic, out-of-the-box thinking. Combining experimental browser features with a traditional visualisation style has been incredibly rewarding, and it demonstrates the power of the web as an application platform. As web browsers keep expanding their support for more advanced and computationally expensive operations, the web will remain an interesting target for future information visualisation applications.

This thesis presented Gizual, an open-source, web-based visualisation tool for exploring Git source code repositories. Powerful input mechanisms were combined into the Gizual Query Bar, so the user can configure the set of selected files and type of visualisation. An interactive SVG-based Timeline component supports the visual selection of a commit range. Files can be selected with a custom File Tree, which can cope with many thousands of nested files. Two kinds of visual encoding allow the user to explore lines of code by last modification date or by most recent author. The Gizual Canvas arranges file tiles in a masonry layout, with a synchronised minimap to maintain orientation within the freely zoomable visualisation canvas. The custom renderer achieves high performance by distributing rendering tasks across a pool of web workers, and is capable of generating the file tiles both as raster images for the visualisation canvas and as SVGs for exporting. Finally, the entire Gizual application is designed responsively, so it can adapt to a variety of viewport sizes and end user devices.

The source code of Gizual is open-source and is available on GitHub [Schintler and Steinkellner 2024]. A deployed version can be accessed at gizual.com.

Appendix A

User Guide

Gizual's interface was created to be as simple and intuitive as possible. Nevertheless, some components and workflows deserve additional explanation. This guide is not a complete overview of Gizual as an application, but serves as an entry point for new users, who want to get started as quickly as possible.

A.1 Opening a Repository

Since Gizual is a web application, no installation or manual setup on the user device is necessary. The application can be opened by visiting `app.gizual.com`, or by cloning the Gizual repository [Schintler and Steinkellner 2024] and building it from source locally. Once the application has loaded, the Gizual welcome screen is displayed, as shown in Figure A.1. The welcome screen provides the interface to open a Git repository in Gizual. Repositories can be loaded from a local directory, from GitHub [GitHub 2024b], or from a set of featured repositories.

A.2 Navigating the User Interface

After a repository has been loaded, Gizual's main user interface is displayed. The interface is split into five main regions, as shown in Figure A.2:

- **Canvas:** A central canvas acts as the heart of the visualisation. Individual tiles represent files, and coloured strips represent lines of code. The colour-coding is based either on the age of the line of code, or the author of the line of code. Tiles are laid out in a masonry grid, and the entire area is interactive. Users can navigate around by zooming, panning, or pinching.
- **Query Bar:** The Query Bar above the Canvas provides all functionality relating to file selection and customisation of the visualisation.
- **Toolbar:** The Toolbar on the left provides quick access to navigation functionality, so that users without a mouse-wheel or people with special needs can navigate within the Canvas.
- **Sidebar:** The Sidebar on the right provides a custom Minimap, which mirrors the content of the main canvas in an abstracted, global overview. Both components are synchronised, so that any movement within either one of them is immediately replicated in the other. Additionally, the Legend component at the bottom of the Sidebar provides an overview of the visualisation colour-coding and controls to change it directly.
- **Status Bar:** The Status Bar provides feedback about ongoing operations and current resource usage. It displays the number of selected files, and the current usage of explorer workers and renderer workers.

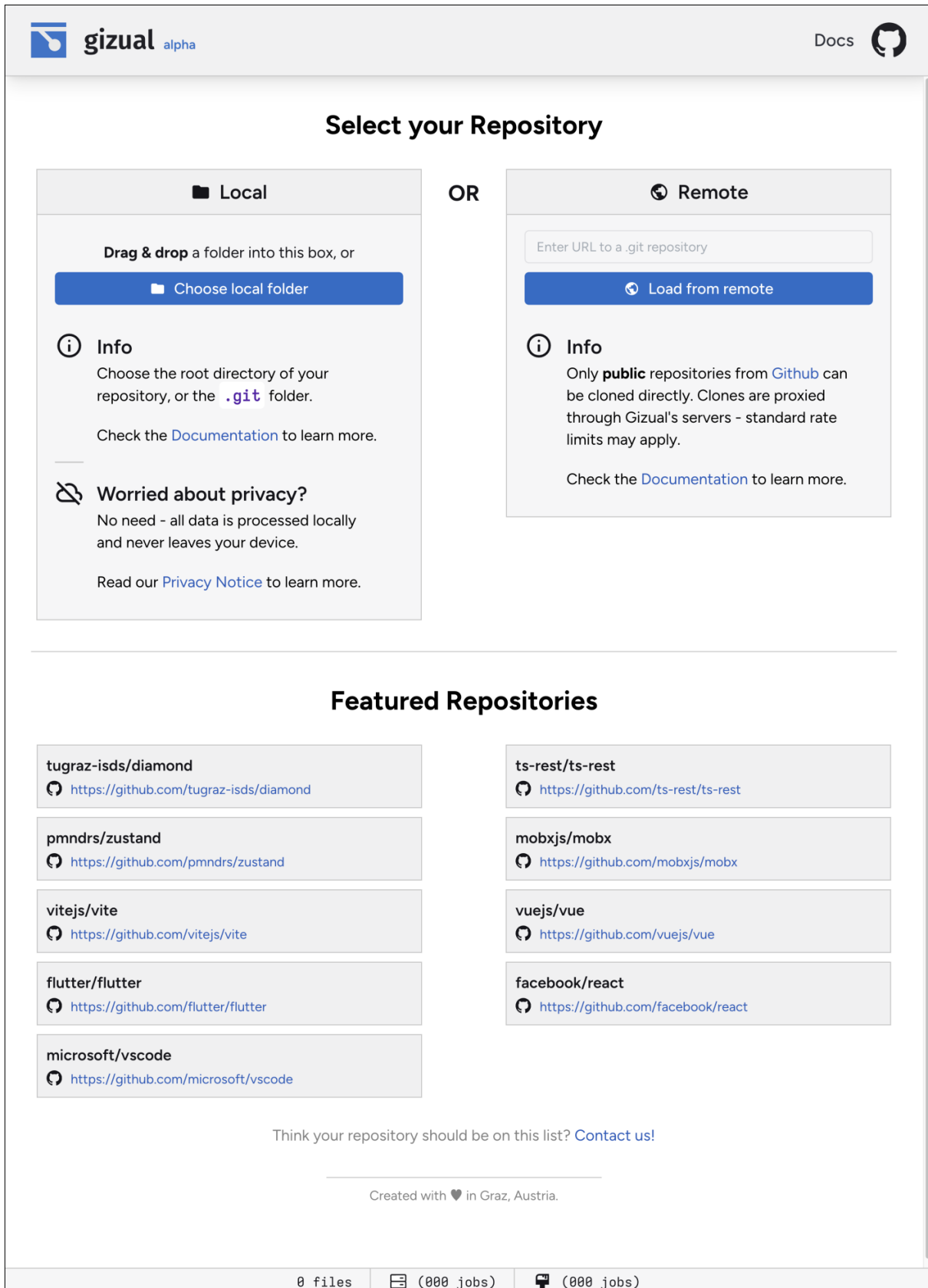


Figure A.1: The Gizual welcome screen. Repositories can be loaded from a local directory, from a remote URL on GitHub, or from a set of featured repositories. [Screenshot created by the author of this thesis.]



Figure A.2: The user interface of Gizual is separated into five main regions. The central Canvas provides the visualisation of files in the repository. The Query Bar at the top controls file selection. The Toolbar on the left provides quick access to navigation functionality. The Sidebar to the right provides an abstracted overview of the entire visualisation canvas. Finally, the Status Bar at the bottom provides feedback about ongoing operations and current resource usage. [Screenshot created by the author of this thesis.]

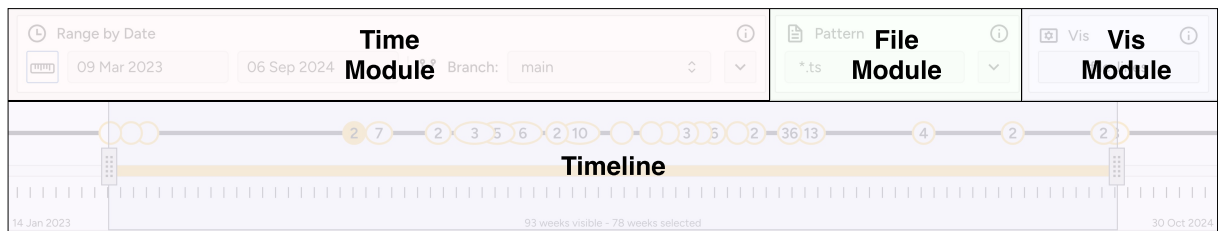


Figure A.3: The Query Bar allows a user to interact with Gizual’s query interface. The Time Module controls the selected commit range and includes access to the Timeline component. The File Module controls the selected files. The Vis Module controls the style of the visualisation. [Image created by the author of this thesis.]

A.3 Modifying the Query

The query in Gizual determines the set of files in a repository to be visualised and how they are visualised. The query can be modified by interacting with the Query Bar at the top of the main Gizual window. It contains all the functionality required to customise the range and style of the main visualisation. The bar is split into three main modules: Time Module, File Module, and Vis Module, as can be seen in Figure A.3. The Time Module also gives access to the Timeline component.

A module in Gizual’s Query Bar represents a structured way to enter data which is translated into Git exploration commands, and leads to a valid output visualisation. Different modules are available for the same scope, where a scope represents a block of information in the query. All modules follow the same basic layout structure. The title field at the top with the associated icon displays information about the current module. An information icon on the top right can be hovered to show a tooltip about this specific module. The Swap Button at the right of every module can be used to swap the current module with a different input module for the same scope.

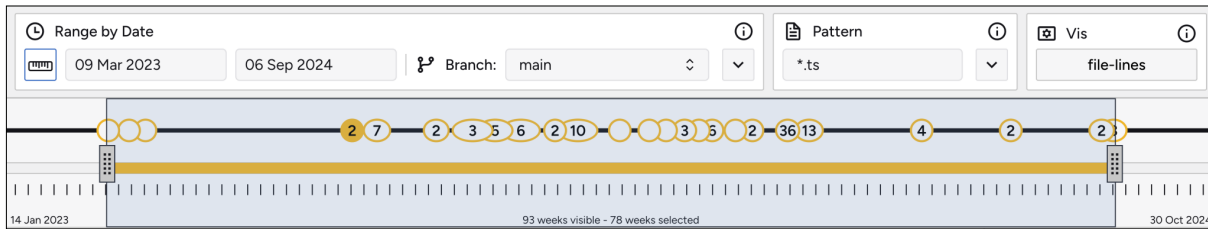



Figure A.4: The Time Module and its associated Timeline supports the selection of a start date and an end date. Commits are represented as circles, and each tick in the ruler represents a time range of one week. The user input from this component is used for the `commit-range` scope. [Image created by the author of this thesis.]

A.3.1 Customising the Commit Range

The range of commits can be adjusted by specifying a desired start and end time in the Time Module, as shown in Figure A.4. The two date fields open up a date picker. Alternatively, the Timeline can be opened by clicking the Timeline Button . If desired, a specific branch can be selected in the dropdown select box on the right.

The Timeline can be used to visually customise the commit range in a quick and intuitive way. It visualises time on a horizontal ruler, and represents commits as circles on that timeline. If too many commits are in close proximity, they automatically merge into ellipses. Hovering over a commit displays corresponding metadata. The selected range is displayed as a rectangle with a light blue background. Handles on the left and right of the selection rectangle can be used to modify the start and end date of the selection. Dragging anywhere in the selected range moves the entire selection. On desktop devices, holding down the `Shift` button while scrolling the mouse wheel triggers a panning motion, moving the viewbox of the Timeline horizontally. This movement can also be achieved by dragging with the middle mouse button. On touch devices, dragging with one finger moves the selection box. Two-finger pinching motions zoom on the timeline, while a three-finger drag pans the entire timeline horizontally.

A.3.2 Customising Selected Files

To customise the set of files to visualise, a valid glob pattern can be entered into the provided input field. This glob pattern follows common Unix pattern rules. A simple example would be to match all files with a certain extension, say `*.ts`.

A.3.3 Customising the Visualisation

The style and output of the visualisation can be configured by clicking the button in the Vis Module, which brings up the Visualisation Type Dialogue, shown in Figure A.5. In this modal dialogue window, a visualisation type can be chosen at the top, and a visual encoding (colour-coding) can be chosen at the bottom. The available visualisation types are:

- File Lines: Displays file contents as a series of coloured lines. Each line represents one line of code.
- File Lines Full Width: Displays file contents as a series of full-width coloured lines. Each line represents one line of code.
- File Mosaic: Displays file contents as a mosaic of coloured boxes. Each box represents one line of code.

After a visualisation type has been selected, one of two visual encodings can be chosen:

- Gradient By Age: Colours are assigned based on the age of the line of code. The end points of the colour gradient can be chosen by picking a start colour and an end colour.

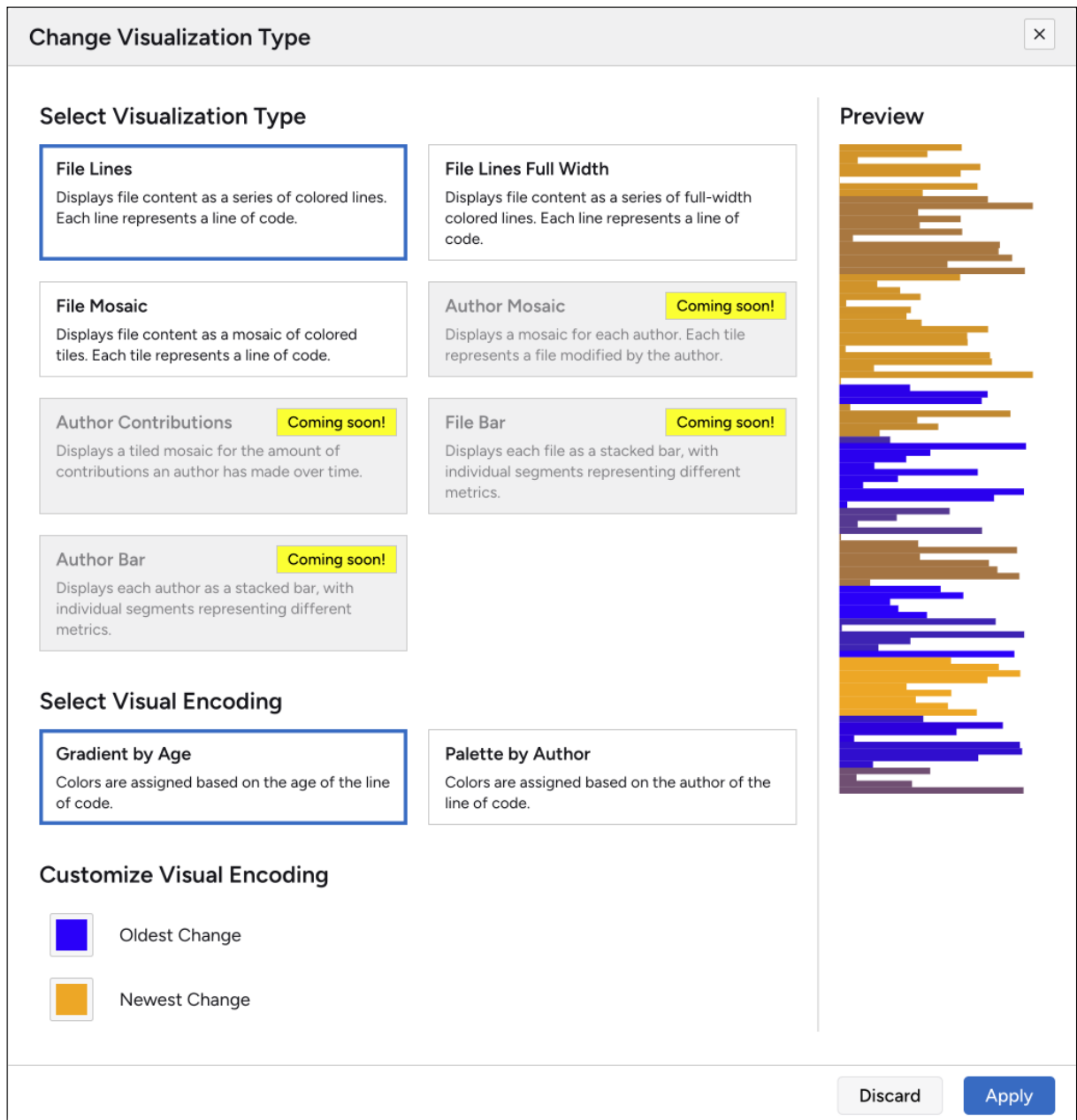





Figure A.5: The Visualisation Type Dialogue allows a visualisation type and visual encoding to be chosen. [Image created by the author of this thesis.]

- Palette By Author: Colours are assigned based on the author of the line of code. The colour value for each author can be customised in the Author Panel, which is shown on the right of the canvas when this encoding type is selected.

A.4 Canvas Navigation

Once the query has been adjusted as desired, the main visualisation on Gizual’s Canvas can be explored. Figure A.6 shows the Gizual Canvas with a File Lines visualisation and Gradient by Age visual encoding. Each of the selected files is represented by a single tile, and all files are laid out in a masonry grid. The canvas can be zoomed smoothly using the mouse wheel, pinch-zoom, or the Zoom In Button  and Zoom Out Button  in the Toolbar on the left side. The Reset View Button  resets the size and position of the visualisation to

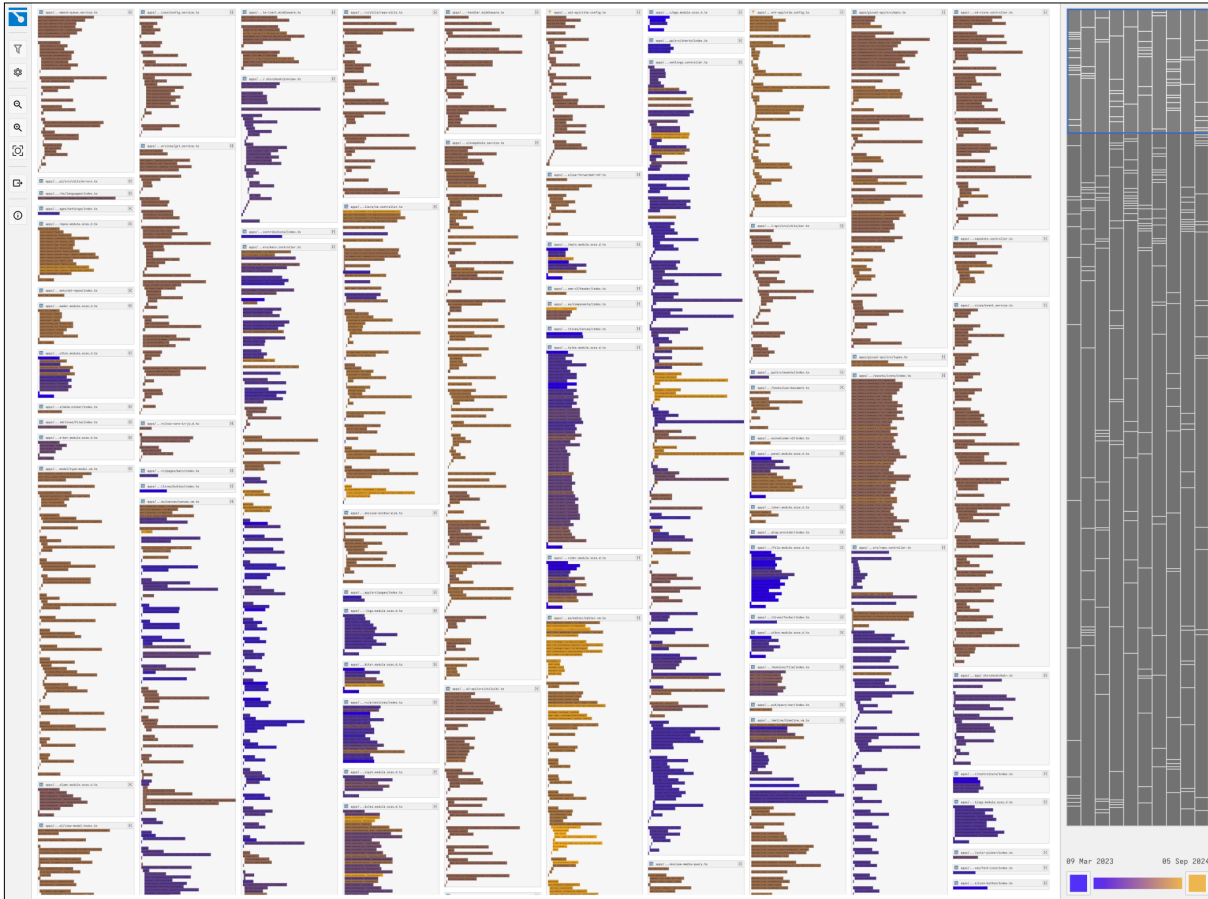


Figure A.6: The Gizual Canvas with a File Lines visualisation and Gradient by Age visual encoding. It provides an interactive viewbox to the entire visualisation canvas. Each tile represents a file of source code. Tiles are arranged in a masonry layout. Lines of code are colour-coded. The Toolbar on the left contains various controls. The Sidebar on the right contains the Minimap and Legend. [Image created by the author of this thesis.]

its default state.

To the right of the Canvas, the Sidebar contains the Minimap and Legend. The Minimap shows the entire visualisation canvas in simplified form. A blue rectangle indicates the current viewbox, and can be dragged to quickly pan across the entire visualisation canvas. The Minimap and Canvas components are synchronised and update bidirectionally. The bottom section of the Sidebar displays the Legend. The colour gradient displays the interpolated range of colours used in a Gradient by Age visual encoding. The start and end colours can be quickly adjusted by clicking on the coloured buttons at the endpoints. The corresponding start and end dates are shown above the colour endpoints.

With the Palette by Author visual encoding selected, the Legend is hidden, and the Author Panel is shown on the right. This configuration can be seen in Figure A.7.

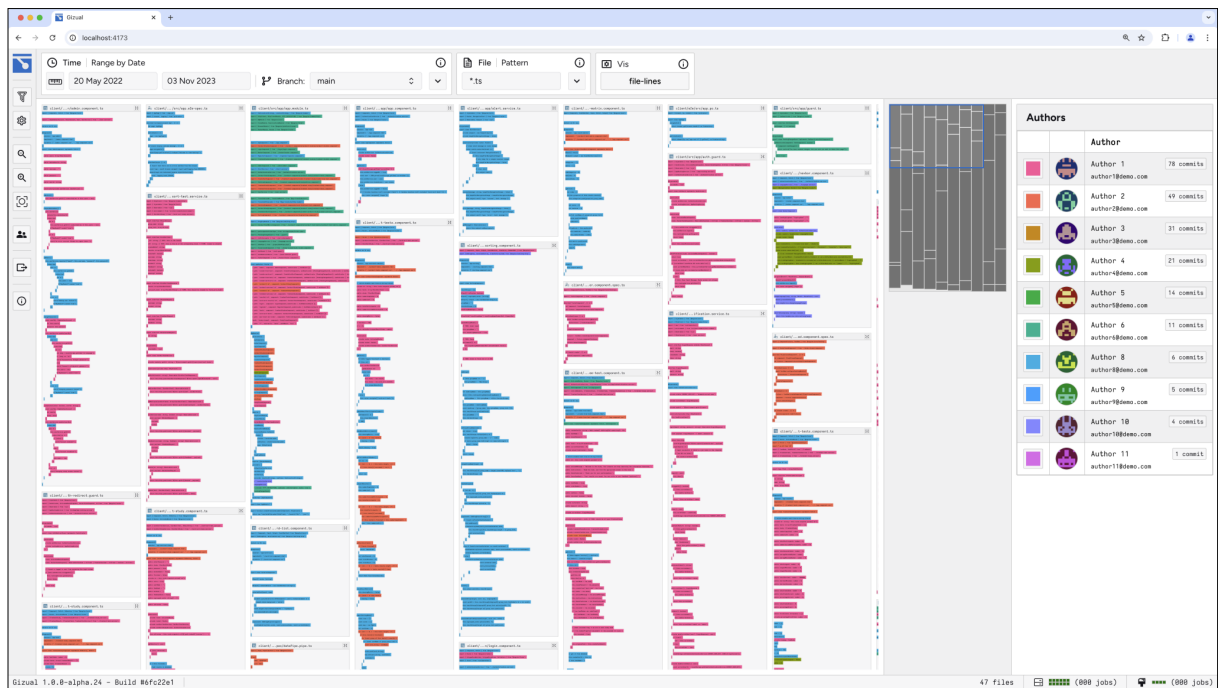


Figure A.7: The Gizual Canvas with a File Lines visualisation and Palette by Author visual encoding. The Author Panel on the right of the Canvas displays the authors of the selected commits. Avatars are anonymised for this example. [Image created by the author of this thesis.]

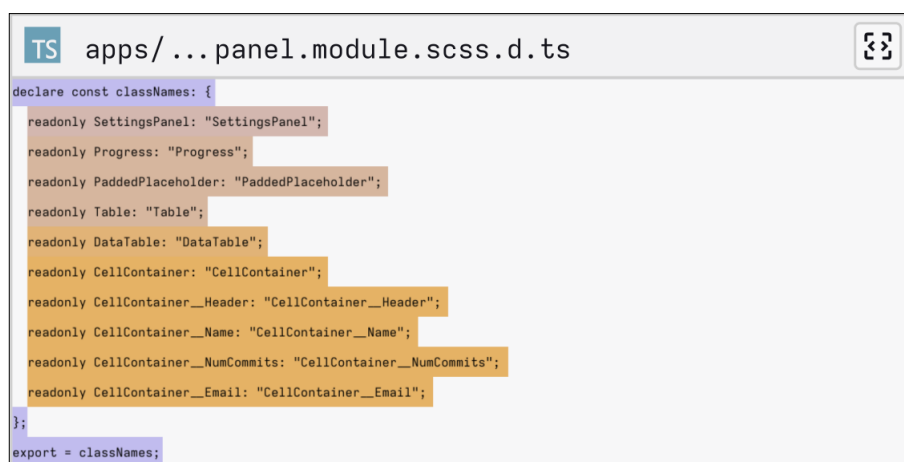



Figure A.8: A tile for a single file in Gizual. The icon on the left of the title bar indicates the file's extension. The file's path is truncated in the middle to display the start and end of the path. The Source Button on the top right opens the Source Editor. [Image created by the author of this thesis.]




Figure A.9: The Source Editor displays the content of a file. A coloured decoration to the left of each line shows the blame information using the current visual encoding. A tooltip displays information about the author and commit date. [Image created by the author of this thesis.]

A.5 Inspecting Individual Files

Each individual file within the visualisation is displayed as a rectangular tile. Each tile has a title bar, displaying the extension icon, file path, and Source Button , as shown in Figure A.8. The Source Button can be used to open the file in Gizual's integrated read-only Source Editor in a separate modal dialogue window, as shown in Figure A.9. A coloured decoration is prepended to each line of code, with the same colour-coding as the main visualisation. Hovering over the coloured decoration displays a tooltip with information about the author and commit date.

A.6 Export

The Export Button  in the Toolbar on the left side of the main window can be used to export the current visualisation as an SVG file. To reduce the size of the SVG file, this export only renders the tiles and strips for each line of code, the text of each line of code is not included.

Appendix B

Developer Guide

This guide explains the structure of the Gizual code base and the development and build processes in the Gizual project. It is aimed at developers who might wish to modify or extend Gizual. The most current version with general contribution guidelines is available online in the Gizual repository [Schintler and Steinkellner 2024].

This appendix was written jointly by Stefan Schintler and Andreas Steinkellner.

B.1 Development Stack

The Gizual project requires the following dependencies to be installed on the local development system:

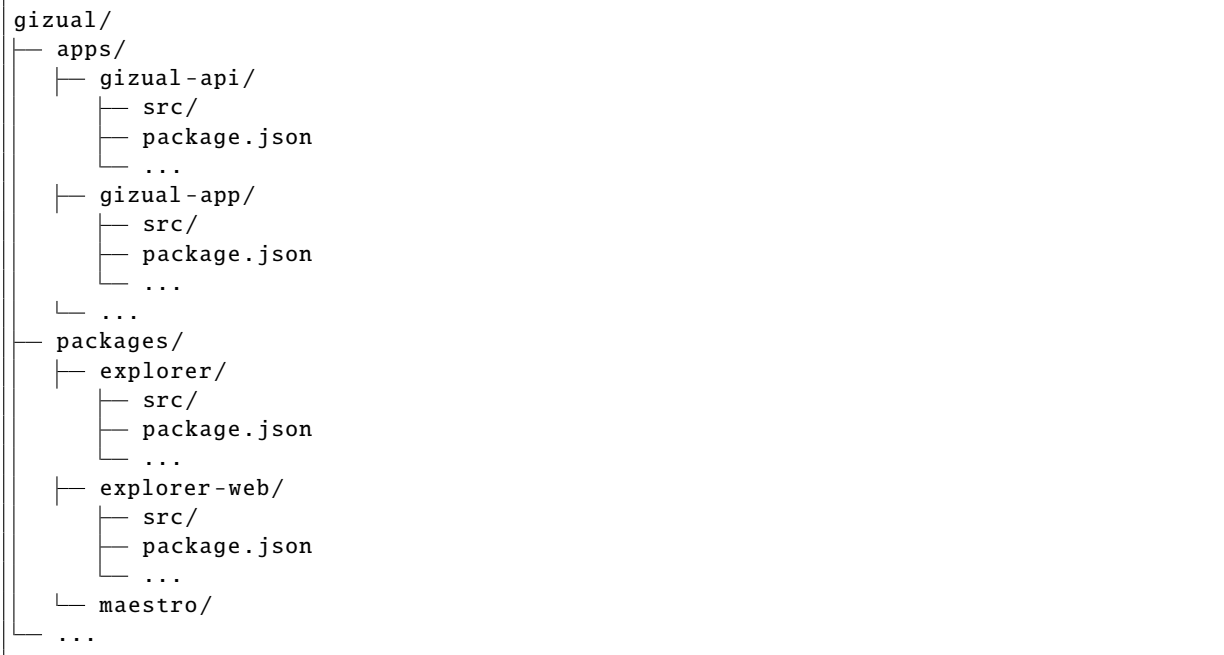
- *Node.js v18 or higher*: Used for building and bundling the application and running the server-side code [OpenJS 2024b].
- *Yarn Berry v3.5.0*: Used as a package manager because of its workspace support [yarn 2024a].
- *Rust v1.79.0 or higher*: Used for building the native parts of Gizual [RF 2024].
- *Git*: Used for version control of the source code [Git 2024].

B.2 Project Structure

Gizual uses the workspace feature of yarn [yarn 2024c] to group packages and their dependencies into self-contained folders. Although technically self-contained, these workspaces can co-depend on other workspaces within the repository. Listing B.1 shows the structure of the Gizual code repository.

B.2.1 Gizual API

The Gizual API workspace is located in the `apps/gizual-api/` folder and contains the source code for the server-side service, which provides a proxy for cloning Git repositories. This is necessary since GitHub and other Git hosting services do not allow cross-origin requests (CORS) [MDN 2024d] from external domains. However, since Gizual also offers the ability to explore local Git repositories, this service is entirely optional. It is written in TypeScript, based on the Express v4 framework [OpenJS 2024a], and makes use of the `simple-git` [King 2024] package to interact with Git hosting services.



Listing B.1: The overall structure of Gizual’s source code repository, showing the most important workspaces. Each workspace defines its dependencies in its own `package.json` file.

B.2.2 Gizual App

The Gizual App workspace is located in the `apps/gizual-app/` folder and contains the source code for the frontend web application. It is written in TypeScript and based on the React framework [Meta 2024b]. For state management, it uses the MobX [MobX 2024] library. Most of its UI components are based on the Mantine [Mantine 2024b] library, and component styles are written in SCSS. This package aggregates all other dependencies and serves as the main build target of the Gizual application. Building and bundling is done with Vite [Vite 2024b].

B.2.3 Maestro

The Maestro workspace is located in the `packages/maestro/` folder and contains the source code for the main controller of the application. It is written in TypeScript and designed as the main gateway between the user interface and the data layer. It is responsible for managing global application state and data processing. A detailed description of this package can be found in Section 4.2.

B.2.4 Explorer

The Explorer workspace is located in the `packages/explorer/` folder and contains the source code for the `git-explorer` module. Its web bindings are located the `packages/explorer-web/` folder. The Explorer is written in Rust [RF 2024] and is based on the `libgit2` [libgit2 2024] library. This module is responsible for exploring local Git repositories and providing a list of files and their blame information. It is used by the Gizual App to display the visualisation via the Maestro package. It also provides the pooling mechanism to use multiple instances of explorer workers in parallel. The package exposes a simple TypeScript API to interact with the underlying native module.

B.2.5 Renderer

The Renderer workspace is located in the `packages/file-renderer/` folder and contains the source code for the file visualisation module. Developed in TypeScript, it uses the `OffscreenCanvas` API to render images off the main thread, enabling parallel processing. It contains rendering functions for each available visualisation type, by leveraging the blame information provided by the Explorer. Communication between the Gizual App and the Renderer is handled via Maestro.

B.3 Data Flow

Gizual depends on many interconnected components, running in separate threads. Figure B.1 provides a structural overview of the interconnected packages and their communication pathways. Data flow in Gizual begins with the `updateQuery()` function, which is executed on initial load, and then each time the user modifies the query. This triggers Maestro to update its states. Maestro determines the set of selected files off the main thread. It estimates the file lengths and generates information for each visualisation tile. This information is then passed back to main thread and triggers a MobX state update. This state is used to render the visualisation tiles in the Gizual App.

A corresponding component is created for each tile. Each component is coupled with a block in Maestro, and an attached `IntersectionObserver` keeps track of its position in the viewport. This information is provided to Maestro, which in turn determines if a re-render of this tile is necessary. Each time a tile is rendered, Maestro passes the cached blame information to the renderer pool and executes the rendering function. If a tile's blame information is not yet cached, it is retrieved from the explorer pool first. The rendered image is stored in an Object URL within the browser environment, and its URL is passed back to Maestro, which in turn triggers a MobX reaction to update the block's state and image. This two-way reactive binding between the tile representation on the canvas and its internal state in Maestro is the foundation of Gizual's data flow.

B.4 Build and Deploy

The Gizual project consists of multiple distinct build targets, which all need to be built in the correct order. Usually, web-based applications rely on a single build and bundling tool like Vite. However, the Rust-based parts of Gizual require a more complex build process, which cannot easily be represented in conventional build scripts, especially with hot module reloading during development.

To address this issue and streamline the overall build process, the Gizual project contains its own designated build tool, called `please`. This tool is based on yarn workspaces and allows a workspace to reference local dependencies, input source files, and necessary build commands. This information is defined within the `package.json` file of each workspace. Based on this information and a target workspace, `please` automatically traverses all referenced workspaces recursively and builds a dependency graph. Recursive dependencies are detected and resolved. This graph is then used to determine the correct order of building and bundling. The building process for each workspace is only executed, if the source files of the corresponding workspace have changed.

To simplify the deployment process, Gizual is bundled into a single docker image. After `please` has built the application, the build artefacts are copied into the docker image. The docker image is then pushed to a container registry and is ready to be deployed.

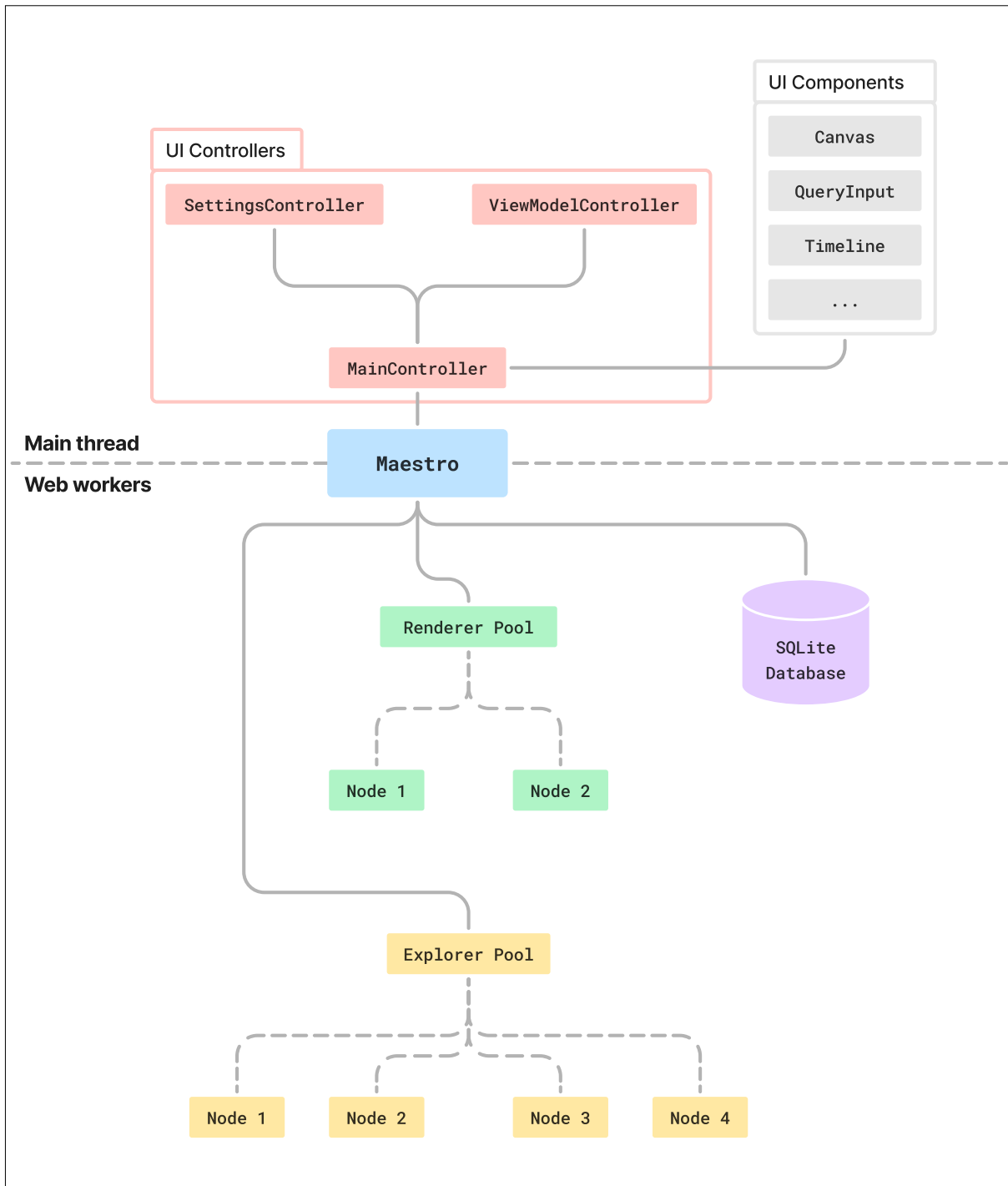


Figure B.1: The software architecture of Gizual. UI controllers and UI components are instantiated in the main thread. The explorer pool, renderer pool, and SQLite database are executed asynchronously in web workers inside the browser. The Maestro provides a unified interface across the different realms. [Diagram created by the authors of this chapter.]

Bibliography

- Abuaddous, Hayfa Y., Mohd Zalisham, and Nurlida Basir [2016]. *Web Accessibility Challenges*. International Journal of Advanced Computer Science and Applications 7.10 (01 Oct 2016), pages 172–181. ISSN 2156-5570. doi:10.14569/IJACSA.2016.071023 (cited on page 14).
- Adepu, Sushma and Rachel F. Adler [2016]. *A Comparison of Performance and Preference on Mobile Devices vs. Desktop Computers*. Proc. 7th IEEE Annual Ubiquitous Computing, Electronics, & Mobile Communication Conference (UEMCON 2016) (Columbia University, New York, USA). Oct 2016, pages 1–7. doi:10.1109/UEMCON.2016.7777808 (cited on page 26).
- Aggarwal, Sanchit [2018]. *Modern Web-Development Using ReactJS*. International Journal of Recent Research Aspects 5.1 (01 Mar 2018), pages 133–137. ISSN 2349-7688. <https://ijrra.net/Vol5issue1/IJRRRA-05-01-27.pdf> (cited on page 22).
- Agosti, Maristella, Nicola Ferro, Pamela Forner, Henning Müller, and Giuseppe Santucci [2013]. *Information Retrieval Meets Information Visualization*. Volume 7757. Lecture Notes in Computer Science. Springer, 2013. ISBN 3642364144. doi:10.1007/978-3-642-36415-0 (cited on page 3).
- Alam, Omar and Jörg Kienzle [2012]. *Designing with Inheritance and Composition*. Proc. 3rd International Workshop on Variability and Composition (VariComp 2012) (Potsdam, Germany). ACM, 26 Mar 2012, page 19. doi:10.1145/2161996.2162002 (cited on page 22).
- Alkharabsheh, Khalid, Yania Crespo, Esperanza Manso, and José A. Taboada [2018]. *Software Design Smell Detection: A Systematic Mapping Study*. Software Quality Journal 27.3 (27 Oct 2018), pages 1069–1148. ISSN 0963-9314. doi:10.1007/s11219-018-9424-8 (cited on page 20).
- Andrew, Rachel [2020]. *Why Are Some Animations Slow?* web.dev, 06 Oct 2020. <https://web.dev/articles/animations-overview#pipeline> (cited on page 16).
- Andrews, Keith [2021]. *Writing a Thesis: Guidelines for Writing a Master’s Thesis in Computer Science*. Graz University of Technology, Austria. 10 Nov 2021. <https://ftp.isds.tugraz.at/pub/keith/thesis/> (cited on page xiii).
- Andrews, Keith [2024]. *Information Visualisation: Course Notes*. 08 Mar 2024. <https://courses.isds.tugraz.at/ivis/ivis.pdf> (cited on page 1).
- Angular [2024a]. *Angular*. Google, 11 Jun 2024. <https://angular.dev/> (cited on pages 21, 23).
- Angular [2024b]. *AngularJS Version Support Status*. Google, 29 Jun 2024. <https://docs.angularjs.org/misc/version-support-status> (cited on page 23).
- Angular [2024c]. *Incremental DOM*. Angular, 29 Jun 2024. <https://google.github.io/incremental-dom/> (cited on page 23).
- Ant Design [2024]. *Ant Design*. Ant Group and Ant Design Community, 04 Jul 2024. <https://ant.design/> (cited on page 46).

- Bardas, Alexandru G. [2010]. *Static Code Analysis*. Journal of Information Systems & Operations Management 4.2 (2010), pages 99–107. <http://rebe.rau.ro/RePEc/rau/jisomg/WI10/JISOM-WI10-A10.pdf> (cited on page 20).
- Behrisch, Michael, Michael Blumenschein, Nam Wook Kim, Lin Shao, Mennatallah El-Assady, Johannes Fuchs, Daniel Seebacher, Alexandra Diehl, Ulrik Brandes, Hanspeter Pfister, Tobias Schreck, Daniel Weiskopf, and Daniel A. Keim [2018]. *Quality Metrics for Information Visualization*. Computer Graphics Forum 37.3 (10 Jul 2018), pages 625–662. doi:10.1111/CGF.13446 (cited on page 4).
- Berg, Maggie [2012]. *CS Undergrad Wins Tech Fellowship*. The Brown Daily Herald. 09 Sep 2012. <https://browndailyherald.com/article/2012/09/cs-undergrad-wins-tech-fellowship> (cited on page 44).
- BetterTyped [2024]. *react-zoom-pan-pinch*. 13 Nov 2024. <https://github.com/BetterTyped/react-zoom-pan-pinch> (cited on page 60).
- Bitbucket [2024]. *Bitbucket*. Atlassian, 30 Jun 2024. <https://bitbucket.org/> (cited on pages 7, 83).
- Bjørner, Dines [2009]. *Role of Domain Engineering in Software Development – Why Current Requirements Engineering Is Flawed!* Proc. 7th International Conference on Perspectives of Systems Informatics (PSI 2009) (Novosibirsk, Russia). Springer, 15 Jun 2009, pages 2–34. doi:10.1007/978-3-642-11486-1_2 (cited on page 3).
- Bogner, Justus and Manuel Merkel [2022]. *To Type or Not to Type? A Systematic Comparison of the Software Quality of JavaScript and TypeScript Applications on GitHub*. Proc. 19th International Conference on Mining Software Repositories (MSR '22) (Pittsburgh, Pennsylvania, USA). ACM, 23 May 2022. doi:10.1145/3524842.3528454. <https://arxiv.org/abs/2203.11115> (cited on pages 20–21).
- Bostock, Michael, Vadim Ogievetsky, and Jeffrey Heer [2011]. *D3: Data-Driven Documents*. IEEE Transactions on Visualization and Computer Graphics 17.12 (23 Oct 2011), pages 2301–2309. doi:10.1109/TVCG.2011.185. <https://idl.cs.washington.edu/files/2011-D3-InfoVis.pdf> (cited on page 64).
- Bostock, Mike [2024a]. *d3*. 07 Jul 2024. <https://d3js.org/> (cited on page 64).
- Bostock, Mike [2024b]. *d3 Linear Scales*. 07 Jul 2024. <https://d3js.org/d3-scale/linear> (cited on page 64).
- Bostock, Mike [2024c]. *d3 Ordinal Scales*. 07 Jul 2024. <https://d3js.org/d3-scale/ordinal> (cited on page 64).
- BrowserStack [2024]. *BrowserStack*. BrowserStack, 30 Jun 2024. <https://browserstack.com/> (cited on page 27).
- Buering, Thorsten, Jens Gerken, and Harald Reiterer [2006]. *User Interaction with Scatterplots on Small Screens - A Comparative Evaluation of Geometric-Semantic Zoom and Fisheye Distortion*. IEEE Transactions on Visualization and Computer Graphics 12.5 (Sep 2006), pages 829–836. ISSN 1941-0506. doi:10.1109/TVCG.2006.187 (cited on page 26).
- Caudwell, Andrew H. [2010]. *Gource: Visualizing Software Version Control History*. Proc. ACM International Conference Object Oriented Programming Systems, Languages and Applications (OOPSLA '10) (Reno, Nevada, USA). 17 Oct 2010, pages 73–74. doi:10.1145/1869542.1869554 (cited on page 7).
- Chen, Min and Luciano Floridi [2013]. *An Analysis of Information Visualisation*. Synthese 190.16 (Nov 2013), pages 3421–3438. ISSN 0039-7857. doi:10.1007/s11229-012-0183-y (cited on page 4).
- Dijkstra, Edsger W. [1974]. *On the Role of Scientific Thought*. In: *Selected Writings on Computing: A Personal Perspective*. Edited by David Gries. EWD 447. Springer, 30 Aug 1974. ISBN 0387906525. doi:10.1007/978-1-4612-5695-3_12. <https://cs.utexas.edu/~EWD/transcriptions/EWD04xx/EWD447.html> (cited on page 32).

- Dur, Banu İnanç Uyan [2014]. *Data Visualization and Infographics in Visual Communication Design Education at The Age of Information*. *Journal of Arts and Humanities* 3.5 (May 2014), pages 39–50. ISSN 2167-9045. doi:10.18533/journal.v3i5.460. <https://theartsjournal.org/index.php/site/article/view/460> (cited on page 4).
- Duru, Hacı Ali, Murat Perit Çakır, and Veysi İşler [2013]. *How Does Software Visualization Contribute to Software Comprehension? A Grounded Theory Approach*. *International Journal of Human-Computer Interaction* 29.11 (Nov 2013), pages 743–763. ISSN 1044-7318. doi:10.1080/10447318.2013.773876 (cited on page 9).
- Ecma [2024a]. *Ecma*. 11 Jun 2024. <https://ecma-international.org/> (cited on page 19).
- Ecma [2024b]. *ECMAScript*. 11 Jun 2024. <https://ecma-international.org/technical-committees/tc39/> (cited on page 19).
- Ecma [2024c]. *Introducing JSON*. 29 Oct 2024. <https://json.org/> (cited on page 67).
- Eden, Brad [2005]. *Information Visualization*. *Library Technology Reports* 5.1 (Jan 2005), pages 7–17. doi:10.5860/ltr.41n1. <https://journals.ala.org/index.php/ltr/article/view/4596/5424> (cited on page 3).
- Eick, Stephen G. [1994]. *Graphically Displaying Text*. *Journal of Computational and Graphical Statistics* 3.2 (Jun 1994), pages 127–142. ISSN 1061-8600. doi:10.2307/1390665. <https://www.cs.kent.edu/~jmaletic/softvis/papers/eick1994.pdf> (cited on page 5).
- Eick, Stephen G. and Joseph L. Steffen [1992]. *Visualizing Code Profiling Line Oriented Statistics*. Proc. 3rd IEEE Conference on Visualization (Vis '92) (Boston, Massachusetts, USA). 19 Oct 1992, pages 210–217. doi:10.1109/VISUAL.1992.235206 (cited on page 5).
- Eick, Stephen G., Joseph L. Steffen, and Eric E. Sumner Jr. [1992]. *Seesoft: A Tool for Visualizing Line Oriented Software Statistics*. *IEEE Transactions on Software Engineering* 18.11 (Nov 1992), pages 957–968. doi:10.1109/32.177365. <http://www.sdml.cs.kent.edu/library/Eick92.pdf> (cited on pages xiii, 5–6, 55).
- Electron [2024]. *Electron*. 09 Dec 2024. <https://electronjs.org/> (cited on page 84).
- esbuild [2024a]. *esbuild*. 30 Jun 2024. <https://esbuild.github.io/> (cited on page 24).
- esbuild [2024b]. *esbuild Performance Benchmark*. 30 Jun 2024. <https://esbuild.github.io/faq/#benchmark-details> (cited on page 25).
- Feiner, Johannes and Keith Andrews [2018]. *RepoVis: Visual Overviews and Full-Text Search in Software Repositories*. Proc. IEEE Working Conference on Software Visualization (VISSOFT 2018) (Madrid, Spain). Sep 2018, pages 1–11. doi:10.1109/VISSOFT.2018.00009. <https://ftp.isds.tugraz.at/pub/papers/feiner-vissoft2018-repovis.pdf> (cited on pages xiii, 7–8).
- Figma [2024]. *Figma*. 04 Jul 2024. <https://figma.com/> (cited on page 44).
- Gechev, Minko [2024]. *Angular v18 is Now Available!* Google, 22 May 2024. <https://blog.angular.dev/angular-v18-is-now-available-e79d5ac0affe> (cited on page 23).
- Geoco, Tommy, Taylor Palmer, and Jordan Bowman [2023]. *2023 Design Tools Survey*. 05 Dec 2023. <https://uxtools.co/survey/2023/ui-design> (cited on page 44).
- Git [2024]. *Git*. 11 Jun 2024. <https://git-scm.com/> (cited on page 95).
- GitHub [2024a]. *AngularJS Releases*. Angular, 29 Jun 2024. <https://github.com/angular/angular.js/releases?after=v0.9.4> (cited on page 23).
- GitHub [2024b]. *GitHub*. 30 Jun 2024. <https://github.com/> (cited on pages 7, 76, 83, 87).

- GitLab [2024]. *GitLab*. 30 Jun 2024. <https://gitlab.com/> (cited on page 7).
- Harris, Rich [2016]. *Frameworks Without the Framework: Why Didn't We Think of This Sooner?* 26 Nov 2016. <https://svelte.dev/blog/frameworks-without-the-framework> (cited on page 23).
- Harris, Rich [2018]. *The Virtual DOM is Pure Overhead*. 27 Dec 2018. <https://svelte.dev/blog/virtual-dom-is-pure-overhead> (cited on page 24).
- Haverbeke, Marijn [2024a]. *CodeMirror*. 28 Oct 2024. <https://codemirror.net/> (cited on page 76).
- Haverbeke, Marijn [2024b]. *Eloquent JavaScript*. 4th Edition. 2024. ISBN 1593279507. https://eloquentjavascript.net/Eloquent_JavaScript.pdf (cited on page 32).
- Hawes, Nathan, Stuart Marshall, and Craig Anslow [2015]. *CodeSurveyor: Mapping Large-Scale Software to Aid in Code Comprehension*. Proc. 3rd IEEE Working Conference on Software Visualization (VISSOFT 2015) (Bremen, Germany). Sep 2015, pages 96–105. doi:10.1109/VISSOFT.2015.7332419 (cited on page 9).
- hclwizard [2024]. *hclwizard*. University of Innsbruck, 06 Jul 2024. <https://hclwizard.org/> (cited on page 63).
- Hu, Miao [2022]. *Information Visualization Design in the Environment of New Media Based on Computer Technology*. Proc. International Conference on Knowledge Engineering and Communication Systems (ICKECS 2022) (Chikkaballapur, Karnataka, India). 29 Dec 2022, pages 1–4. doi:10.1109/ICKECS5652.3.2022.10059871 (cited on page 4).
- Huang, Cary, Peter Ruetter, Dave Caruso, Ben Plate, and James O'Loughlin [2024]. *Scale of the Universe*. 04 Jul 2024. <https://scaleofuniverse.com/> (cited on page 53).
- IBM [2024]. *What is Data Visualization?* 29 Jun 2024. <https://ibm.com/topics/data-visualization> (cited on page 4).
- Ibrus, Indrek [2013]. *Evolutionary Dynamics of Media Convergence: Early Mobile Web and its Standardisation at W3C*. Telematics and Informatics 30.2 (May 2013), pages 66–73. ISSN 0736-5853. doi:10.1016/j.tele.2012.04.004 (cited on page 11).
- IETF [2024]. *IETF | Internet Engineering Task Force*. Internet Engineering Task Force, 10 Jun 2024. <https://ietf.org/> (cited on page 11).
- Ihaka, Ross [2003]. *Colour for Presentation Graphics*. Proc. 3rd International Workshop on Distributed Statistical Computing (DSC 2003) (Vienna, Austria). 20 Mar 2003. <https://stat.auckland.ac.nz/~ihaka/courses/787/color.pdf> (cited on page 63).
- Irish, Paul and Tali Garsiel [2011]. *How Browsers Work*. web.dev, 05 Aug 2011. https://web.dev/articles/howbrowserswork#global_and_incremental_layout (cited on page 16).
- Jerding, Dean and John Stasko [1998]. *The Information Mural: A Technique for Displaying and Navigating Large Information Spaces*. IEEE Transactions on Visualization and Computer Graphics 4.3 (03 Jul 1998), pages 257–271. ISSN 1077-2626. doi:10.1109/2945.722299 (cited on page 55).
- Kaur, Rupinder and Jyotsna Sengupta [2011]. *Software Process Models and Analysis on Failure of Software Development Projects*. International Journal of Scientific & Engineering Research 2.2 (Feb 2011). <https://arxiv.org/abs/1306.1068> (cited on page 3).
- Kerrisk, Michael [2024]. *glob(7) - Linux Manual Page*. 15 Jun 2024. <https://man7.org/linux/man-pages/man7/glob.7.html> (cited on page 69).
- Kim, Bohyun [2013]. *Responsive Web Design, Discoverability, and Mobile Challenge*. Library Technology Reports 49.6 (Aug 2013), pages 29–39. <https://journals.ala.org/index.php/ltr/article/viewFile/4507/5286> (cited on page 27).

- King, Steve [2024]. *Simple Git*. 19 Nov 2024. <https://npmjs.com/package/simple-git> (cited on page 95).
- Knight, Claire and Malcolm Munro [1999]. *Comprehension with[in] Virtual Environment Visualisations*. Proc. 7th IEEE International Workshop on Program Comprehension (ICPC 1999) (Pittsburgh, Pennsylvania, USA). 05 May 1999, pages 4–11. doi:10.1109/WPC.1999.777733 (cited on page 5).
- Korduba, Yaryna, Stefan Schintler, and Andreas Steinkellner [2022]. *Gizual - Repository Visualization for Git*. 706.057 Information Visualisation SS 2022 Project Report. Graz University of Technology, 07 Jul 2022. <https://gizual.com/resources/gizual-paper-ss2022.pdf> (cited on page 51).
- Kulkarni, Vinay and Sreedhar Reddy [2003]. *Separation of Concerns in Model-Driven Development*. IEEE Software 20.5 (Sep 2003), pages 64–69. ISSN 0740-7459. doi:10.1109/MS.2003.1231154. <https://citeseerx.ist.psu.edu/document?doi=5b27c112f72fa5bc8e6f412412674ac0968a3c0b> (cited on page 32).
- LambdaTest [2024]. *LambdaTest*. LambdaTest, 30 Jun 2024. <https://lambdatest.com/> (cited on page 27).
- Lee, Bongshin, Petra Isenberg, Nathalie Henry Riche, and Sheelagh Carpendale [2012]. *Beyond Mouse and Keyboard: Expanding Design Considerations for Information Visualization Interactions*. IEEE Transactions on Visualization and Computer Graphics 18.12 (Dec 2012), pages 2689–2698. ISSN 1941-0506. doi:10.1109/TVCG.2012.204 (cited on page 26).
- Li, Hairong [2011]. *The Interactive Web: Toward a New Discipline*. Journal of Advertising Research 51.1, 50th Anniversary Supplement (01 Mar 2011), pages 13–26. ISSN 0021-8499. doi:10.2501/JAR-51-1-013-026 (cited on page 53).
- libgit2 [2024]. *libgit2*. <https://libgit2.org/> (cited on pages 34, 96).
- Lidwell, William, Kritina Holden, and Jill Butler [2023]. *Universal Principles of Design*. 3rd Edition. Rockport, 23 May 2023. 424 pages. ISBN 076037516X (cited on page 43).
- Linberg, Kurt R. [1999]. *Software Developer Perceptions About Software Project Failure: a Case Study*. Journal of Systems and Software 49.2–3 (Dec 1999), pages 177–192. ISSN 0164-1212. doi:10.1016/S0164-1212(99)00094-1 (cited on page 3).
- Louridas, Panos [2006]. *Static Code Analysis*. IEEE Software 23.4 (Jul 2006), pages 58–61. ISSN 0740-7459. doi:10.1109/MS.2006.114 (cited on page 20).
- Manovich, Lev [2011]. *What is visualisation?* Visual Studies 26.1 (Mar 2011), pages 36–49. ISSN 1472-586X. doi:10.1080/1472586X.2011.548488. http://manovich.net/content/04-projects/064-what-is-visualization/61_article_2010.pdf (cited on page 3).
- Mantine [2024a]. *Mantine*. Mantine Contributors, 04 Jul 2024. <https://mantine.dev/> (cited on page 46).
- Mantine [2024b]. *Mantine*. 19 Nov 2024. <https://npmjs.com/package/@mantine/core> (cited on page 96).
- Marcus, Andrian, Louis Feng, and Jonathan I. Maletic [2003]. *3D Representations for Software Visualization*. Proc. 2003 ACM Symposium on Software Visualization (SoftVis '03) (San Diego, California, USA). 11 Jun 2003, pages 27–36. doi:10.1145/774833.774837 (cited on page 5).
- Martins, Beatriz and Carlos Duarte [2023]. *A Large-Scale Web Accessibility Analysis Considering Technology Adoption*. Universal Access in the Information Society 23 (Jul 2023), pages 1857–1872. ISSN 1615-5289. doi:10.1007/s10209-023-01010-0 (cited on page 14).
- Mattila, Anna-Liisa, Ihantola Petri, Terhi Kilamo, Antti Luoto, Mikko Nurminen, and Heli Väättäjä [2016]. *Software Visualization Today: Systematic Literature Review*. Proc. 20th International Academic Mindtrek Conference (Mindtrek'16) (Tampere, Finland). ACM, 17 Oct 2016, pages 262–271. doi:10.1145/2994310.2994327 (cited on page 5).
- Matuzovic, Manuel [2022]. *Lost in Translation*. 02 May 2022. <https://beyondtellerrand.com/events/dusseldorf-2022/speakers/manuel-matuzovic> (cited on page 14).

- McGhee, Geoff [2010]. *Journalism in the Age of Data*. 07 Sep 2010. <https://vimeo.com/14777910> (cited on page 1).
- MDN [2024a]. *ARIA*. MDN Web Docs, 10 Jun 2024. <https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA> (cited on page 14).
- MDN [2024b]. *Cascade Layers*. MDN Web Docs, 10 Jun 2024. https://developer.mozilla.org/en-US/docs/Learn/CSS/Building_blocks/Cascade_layers (cited on page 16).
- MDN [2024c]. *Client-side tooling overview*. MDN Web Docs, 29 Jun 2024. https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Understanding_client-side_tools/Overview (cited on page 24).
- MDN [2024d]. *Cross-Origin Resource Sharing (CORS)*. MDN Web Docs, 19 Nov 2024. <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS> (cited on page 95).
- MDN [2024e]. *CSS Masonry Layout*. MDN Web Docs, 01 Dec 2024. https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_grid_layout/Masonry_layout (cited on pages 55, 60).
- MDN [2024f]. *CSS Object Model (CSSOM)*. MDN Web Docs, 10 Jun 2024. https://developer.mozilla.org/en-US/docs/Web/API/CSS_Object_Model (cited on page 17).
- MDN [2024g]. *CSS Syntax*. MDN Web Docs, 10 Jun 2024. <https://developer.mozilla.org/en-US/docs/Web/CSS/Syntax> (cited on page 14).
- MDN [2024h]. *Document Object Model (DOM)*. MDN Web Docs, 11 Jun 2024. https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model (cited on page 19).
- MDN [2024i]. *HTML Elements Reference*. MDN Web Docs, 10 Jun 2024. <https://developer.mozilla.org/en-US/docs/Web/HTML/Element> (cited on page 12).
- MDN [2024j]. *Intersection Observer API*. MDN Web Docs, 13 Nov 2024. https://developer.mozilla.org/en-US/docs/Web/API/Intersection_Observer_API (cited on page 60).
- MDN [2024k]. *Introducing the CSS Cascade*. MDN Web Docs, 10 Jun 2024. https://developer.mozilla.org/en-US/docs/Web/CSS/Cascade#origin_types (cited on page 14).
- MDN [2024l]. *JavaScript Modules*. MDN Web Docs, 30 Jun 2024. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules> (cited on page 25).
- MDN [2024m]. *Progressive Enhancement*. MDN Web Docs, 11 Jun 2024. https://developer.mozilla.org/en-US/docs/Glossary/Progressive_Enhancement (cited on page 20).
- MDN [2024n]. *Semantic HTML*. MDN Web Docs, 10 Jun 2024. <https://developer.mozilla.org/en-US/curriculum/core/semantic-html/> (cited on page 12).
- Merino, Leonel, Mohammad Ghafari, Craig Anslow, and Oscar Nierstrasz [2018]. *A Systematic Literature Review of Software Visualization Evaluation*. *Journal of Systems and Software* 144 (Oct 2018), pages 165–180. ISSN 0164-1212. doi:10.1016/j.jss.2018.06.027 (cited on page 5).
- Merino, Leonel, Mohammad Ghafari, and Oscar Nierstrasz [2016]. *Towards Actionable Visualisation in Software Development*. Proc. 4th IEEE Working Conference on Software Visualization (VISSOFT 2016) (Raleigh, North Carolina, USA). 03 Oct 2016, pages 61–70. doi:10.1109/VISSOFT.2016.10 (cited on page 7).
- Meta [2024a]. *Meta*. Meta Platforms, 11 Jun 2024. <https://about.meta.com/> (cited on page 22).
- Meta [2024b]. *React*. Meta Platforms, 19 Nov 2024. <https://npmjs.com/package/react> (cited on page 96).
- Meta [2024c]. *React Design Principles*. Meta Platforms, 27 Jun 2024. <https://legacy.reactjs.org/docs/design-principles.html> (cited on page 22).

- Meta [2024d]. *Reconciliation*. Meta Platforms, 13 Jun 2024. <https://legacy.reactjs.org/docs/reconciliation.html> (cited on page 22).
- Meta [2024e]. *Virtual DOM and Internals*. Meta Platforms, 13 Jun 2024. <https://legacy.reactjs.org/docs/faq-internals.html> (cited on page 22).
- Meta [2024f]. *Why did we build React?* Meta Platforms, 13 Jun 2024. <https://legacy.reactjs.org/blog/2013/06/05/why-react.html> (cited on page 22).
- Microsoft [2024a]. *Monaco Editor*. 29 Oct 2024. <https://microsoft.github.io/monaco-editor/> (cited on page 76).
- Microsoft [2024b]. *TypeScript*. 11 Jun 2024. <https://typescriptlang.org/> (cited on page 20).
- MobX [2024]. *MobX: Simple, Scalable State Management*. MobX Contributors, 28 Jun 2024. <https://mobx.js.org/> (cited on pages 24, 34, 36, 96).
- Müller, Jonas, Lea Rieger, Ilhan Aslan, Christoph Anneser, Malte Sandstede, Felix Schwarzmeier, Björn Petrak, and Elisabeth André [2019]. *Mouse, Touch, or Fich: Comparing Traditional Input Modalities to a Novel Pre-Touch Technique*. Proc. 18th International Conference on Mobile and Ubiquitous Multimedia (MUM 2019) (Pisa, Italy). ACM, 27 Nov 2019, pages 1–7. doi:10.1145/3365610.3365622 (cited on page 26).
- Munsell, Albert Henry [1919]. *A Color Notation*. Munsell Color Company, 1919. <https://gutenberg.org/files/26054/26054-h/26054-h.htm> (cited on page 63).
- npm [2024]. *npm - A JavaScript Package Manager*. 11 Jun 2024. <https://npmjs.com/package/npm> (cited on pages 21, 25).
- OpenJS [2024a]. *Express - Node.js Web Application Framework*. OpenJS Foundation, 19 Nov 2024. <https://expressjs.com/> (cited on page 95).
- OpenJS [2024b]. *Node*. OpenJS Foundation, 19 Nov 2024. <https://nodejs.org/> (cited on page 95).
- Pacione, Michael J., Marc Roper, and Murray Wood [2004]. *A Novel Software Visualisation Model to Support Software Comprehension*. Proc. 11th IEEE Working Conference on Reverse Engineering (WCRE 2004) (Delft, The Netherlands). 08 Nov 2004, pages 70–79. doi:10.1109/WCRE.2004.7 (cited on page 9).
- Parcel [2024]. *Parcel*. 30 Jun 2024. <https://parceljs.org/> (cited on page 24).
- Peeters, Guido [1991]. *Evaluative Inference in Social Cognition: The Roles of Direct Versus Indirect Evaluation and Positive-Negative Asymmetry*. European Journal of Social Psychology 21.2 (Mar 1991), pages 131–146. doi:10.1002/ejsp.2420210204 (cited on page 43).
- Peeters, Guido and Janusz Czapinski [1990]. *Positive-Negative Asymmetry in Evaluations: The Distinction Between Affective and Informational Negativity Effects*. European Review of Social Psychology 1.1 (Jan 1990), pages 33–60. ISSN 1046-3283. doi:10.1080/14792779108401856 (cited on page 43).
- Pereyra, Irene [2023]. *Universal Principles of UX*. Rockport, 07 Mar 2023. 224 pages. ISBN 0760378045 (cited on page 43).
- Petre, Marian and Ed de Quincey [2006]. *A Gentle Overview of Software Visualisation*. PPIG Newsletter. 01 Sep 2006. 10 pages. <https://web.archive.org/web/20061011012412/http://www.ppig.org/newsletters/2006-09/1-overview-svwiz.pdf> (cited on page 5).
- Pickering, Heydon [2016]. *Inclusive Design Patterns: Coding Accessibility Into Web Design*. Smashing Magazine, 01 Jan 2016. 312 pages. ISBN 3945749433 (cited on page 14).

- Pinheiro de Souza, Thomas, Stefan Schintler, and Andreas Steinkellner [2023]. *Gizual - Repository Visualization for Git*. 706.041 Information Architecture and Web Usability WS 2022 Project Report. Graz University of Technology, 08 Feb 2023. <https://gizual.com/resources/gizual-paper-ws2022.pdf> (cited on pages 51–52).
- pnpm Contributors [2024]. *pnpm*. 11 Jun 2024. <https://pnpm.io/> (cited on page 21).
- Potter, John [2024]. *npm Trends - Parcel vs Rollup vs Vite vs Webpack*. npm trends, 29 Jun 2024. <https://npm trends.com/parcel-vs-rollup-vs-vite-vs-webpack> (cited on pages xiii, 24–25).
- Price, Blaine A., Ian S. Small, and Ronald M. Baecker [1992]. *A Taxonomy of Software Visualization*. Proc. 25th IEEE Hawaii International Conference on System Sciences (HICSS 1992) (Kauai, Hawaii, USA). Volume 2. 06 Aug 1992, pages 597–606. doi:10.1109/HICSS.1992.183311 (cited on page 5).
- Procaccino, J. Drew, June M. Verner, Scott P. Overmyer, and Marvin E. Darter [2022]. *Case Study: Factors for Early Prediction of Software Development Success*. Information and Software Technology 44.1 (15 Jan 2022), pages 53–62. ISSN 0950-5849. doi:10.1016/S0950-5849(01)00217-8 (cited on page 3).
- Punchoojit, Lumpapun and Nuttanont Hongwarittorn [2017]. *Usability Studies on Mobile User Interface Design Patterns: A Systematic Literature Review*. Advances in Human-Computer Interaction 2017 (Nov 2017), pages 1–22. ISSN 1687-5893. doi:10.1155/2017/6787504 (cited on page 26).
- Rainer, Austen [2010]. *Representing the Behaviour of Software Projects using Multi-Dimensional Timelines*. Information and Software Technology 52.11 (Nov 2010), pages 1217–1228. ISSN 0950-5849. doi:10.1016/j.infsof.2010.06.004 (cited on page 5).
- Raval, Nihar [2024]. *React vs Angular: Which JS Framework to Choose for Front-end Development?* Radix, 01 Apr 2024. <https://radixweb.com/blog/react-vs-angular> (cited on page 23).
- React [2024a]. *React*. Meta Platforms, 11 Jun 2024. <https://reactjs.org> (cited on pages 21–22).
- React [2024b]. *React 19 RC*. Meta Platforms, 25 Apr 2024. <https://react.dev/blog/2024/04/25/react-19> (cited on page 24).
- React [2024c]. *React Compiler*. Meta Platforms, 01 Jul 2024. <https://react.dev/learn/react-compiler> (cited on page 24).
- Reddy, Nishaanth H., Junghun Kim, Vijay Krishna Palepu, and James A. Jones [2015]. *Spider Sense: Software-Engineering, Networked, System Evaluation*. Proc. 3rd IEEE Working Conference on Software Visualization (VISSOFT 2015) (Bremen, Germany). 27 Sep 2015, pages 205–209. doi:10.1109/VISSOFT.2015.7332438. https://vijaykrishna.github.io/publications/vissoft15_reddy_etal.pdf (cited on pages xiii, 5, 7–8, 84).
- Reel, John S. [1999]. *Critical Success Factors in Software Projects*. IEEE Software 16.3 (May 1999), pages 18–23. ISSN 0740-7459. doi:10.1109/52.765782 (cited on page 3).
- Reid, Brittany, Christoph Treude, and Markus Wagner [2023]. *Using the TypeScript Compiler to Fix Erroneous Node.js Snippets*. Proc. 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM 2023) (Bogotá, Colombia). IEEE, 02 Oct 2023, pages 220–230. doi:10.1109/SCAM59687.2023.00031 (cited on page 20).
- ResponsivelyApp [2024]. *GitHub - ResponsivelyApp*. ResponsivelyApp Contributors, 30 Jun 2024. <https://github.com/responsively-org/responsively-app> (cited on page 27).
- RF [2024]. *Rust Programming Language*. Rust Foundation. <https://rust-lang.org/> (cited on pages 95–96).
- Rhyne, Theresa-Marie [2021]. *Color in a Perceptual Uniform Way*. Medium, 20 Mar 2021. <https://nig htingleadvs.com/color-in-a-perceptual-uniform-way/> (cited on page 63).

- Rollup [2024]. *Rollup*. 30 Jun 2024. <https://rollupjs.org/> (cited on page 24).
- Ryan, Marie-Laure [2003]. *Narrative as Virtual Reality: Immersion and Interactivity in Literature and Electronic Media*. Johns Hopkins University Press, 03 Oct 2003. ISBN 0801877539 (cited on page 53).
- Schintler, Stefan [2024]. *Gizual Data Layer: Enabling Browser-Based Exploration of Git Repositories*. Master's Thesis. Graz University of Technology, Austria, 09 Dec 2024. <https://ftp.isds.tugraz.at/pub/theses/sschintler-2024-msc.pdf> (cited on page 1).
- Schintler, Stefan and Andreas Steinkellner [2024]. *Gizual*. 18 Nov 2024. <https://github.com/gizual/gizual> (cited on pages 1, 85, 87, 95).
- Setlur, Vidya and Bridget Cogley [2022]. *Functional Aesthetics for Data Visualization*. Wiley, 08 Aug 2022. 336 pages. ISBN 1119810086 (cited on page 43).
- Shneiderman, Ben, Catherine Plaisant, Maxine Cohen, Steven Jacobs, Niklas Elmqvist, and Nicholas Diakopoulos [2017]. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. 6th Edition. Pearson, 20 Jun 2017. 624 pages. ISBN 1292153911 (cited on page 43).
- Small, Hugh [2013]. *Florence Nightingale - Avenging Angel*. 2nd Edition. Knowledge Leak, 31 May 2013. 248 pages. ISBN 095727971X (cited on page 3).
- Sourcegraph [2024]. *Sourcegraph*. 30 Jun 2024. <https://sourcegraph.com/> (cited on pages 7, 76).
- SQLite [2024]. *SQLite*. 10 Aug 2024. <https://sqlite.org/> (cited on page 34).
- Stack Overflow [2024a]. *Stack Overflow*. 11 Jun 2024. <https://stackoverflow.com> (cited on page 21).
- Stack Overflow [2024b]. *Stack Overflow Trends*. 11 Jun 2024. <https://insights.stackoverflow.com/trends?tags=reactjs,vue.js,angular,svelte,angularjs,vuejs3> (cited on pages xiii, 21).
- Staiano, Fabio [2023]. *Designing and Prototyping Interfaces with Figma*. 2nd Edition. Packt Publishing, 29 Dec 2023. ISBN 1835464602 (cited on page 44).
- StatCounter [2024]. *Browser Market Share Worldwide*. 10 Jun 2024. <https://gs.statcounter.com/browser-market-share> (cited on page 11).
- Steinkellner, Andreas [2024]. *Gizual User Interface: Browser-Based Visualisation for Git Repositories*. Master's Thesis. Graz University of Technology, Austria, 09 Dec 2024. <https://ftp.isds.tugraz.at/pub/theses/asteinkellner-2024-msc.pdf> (cited on page 1).
- Surma, Das [2019]. *Use Web Workers to Run JavaScript Off the Browser's Main Thread*. web.dev, 05 Dec 2019. <https://web.dev/articles/off-main-thread> (cited on page 31).
- Svelte [2024]. *Svelte*. Svelte Contributors, 11 Jun 2024. <https://svelte.dev/> (cited on pages 21, 23).
- Tauri [2024]. *Tauri*. 09 Dec 2024. <https://tauri.app/> (cited on page 84).
- Todorovic, Dejan [2008]. *Gestalt Principles*. Scholarpedia 3.12 (21 Dec 2008), page 5345. <https://pdfs.semanticscholar.org/49fb/87e3f0a70160c6c6089c7127e85ef7e3ac04.pdf> (cited on page 43).
- Tominski, Christian and Heidrun Schuhmann [2020]. *Interactive Visual Data Analysis*. CRC Press, 30 Apr 2020. 346 pages. ISBN 0367898756 (cited on page 1).
- Turner, Jonathan [2014]. *Announcing TypeScript 1.0*. 01 Apr 2014. <https://devblogs.microsoft.com/typescript/announcing-typescript-1-0/> (cited on page 20).
- Ubl, Malte [2015]. *Introducing Incremental DOM*. Medium, 09 Jul 2015. <https://medium.com/google-developers/introducing-incremental-dom-e98f79ce2c5f> (cited on page 23).

- Unwin, Antony [2020]. *Why Is Data Visualization Important? What Is Important in Data Visualization?* Harvard Data Science Review 2.1 (31 Jan 2020). doi:10.1162/99608f92.8ae4d525. <https://hdsr.mitpress.mit.edu/pub/zok97i7p> (cited on page 4).
- Vaish, Amrisha, Tobias Grossmann, and Amanda Woodward [2008]. *Not all Emotions are Created Equal: The Negativity Bias in Social-Emotional Development*. Psychological Bulletin 134.3 (May 2008), pages 383–403. ISSN 1939-1455. doi:10.1037/0033-2909.134.3.383 (cited on page 43).
- Vite [2024a]. *Vite*. 30 Jun 2024. <https://vitejs.dev/> (cited on page 24).
- Vite [2024b]. *Vite*. 19 Nov 2024. <https://npmjs.com/package/vite> (cited on page 96).
- Vue [2024a]. *Vue*. VueJS Contributors, 11 Jun 2024. <https://vuejs.org/> (cited on pages 21, 23).
- Vue [2024b]. *Vue Rendering Mechanism*. VueJS Contributors, 29 Jun 2024. <https://vuejs.org/guide/extras/rendering-mechanism> (cited on page 23).
- W3C [2011]. *Grid Layout*. World Wide Web Consortium, 07 Apr 2011. <https://w3.org/TR/2011/WD-css3-grid-layout-20110407/> (cited on page 18).
- W3C [2018]. *CSS Flexible Box Layout Module Level 1*. World Wide Web Consortium, 19 Nov 2018. <https://w3.org/TR/css-flexbox-1/> (cited on page 18).
- W3C [2024a]. *ARIA in HTML*. World Wide Web Consortium, 07 May 2024. <https://w3.org/TR/html-aria/> (cited on page 14).
- W3C [2024b]. *Largest Contentful Paint*. World Wide Web Consortium, 15 Jan 2024. <https://w3c.github.io/largest-contentful-paint/> (cited on page 20).
- W3C [2024c]. *W3C*. World Wide Web Consortium, 10 Jun 2024. <https://w3.org/> (cited on pages 11–12, 19).
- W3C [2024d]. *WebAssembly*. 10 Jun 2024. <https://webassembly.org/> (cited on pages 1, 11).
- W3C [2024e]. *What is the Document Object Model?* 11 Jun 2024. <https://w3.org/TR/WD-DOM/introduction.html> (cited on page 19).
- WACG [2024]. *WebAssembly*. WebAssembly Community Group, 30 Aug 2024. <https://webassembly.org/> (cited on page 34).
- Ware, Colin [2021]. *Visual Thinking for Information Design*. 2nd Edition. Morgan Kaufmann, 14 Jul 2021. 224 pages. ISBN 0128235675 (cited on page 1).
- WaSP [2024]. *Web Standard Project - Mission*. Web Standard Project, 10 Jun 2024. <https://webstandards.org/about/mission/> (cited on page 11).
- Webpack [2024]. *Webpack*. 30 Jun 2024. <https://webpack.js.org/> (cited on page 24).
- Wettel, Richard and Michele Lanza [2007]. *Visualizing Software Systems as Cities*. Proc. 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2007) (Banff, Alberta, Canada). 24 Jun 2007, pages 92–99. doi:10.1109/VISSOFT.2007.4290706 (cited on page 5).
- WHATWG [2024a]. *HTML Living Standard*. Web Hypertext Application Technology Working Group, 10 Jun 2024. <https://html.spec.whatwg.org/multipage/> (cited on page 12).
- WHATWG [2024b]. *Web Hypertext Application Technology Working Group*. Web Hypertext Application Technology Working Group, 10 Jun 2024. <https://whatwg.org/> (cited on page 12).
- WHATWG [2024c]. *Web Workers. Chapter 10 of HTML Living Standard*. Web Hypertext Application Technology Working Group, 29 Aug 2024. <https://html.spec.whatwg.org/#toc-workers> (cited on page 31).

- Whitworth, Greg [2018]. *Braces to Pixels*. A List Apart, 15 Nov 2018. <https://alistapart.com/article/braces-to-pixels/> (cited on pages 16–17).
- Wirfs-Brock, Allen and Brendan Eich [2020]. *JavaScript: the first 20 years*. Proceedings of the ACM on Programming Languages 4 (Jun 2020), pages 1–189. ISSN 2475-1421. doi:10.1145/3386327 (cited on page 19).
- Wojciech, Maj [2023]. *Package Manager Wars. The Real Picture*. 22 Oct 2023. <https://dev.to/wojtekmaj/package-manager-wars-the-real-picture-e9p> (cited on page 25).
- yarn [2024a]. *Yarn*. 11 Jun 2024. <https://yarnpkg.com/> (cited on pages 21, 25, 95).
- yarn [2024b]. *Yarn Workspaces*. 29 Jun 2024. <https://classic.yarnpkg.com/lang/en/docs/workspaces/> (cited on page 25).
- yarn [2024c]. *Yarn Workspaces*. 19 Nov 2024. <https://yarnpkg.com/features/workspaces> (cited on page 95).
- You, Evan [2014]. *First Week of Launching Vue.js*. 11 Feb 2014. <https://blog.evanyou.me/2014/02/11/first-week-of-launching-an-oss-project/> (cited on page 23).
- Zeileis, Achim, Kurt Hornik, and Paul Murrell [2009]. *Escaping RGBland: Selecting Colors for Statistical Graphics*. Computational Statistics & Data Analysis 53.9 (Jul 2009), pages 3259–3270. ISSN 0167-9473. doi:10.1016/j.csda.2008.11.033 (cited on page 63).