Visualisation Research Center

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Visual Analysis of Evolution and Uncertainty in Hierarchical Data

Adrian Zeyfang

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Daniel Weiskopf
Supervisor:	Dipl.-Inf. Christoph Schulz
Commenced:	July 10, 2017
Completed:	January 10, 2018

Abstract

Large software projects are often worked on over the course of several years by multiple developers. The history of such projects is typically checked into version control repositories and can be tapped into as a data source for an explorative hierarchy visualization. Our goal was the development of a tool for visualizing the evolution of software repositories. Focus is placed on visualizing a range of time-dependent hierarchies as a whole, rather than looking at individual, isolated snapshots of each timestamp. Hierarchies with attached metrics are derived from the repository at each revision and merged into a supertree, which is mapped using customizable transfer functions and rendered interactively. The tool offers an overview of the entire supertree of a user-defined timespan, while allowing detailed inspection of areas of interest.

Kurzfassung

Große Softwareprojekte werden häufig über mehrere Jahre hinweg von mehreren Entwicklern geschrieben. Der Werdegang solcher Projekte wird meist in Versionsverwaltungssystemen festgehalten und kann als Datensatz für eine explorative Hierarchievisualisierung angezapft werden. Unser Ziel war die Entwicklung eines Werkzeugs zur Visualisierung der Evolution von Software-Repositorys. Der Fokus liegt darauf, eine Reichweite von zeitabhängigen Hierarchien zu visualisieren, statt individuelle Schnappschüsse zu jedem Zeitpunkt einzeln zu betrachten. Hierarchien mit angehängten Metriken werden zu jeder Revision aus dem Repository erzeugt und in einen Superbaum vereinigt, welcher mit anpassbaren Transferfunktionen abgebildet und interaktiv dargestellt wird. Das Werkzeug bietet einen Überblick über den gesamten Superbaum einer benutzerdefinierten Zeitspanne, ermöglicht aber auch die detaillierte Inspektion interessanter Bereiche.

Contents

List of Figures

1 Introduction

Over the course of their development lifecycle, modern large-scale software projects typically undergo significant changes, as teams consisting of multiple developers contribute source code additions and changes. These changes, affecting the structure and contents of the software system, are commonly tracked using version control software such as git or Subversion. Each set of changes made to the software system and checked into the version control repository can be retrieved later, providing a journal of all changes, including a snapshot of the software system's complete state at any given point in time.

In conventional file systems and source control systems, the source code of the software system is organized as a hierarchical dataset (also referred to as a "tree"). Source folders and files make up the inner nodes and leaf nodes of the tree, respectively.

Using version control software, it is possible to extract the hierarchies for each recorded snapshot of the software project from its repository. This "list of trees" can then be used as a dataset for an explorative visualization of the software project's entire development history, granting researchers and developers an overview of the structural and quantitative changes the system's source code tree has undergone over time.

Additionally, the dataset can be enriched with metrics derived from the source code, allowing for further insights and trends to be identified, such as the evolution of function complexity in a specific sub-hierarchy of the dataset. Both the metrics themselves, as well as values representing their uncertainty, such as data point count and standard deviation, can serve as basic data attributes for nodes within the hierarchies.

Beyond repository contents themselves, data can be acquired from auxiliary sources related to the software system. For instance, tracking the issues that have been filed over time and comparing them with other metrics could be useful in identifying possible correlations between certain code characteristics and real-world effects on the software's quality.

Our goal is the development of a set of tools that allows for the acquisition of relevant data from software repositories, the extraction of structural and content differences and metrics from the repository at each point in time, and the interactive visualization of the repository history. The tools should be scalable, so that they can be applied to very large repositories with long revision histories (such as the Linux kernel), but still produce meaningful visualizations for small projects with few revisions.

Finally, rather than visualizing the source code tree of the repository itself, metadata derived from the system at each revision can be used as the underlying hierarchical dataset instead, such as the dependency tree for external libraries required by the system.

1.1 Requirements

The tools developed as part of this thesis must support the full visualization pipeline, from the data acquisition step, over filtering and mapping, to the final rendering step. The user must be able to supply any valid Git repository as a data source, of which all revisions are pre-processed into an intermediate format, which can then be visualized with customizable filtering and mapping parameters.

A set of metrics must be collected for each revision of each file within the repository, by examining the file's metadata and contents. These metrics must be acquired in a pre-processing step, and then stored persistently.

Files and directories checked into the repository must be aggregated over several revisions at the user's choice, resulting in a "supertree" that represents all nodes to have existed during the selected timespan, as well as the evolution of their attributes. To minimize redundancy within this supertree, sufficiently similar subtrees must be merged logically, even if inner nodes leading to the subtree were renamed. Temporal aggregation of node metrics across revisions must follow a customizable reduction function, to allow evaluating various aspects of the change in node attributes, such as upper and lower bounds for metrics, or the standard deviation of a specific attribute over time.

Finally, the visualization must be interactive and responsive, allowing the user to explore any section of the revision history and the merged supertree at any level of detail, ranging from gaining a complete overview of the hierarchy and history, to allowing detailed insights into the features of a specific temporal or structural subregion.

2 Foundation

2.1 Jaccard Similarity Coefficient

The Jaccard coefficient is a scalar measure of similarity between two sets, yielding a number in the interval $[0; 1]$ for any pair of finite sets [Jac12]. A coefficient of 0 indicates disjoint sets with no elements in common, whereas a coefficient of 1 indicates a pair of equal sets.

The Jaccard coefficient of two sets A and B (of which at least one is non-empty) is defined as:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

If both sets are empty, they are considered equal, thus the exception $J(\emptyset, \emptyset) = 1$ is applied to avoid division by zero.

As both the set union and set intersection operations are commutative, the Jaccard coefficient itself is also a commutative operation:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|B \cap A|}{|B \cup A|} = J(B, A)$$

2.1.1 Adaptation to Trees

An extended version of the Jaccard coefficient can be used to compute the similarity of two unordered trees with labeled or otherwise comparable nodes. Collecting the set of all distinct nodes within each tree, then applying the Jaccard coefficient to those sets, yields a measure for the similarity of the trees' contents.

As all structural information about the subtrees within the hierarchy is lost during the set reduction step, only the unique labels of the nodes within each tree are taken into account by this Jaccard-based tree similarity measure.

Due to these properties, hierarchies with certain characteristics are more suitable than others for use with this similarity metric. A dataset in which most nodes have distinct labels will yield a large set, improving the responsiveness of the Jaccard similarity coefficient to node additions and removals, as well as label changes.

In contrast, pairs of trees with a limited, but shared set of node labels and a large number of nodes will typically yield a high similarity coefficient, even if their structure is significantly different, as label collisions are unavoidable as the number of nodes surpasses the number of available labels.

Taken to the extreme, when applied to pairs of trees with nodes that all share one common label (making them essentially unlabeled), the Jaccard coefficient will yield maximum similarity, causing it to be unsuitable for comparing unlabeled hierarchies that only differ in structure.

Additionally, as the order of nodes in ordered trees is not preserved when reducing their labels to a set, the Jaccard similarity coefficient is more appropriate for comparing unordered hierarchies, as differences in node order will not reduce the similarity.

2.2 Tree Edit Distance

Another method of comparing trees is the Tree Edit Distance. It operates on pairs of ordered trees with labeled nodes, and yields the sequence of insertions, deletions and label changes required to transform the first tree into the second tree that has the minimum total cost according to a cost function defined for each of the operations [Bil05].

Cost functions can be defined as constant values, causing the total cost to be the number of operations required to transform the tree. Alternatively, more specialized cost functions can be defined, varying the cost of an operation depending on the length of the node labels, or a measure of string edit distance (such as the Levenshtein distance) between the old and new label for the case of a node rename.

Unlike the Jaccard similarity coefficient, Tree Edit Distance takes structural differences between the two trees into account. Due to being defined as a sum of operation costs, the Tree Edit Distance is not a normalized measure of similarity, and instead scales with the absolute difference between the two trees.

Algorithms for computing the Tree Edit Distance typically make use of a dynamic programming approach, in which problems are recursively decomposed into smaller subproblems, the solutions of which are memoized and reused. This allows for the computation of Tree Edit Distance for ordered trees within polynomial time.

Additionally, while algorithms for computing the Tree Edit Distance between unordered hierarchies exist, the problem is known to be NP-hard for the general case [Bil05].

2.3 Tree Visualizations

A great number of published techniques exists for visualizing hierarchies. The TreeVis project maintains an extensive collection of such methods [Sch11], comprising over 300

techniques at the time of writing. The different techniques are designed for processing hierarchies of a certain size, highlighting specific features of the dataset, or taking advantage of a variety of interaction methods.

Choosing an appropriate tree visualization method for the hierarchy at hand based on its scale, structural features and attributes is crucial to ensure the its readability to the user and fitness for its intended purpose, which varies depending on the application.

Two such techniques are described, applied and implemented in this thesis.

2.3.1 Node-Link Diagrams

In a node-link diagram, nodes within a hierarchy are displayed as distinct visual elements, which are then connected to each other using lines to indicate parent-child relationship between pairs of nodes.

Nodes within a node-link diagrams are typically laid out in a way that conveys information about each node's location within the hierarchy through its position in the diagram. For instance, placing the root at the top and rendering child nodes beneath their parent highlights the parent-child hierarchy and allows judging the depth of a node within the hierarchy by its vertical position in the diagram.

Various algorithms exist for laying out nodes in a node-link diagram for efficient usage of space without causing any nodes and links to overlap.

Due to the explicit links between parents and children, node-link diagrams are an intuitive way to visualize small hierarchies. However, when visualizing larger datasets, leaf nodes are densely packed in the diagram, while inner nodes in higher levels of the hierarchy will be sparsely laid out and separated by large amounts of white space [ZMC05].

2.3.2 Indented Tree Plots

Indented tree plots organize nodes along a uniform, 2-dimensional, rectangular grid. A "primary" axis is chosen, along which nodes are placed at a regular interval, each occupying exactly one row or column of the diagram. The "secondary" axis, which is perpendicular to the primary, indicates the depth of each node within the hierarchy, with the amount of indentation along this axis corresponding to the distance from the node to the root.

Along the primary axis, nodes are arranged according to the "pre-order" tree walk: beginning at the root, each node is first plotted to the diagram itself, followed by each of its children, which are in turn followed by their own children, until a leaf is reached, in which case the recursion terminates and the next unvisited sibling is plotted. This continues until the entire hierarchy has been traversed. The position on the primary axis is incremented by a constant amount for each visited node, resulting in an even distribution. The depth of the current recursion step determines the position of the node along the secondary axis. The root is always located in one of the corners of the diagram.

As the positions of nodes in relation to each other convey sufficient information to deduce their nesting depth within the hierarchy, as well as parent-child and sibling relationships, the explicit rendering of edges connecting parents and children is not necessary and typically omitted to reduce visual clutter. However, edges can be left intact as a redundant visual aid to improve the readability of the diagram.

The nodes within an indented tree plot can be downscaled to pixel or subpixel level, resulting in a variant diagram type known as "Indented Pixel Tree Plot" (IPTP), which can be used for visualizing extremely large hierarchies consisting of hundreds of thousands of nodes [BRW10]. Indented Pixel Tree Plots display an overview of the coarse structure of the whole dataset while offering interactivity features, allowing subsections of the hierarchy to be explored in more detail.

3 Software Design

3.1 Requirements

Data acquisition: The software must provide functionality to analyze a Git repository, extracting the directory and file structure at each commit and generating metrics for each file. The collected information should be stored on the file system.

Revision overview: The user must be presented with the list of available revisions parsed from the repository, and the functionality to select a range of revisions for visualization must be provided.

Tree merging: When selecting a range of revisions, a super-tree must be constructed by merging the contents of the repository at each revision. All metrics must be spatially and temporally aggregated in a meaningful way, depending on the nature of the metric to be aggregated. Customization options must be provided, allowing for the mean, standard deviation, minimum and maximum to be computed for each metric.

Transfer functions: All available metrics on a node within the hierarchy must be mappable to visual properties in the diagram, such as fill color, label color, outline thickness or visual extensions attached to nodes.

Node-link diagram: The merged repository tree must be visualized as a node-link diagram. The orientation of the diagram must be adjustable. The user must be able to scroll and zoom the diagram interactively.

Indented tree plot: The merged repository tree must be visualized as an indented tree plot. The orientation of the diagram must be adjustable. The user must be able to scroll and zoom the diagram interactively. Labels must be rotated when the diagram layout is set to left-to-right or right-to-left.

3.1.1 Quantity Structure

The software must be able to handle large repositories with up to 30000 commits, each of which may contain up to 50000 files.

3.1.2 Realization of Requirements

The aforementioned requirements specify a complete software system for an extensive visual exploration of software repositories. In order to focus and conserve development resources, the software system developed as part of this thesis is initially implemented as a prototype, omitting some requirements of lesser priority.

In particular, only a subset of the metrics described in section 3.5 will be implemented, as the technical details of metrics collection itself are not within the scope of this thesis. Instead, the full range of customization options for the temporal and structural aggregation of metrics described in section 3.6 will be provided.

3.2 Merging Unordered Trees

In order to visualize the combined file system structure from a range of revisions within the repository, a "super-tree" has to be computed. An operation $Merge(T_1, T_2)$ must be defined, which is then applied sequentially to the file system tree of each of the selected revisions to obtain the merged tree:

$$Merge(T_1, T_2, ...T_n) = Merge(T_1, Merge(T_2, ..., Merge(T_{n-1}, T_n)...))$$

In a naive approach to this problem, the two trees T_1 and T_2 are simply merged based on label equality. First, the immediate children of T_1's root node are iterated. For each child C_1 of T_1, T_2's children are checked. If T_2 contains a node C_2 with an equal label, C_1 and C_2 are marked as merged and added to the result tree. The algorithm is then applied recursively to their subtrees. Nodes in T_1 or T_2 for which no equally labeled counterpart was found in the other tree are added directly to the resulting tree.

This approach can be thought of as a union operation between the two trees. While this algorithm is straightforward to implement, it has the effect of duplicating entire subtrees when the label of an inner node is changed, losing correlation and cluttering the tree with large numbers of nodes that did not undergo changes themselves (fig. 3.1).

When working with a repository as the source of data, this situation can occur when a directory is renamed in a commit. When such a commit is included in the range of revisions to be merged, all files and subdirectories appear twice in the resultant supertree, with one subtree containing the accumulated metrics from the time before the offending commit was

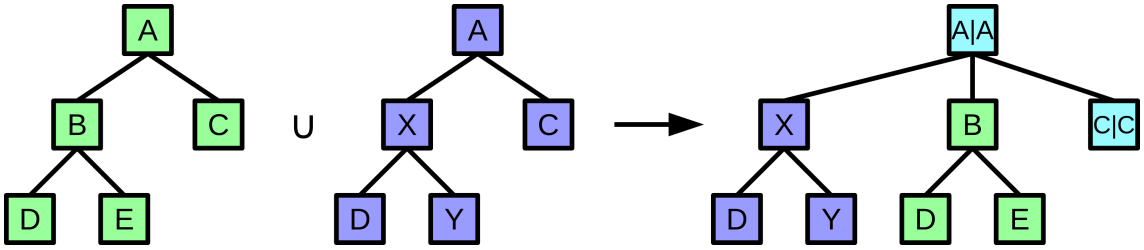

Figure 3.1: Merging two unordered trees using the "Union Merge" algorithm. Similar subtrees with differing inner node labels are duplicated, rather than being logically unified.

made, and the other subtree containing the metrics from the timespan after the commit in question.

The tree merging algorithm can be improved using the adaptation of the Jaccard similarity coefficient for trees, as discussed in section 2.1.1. Rather than only merging nodes with equal labels, the set of all descendant nodes' unique labels is used as a secondary similarity measure between two subtrees if no exact match is found.

For each immediate child node C_1 within T_1, T_2's child nodes are first checked for label equality. If a child C_2 with an equal label is found, C_1 and C_2 are merged, as is the case in the previously described naive approach. However, the remaining sets of unmatched nodes U_T1 and U_T2 are not added to the result tree. Instead, the two sets of nodes will be matched based on their mutual Jaccard similarity coefficient, with a greater coefficient indicating a more suitable match. This problem is essentially a variant of the Stable Marriage Problem, which is known to be solvable in quadratic time with respect to the number of entries in each set [GS62].

Once a stable marriage between the members of the two sets has been computed, the pairs of "married" nodes are merged if the Jaccard similarity of their subtrees falls above a specific threshold ε. For instance, selecting a threshold of $\varepsilon = 0.2$ requires that at least 20% of unique node labels are shared between the two merge candidates. This threshold ensures that unrelated nodes with a low degree of similarity are not falsely merged, as this could skew attributes attached to the nodes and lead to an unexpected hierarchical structure. The label of the resulting node is modified to include both source labels, indicating that the node was renamed.

This improved algorithm reduces redundancy and improves logical coherence between the source trees and the result tree, even if inner node labels are changed between the two trees (fig. 3.2).

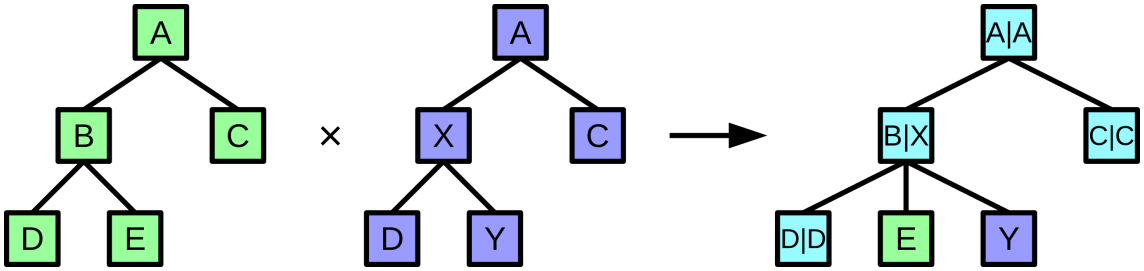

Figure 3.2: Merging two unordered trees using the "Jaccard Similarity Merge" algorithm ($\varepsilon = 0.2$). Renamed subtrees with sufficient similarity are unified based on child node labels.

3.3 Visualization Pipeline

The software uses a pipeline architecture for data acquisition and visualization (fig. 3.3). The initial data source is a version control repository of the user's choice. Currently, repositories managed by the Git versioning system are supported.

Data acquisition: Once a Git repository is selected, the commits leading up to the currently checked out branch are enumerated and iterated. Each commit is sequentially checked out, and the repository's working directory is traversed. All files and folders are tracked as nodes in an in-memory tree structure, with the label of each node being set to the name of the directory or file it represents.

For each file, a series of metrics is collected and attached to the node that represents it. The user can choose to omit collecting certain metrics, as their computation may be time-intensive, resulting in long parsing times for large repositories with long histories. As an optimization, metrics are only collected for files that were changed in the current commit. The available metrics are described in more detail in section 3.5.

Finally, the annotated trees for each commit are stored in intermediate storage files, which can later be read by the software.

Filtering: When a parsed repository is opened, a timeline of all revisions is displayed. The user can select a continuous range of revisions, from which a merged tree will be generated, as described in section 3.2. This effectively filters out the remainder of the data, as only the selected range of revisions is aggregated to form the supertree that serves as the dataset to be visualized.

Within the tree itself, nodes can be collapsed based on customizable conditions, such as node depth or thresholds for specific metrics. The descendants of collapsed nodes are ignored by the mapping and rendering steps by default, but can be revealed by interactively "expanding" the visualized node.

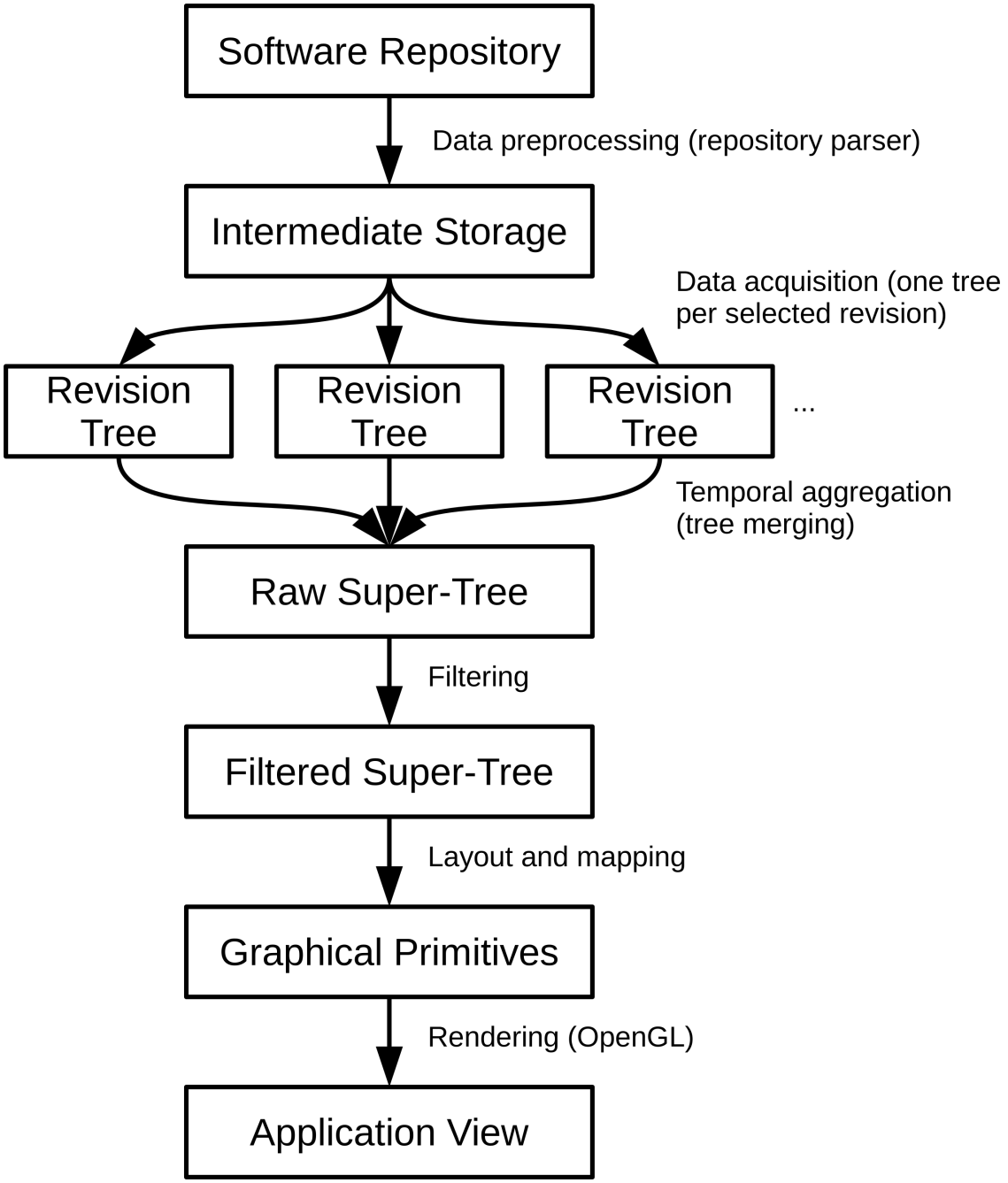

Figure 3.3: Visualization pipeline

In addition to conditionally collapsing inner nodes, filters that apply to all nodes can be specified, based on similar customizable conditions. Unlike collapsed nodes, filtered nodes will be prevented from showing up in the visualization at all, and can not be revealed by user interaction.

Mapping: The transfer function that converts nodes into graphical primitives can be adjusted by the user to visualize the values of specific attributes attached to each node. Fill and outline colors for the graphical entities representing a node can be set depending on the value of any metric according to a customizable color scale. To view correlations between multiple different metrics, additional visual elements can be attached to each node and colored or resized based on node-specific attributes.

Mapped nodes are then laid out according to the selected layout algorithm. The Node-Link layout algorithm organizes nodes by depth and connects parents and children using lines. The Indented Tree layout algorithm spaces nodes evenly along one axis, and indents them along the perpendicular axis to indicate depth; no explicit links are drawn in this type of layout.

Rendering: The mapped primitives are rendered and displayed on the screen. By default, the view is zoomed out enough to display an overview of the entire diagram. The user can use the mouse cursor to interact with the application to pan and zoom the view.

3.4 Visualization and Interaction Design

The application's graphical user interface is subdivided into two primary sections, which are arranged vertically within the main window (see fig. 3.4).

Near the top of the window, a horizontal panel displays a list of all revisions, represented as vertical lines, which are chronologically sorted from left to right and spaced according to the commit timestamp. The user can click and drag to select a continuous range of revisions. The selected revisions will then be merged asynchronously and subsequently visualized.

The remainder of the window is dedicated to the main view, which visualizes the merged tree dataset based on the series of revisions selected in the top panel. The visualization is designed to aid in the exploration of the dataset, enabling the user to gain an overview of the repository's structure and evolution over the selected timespan. The displayed hierarchy can be enriched with collected metrics, which can be mapped using customizable transfer functions.

By clicking and dragging with the mouse cursor, the tree view can be panned freely. Scrolling with the mouse wheel zooms the view in or out on the current mouse cursor position, depending on the scroll direction.

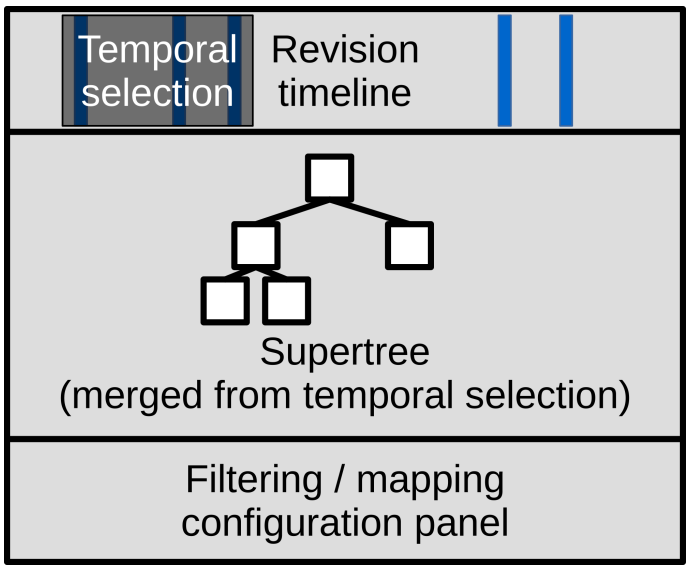

Figure 3.4: Mockup of the primary application window

3.4.1 Filtering

At multiple stages within the visualization pipeline, the current dataset can be filtered. The filters and their parameters can be interactively chosen and customized by the user of the application.

When a parsed repository dataset is initially loaded, the list of all revisions is displayed in the revision selection panel near the top of the application window. No revisions are selected by default. By clicking and dragging across a range of revisions, the input dataset for the tree merging algorithm is reduced to the revisions contained within the user's selection.

Once a range of revisions is selected, a zoomed-in view of the range is displayed as a separate panel below the revision overview. The view is limited to revisions selected in the above view, and can be used to refine the selection with more precision through the same interaction method of clicking and dragging. This process can be repeated recursively, creating an increasing number of revision selection bars, until the desired level of detail has been reached and the correct range of revisions has been selected.

The merged supertree can then be further filtered at node level. Using a menu function, the user can open a filter panel that displays a list of active filters. No filters are specified by default. When creating or editing a filter, the user can choose a node attribute to check, a threshold and comparison operation, and a resultant action (collapsing or hiding the node). Once the filter has been created, it will be applied to all displayed nodes, which will be hidden or collapsed when the criteria specified by the user are met.

3.4.2 Mapping

Two visualization methods are available for displaying the merged hierarchy. Both of them share many customizability options regarding transfer functions from node attributes to graphical properties, but use a different approach for laying out the visual representations of nodes and their relations.

The Node-Link Diagram mapper generates a layout that first arranges nodes by depth, while placing sibling nodes aligned perpendicular to the direction of depth. For instance, in a "top-down" node-link diagram, the root node is placed at the top, with descendants being placed beneath their respective parents, leading to an increasing node depth within the hierarchy corresponding to a lower vertical position on the diagram. Sibling nodes are arranged left-to-right in this diagram orientation. Once placed, pairs of parent and child nodes are then connected using straight lines.

The Indented Tree Plot mapper organizes nodes along a regular grid, where one axis corresponds to the node index when traversing the tree in pre-order, starting at the root, while the perpendicular axis corresponds to the depth of each node, "indenting" it by an amount linear to the ancestral distance to the root node. In a "top-down" indented tree plot, the top left corner of the diagram is occupied by the root node, while its children are placed beneath it with a fixed offset to the right from their parent. Each of these children then forms the new root of its own subtree, displaying all its own children beneath it before moving on to the next unvisited sibling. No lines are used to connect nodes in this type of diagram.

In both diagram types, nodes are displayed as rectangles. Using a menu function, the user can open a mapping settings panel, in which each of the available graphical properties per node can be customized. The color of node fills, outlines, labels and links can be set according to a user-defined transfer function, which samples from a color map depending on a specific attribute, with an optional mathematical scaling modifier (linear, logarithmic, square root). Additionally, the thickness of the outline, as well as the size and color of "decorations" can be defined to scale with certain node attributes, which allows for visualizing and correlating additional node properties.

3.4.3 Brushing and Linking

The merged tree view is linked with the revision panel. As the merged tree is derived from the subset of data chosen in the revision panel, selecting a range of revisions directly affects the merged tree displayed in the main view.

Updating the main view may take an indeterminate amount of time, as potentially large hierarchical datasets are merged over all selected commits. Therefore, the update is performed asynchronously and can be interrupted by simply reselecting a timespan in the revision panel, allowing the user to correct their choice if a series of commits was selected by mistake.

3.5 Metrics

3.5.1 File System

Some metrics can be collected for any file or directory in the repository, without the need to perform any static analysis on its contents. As a result, gathering file system-level metrics takes less time, as only file attributes have to be queried.

Structural metrics: Certain properties depend solely on the structure of the hierarchy itself. The immediate child count, total descendant count, total leaf count and maximum subtree depth can be selected as metrics, which are then computed for each node on the parsed repository tree.

These metrics can give insights into the quantity structure of parts of the hierarchy.

File content metrics: The byte count and line count of files can be determined for all text files, yielding insights about the approximate amount of information contained within each file, and allowing particularly large directories or files to be identified. For binary files, only the byte count is meaningful.

3.5.2 Code metrics for C and C++ source files

In order to gain insights into the code contained within source files in the repository, static analysis tools can provide language-specific metrics. The software supports a set of metrics for valid C and C++ source files, which will be parsed with the aid of an external tool.

Lines of code: The lines within the source file are analyzed, categorized into different types and counted. Lines containing statements, empty lines and lines with comments are counted separately. In particular, measuring the comment density of a source file can give an impression of its maintainability, especially by developers unfamiliar with the code.

Unit counts: By counting the statements and function definitions in each source file, the amount of code contained within specific files can be estimated and compared across files.

Cyclomatic complexity: The cyclomatic complexity, defined as the number of linearly independent paths through the control flow graph of a function, can be used as a measure for the maintainability of a function. By parsing and traversing the abstract syntax tree and counting the number of branches and loops, the cyclomatic complexity of any function can be computed statically.

As the cyclomatic complexity metric works on individual functions, but metric results are attached to nodes and thus cover the entire source file, the complexity is computed for all functions defined in the source file, then aggregated using a customizable function, allowing the user to choose between averaging the cyclomatic complexity, or keeping the maximum value, returning the most complex function in the source file.

Nesting depth: A group of related metrics that can be indicative of potential issues in source code is the maximum depth by which loops, conditional branches or expressions are nested. Deeper nesting results in harder to understand code, as context of surrounding conditions or loops is lost when reading deeply nested statement blocks [HM81]. As another function-level metric, the nesting depth can also be aggregated over the entire file by a user-defined reduction function, with the same options that are available for the cyclomatic complexity metric.

3.6 Attribute Aggregation

During the tree merging step, the attributes of all input nodes are collected within the nodes of the result tree. Only values from changed nodes are added to the metrics aggregation list. Once all revisions have been merged, each node within the supertree contains a list of metrics, each of which contains a list of all values the metric has assumed over each revision that changed the file the node represents. The metric values are then reduced to a single value, based on a function that can be customized per-attribute.

Several such reduction functions are provided.

Maximum and minimum: The greatest/lowest metric values are stored. This reduction function discards all other data points, but can still be useful to quantify extreme values.

Mean: All metric values are summed up and divided by the total number of values. This yields the average value for the metric, including any outliers that may skew the result.

Median: The metric value that has an equal number of values greater and less than itself is chosen as the aggregated value. Even-sized lists are "rounded" upwards or downwards at the user's choice. This reduction function is slightly less performant in comparison to the mean, but yields results that are less affected by outliers.

Difference: The earliest and latest metric values are subtracted from each other, yielding the total difference within the metric from the beginning to the end of the merged timespan.

Standard deviation: The following expression is used for aggregating the metric values:

$$\sqrt{\frac{1}{N}\sum_{i=0}^{N}\left(x_i - \frac{1}{N}\sum_{i=0}^{N}x_i\right)^2}$$

N is the number of metric samples and $x_0 ... x_N$ are the individual values assumed by the metric.

This function yields the standard deviation of the metric value, quantifying how much the individual samples differ from the mean. When used in the visualization, this function enables identifying sections of the repository with significant changes in certain metrics.

Data point count: This aggregation function simply counts the number of distinct data points given for a specific metric on each node. This allows inferring the uncertainty of other aggregation types of the same metric.

4 Implementation

The application is implemented based on the design specified in chapter 3, yielding a functional software system that meets the specified requirements to a sufficient extent. The subset of requirements realized is described in section 3.1.2.

4.1 Technology and Libraries

The software is written in C++ using the C++14 standard. The open source library Qt 5 is used for the graphical user interface and platform-specific functionality not provided by the C++ standard library. Rendering is performed using OpenGL wrappers provided by Qt.

Development is performed in Qt Creator and executables are built using GCC 5.4. A Subversion repository is used for source code version control. The project is developed, built and executed primarily on Linux Mint 18 (64-bit).

4.2 Data Acquisition

Before a software repository can be visualized, it must be transformed into a format recognized by the application. This pre-processing step also performs metrics collection. A graphical user interface to the repository parser, accessible via an option in the "File" menu of the main application window, is provided to allow the user to select input and output directories, the type of repository and the metrics to be collected (fig. 4.1).

In order to collect information from a repository chosen by the user to be parsed, the Git executable installed on the system is invoked several times, serving as an interface to Git's internal storage format. If no such executable is found, repository parsing fails and the user is prompted to supply a valid Git executable.

First, several preconditions are checked. If the selected directory is not a valid Git repository, parsing is canceled, as no revision history can be extracted without a version control database.

If the Git repository is not pristine (if unstaged or uncommitted changes exist in the working tree), parsing is also canceled. This is because checking out past commits requires a clean working directory, and to avoid the accidental loss of any uncommitted changes.

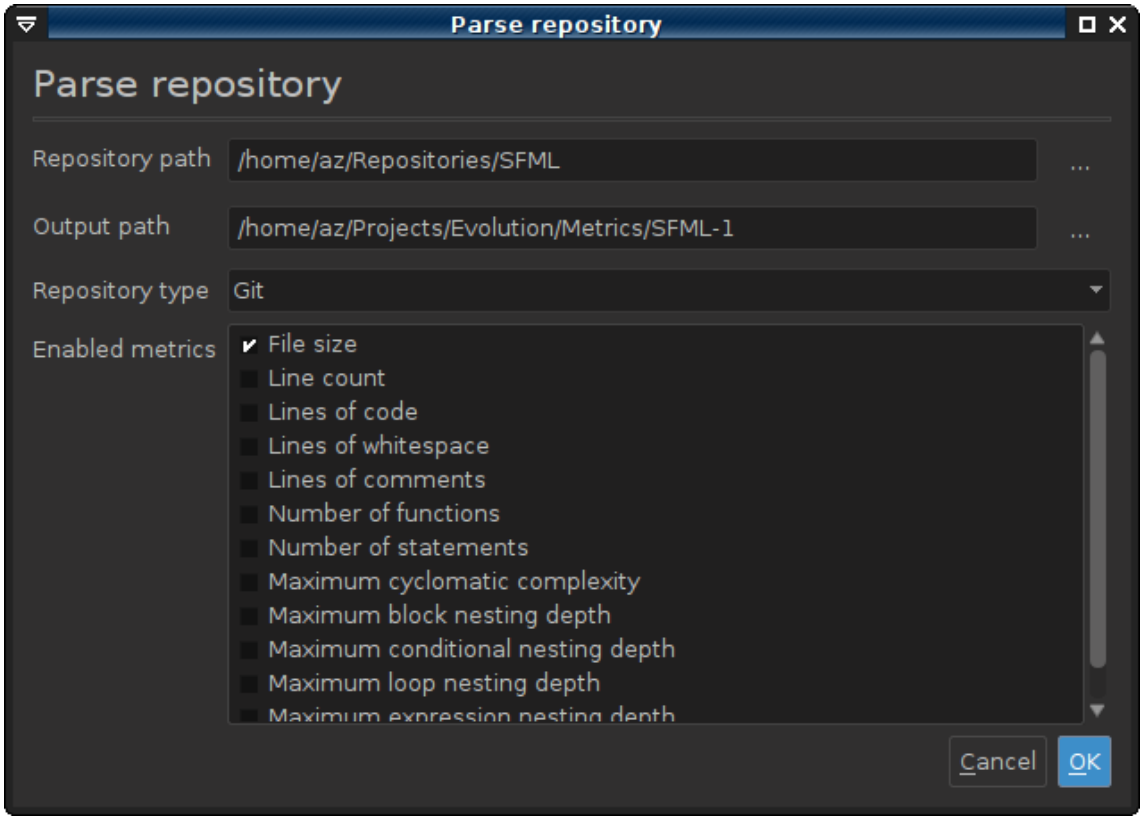

Figure 4.1: Graphical user interface for the repository parser.

Finally, if the repository is in a "detached HEAD" state, meaning that the currently checked out commit is not associated with a branch, parsing is canceled.

If all precondition checks are successful, the commit history of the current branch is enumerated. Each of the commits is then sequentially checked out, replacing the contents of the working directory with those of the past commit. The metrics selected by the user for acquisition are then computed on the working tree, by iterating through all files changed in the current commit, adding them to an in-memory tree structure and evaluating the file contents, generating metrics which are then attached to the node representing the file.

In the prototype developed alongside this thesis, only the byte count metric is implemented, which is measured by querying each file in the repository for its file size. For directories, the byte count of all child files and directories is recursively summed up.

For each commit, the hierarchy is serialized and saved to a file within user-specified output directory. The file name contains the commit date as a UNIX timestamp.

4.2.1 Data Storage Format

The complete snapshot hierarchy is stored in a binary file format. One such file is saved per revision. Within each file, information about each node in the hierarchy is stored in

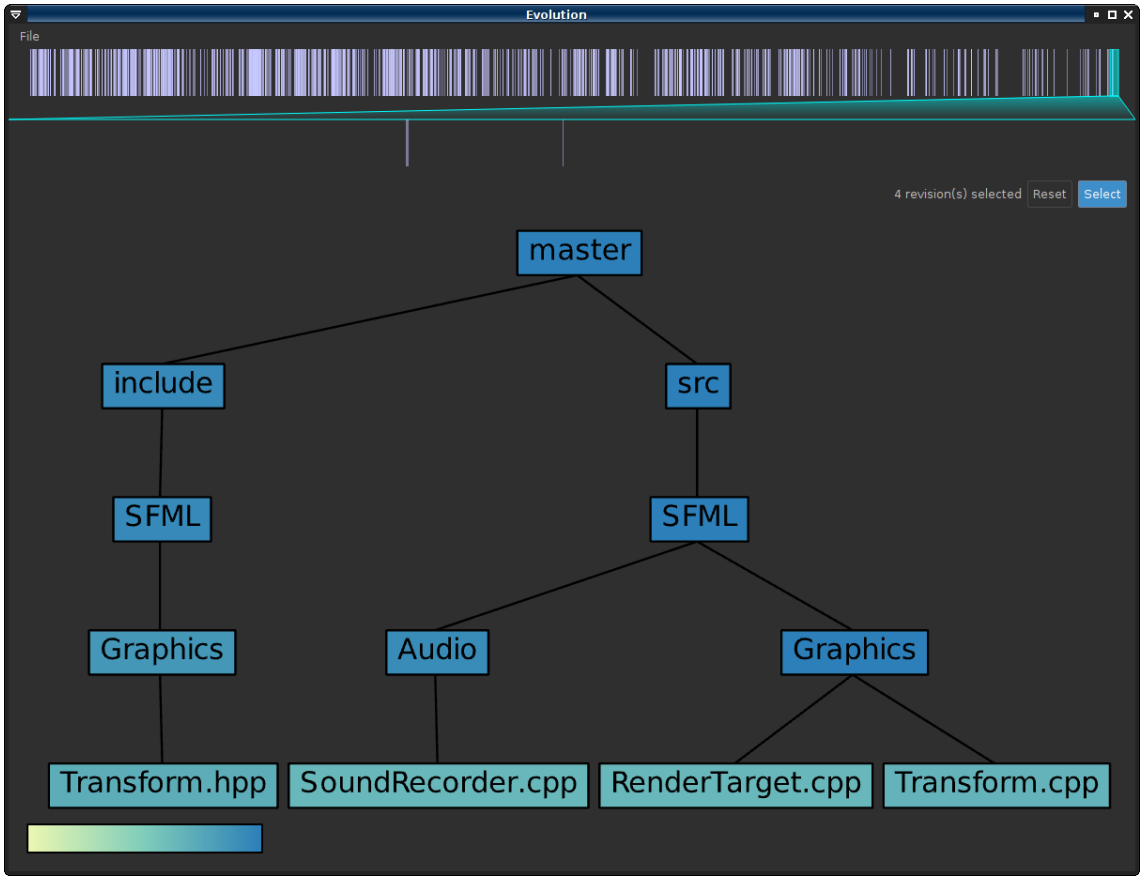

Figure 4.2: Primary application window, coarse revision range selected. The SFML repository is currently opened as a dataset (see chapter 5).

a packed format. Nodes are arranged in pre-order, placing the root at the beginning of the file. Structure is encoded by first storing the node depth within the hierarchy as a 32-bit integer, enabling deserialization code to infer parent-child relationships. This is followed by a tuple of node attributes: the label as a length-prefixed UTF-8 string, a bitfield representing various flags (such as whether the node is a file or directory, or whether a node was changed in the current commit), and the list of raw metrics collected for this node. To encode a variable number of metrics, the metrics count is first serialized as an integer, followed by a packed list of doubles representing the values of each metric.

A binary format was chosen over a text format in order to maximize serialization and deserialization performance. While text-based formats offer better interoperability across programming languages and hardware architectures, the overhead of converting numbers to and from a decimal format can cause a significant loss in performance.

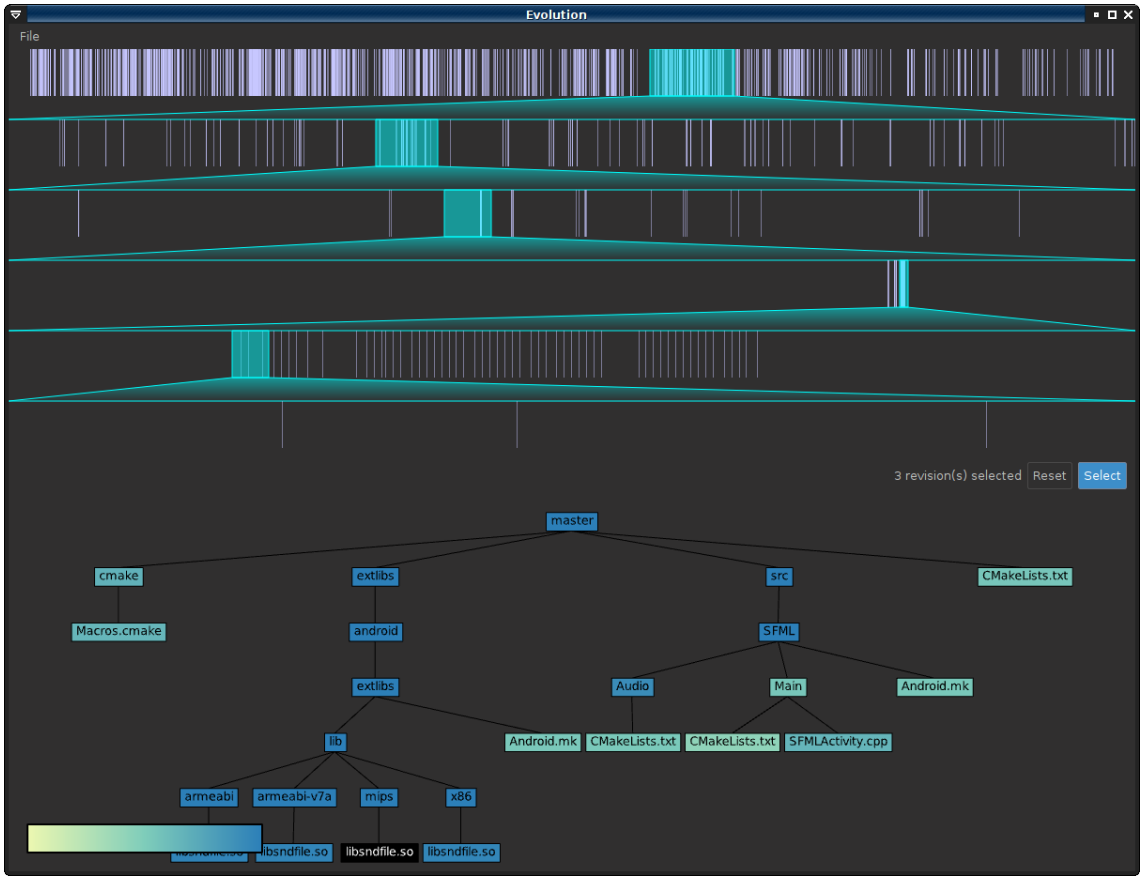

Figure 4.3: Primary application window, finer revision range selected in multiple steps

4.3 Main window

As specified in section 3.4, the application's main window is subdivided into a revision range selector, and a view of the aggregated hierarchy (fig. 4.2).

Within the revision selector, each commit is represented by a vertical line at a horizontal position corresponding to its timestamp relative to other visible commits. Initially, the entire commit history is visible. Each commit is a single pixel thick. If multiple commits fall on the same pixel, they are rendered with a slightly greater brightness to highlight stacked commits within close vicinity of each other.

Clicking and dragging across the selector creates a selection rectangle and adds another revision selector beneath the one the selection was made in, with the view zoomed in exactly on the selection above it. This link is indicated by a funnel shape that is drawn between adjacent selectors (see fig. 4.3).

When a selection is made, a label beneath the lowest selector displays the number of revisions contained within the selected timespan. Next to the label, "Reset" and "Select" buttons are provided. Clicking the "Reset" button will clear the current selection and

remove all additional revision selectors, leaving only the complete overview. The "Select" button will merge all commits within the selection, map the resulting supertree and render an overview of the complete hierarchy to the main view.

The decision to require the user to click the "Select" button before performing the merge was made because of the unpredictable performance of merges, particularly when working with large-scale repositories with a great number of revisions and files. This design choice allows the user to recursively refine their temporal selection without having to wait for the entire hierarchy to render at each intermediate selection step.

4.3.1 Hierarchy Visualization

After the merged supertree is passed through a series of optional filters, which traverse the hierarchy and remove nodes that match certain conditions (such as files that were not changed in the current commit), a customizable transfer function maps each node into a data structure representing a graphical description of the node and its link. The structure holds colors for node fill, outline, link, label and decoration, as well as the link's and outline's thickness and label text.

The transfer function can set any of these graphical attributes depending on each node's aggregated metrics. Colors are mapped by sampling from a color gradient via interpolation, using a linear or logarithmic scale with customizable upper and lower limits, as well as an arbitrary number of differently colored stops in the gradient. Stroke thickness values can be mapped to be directly based on metrics or other node attributes. Finally, the node label can either by transferred directly, or enhanced with node metrics to display a more accurate readout, redundantly supplementing other graphical properties and granting a frame of reference for color mappings.

In the application prototype, the transfer function is specified at compile time.

Once the transfer function is applied, each node is passed to the current tree renderer. The abstract tree renderer superclass contains common rendering features needed for all implemented node visualizations: it handles conversion of the intermediate node descriptions to graphical primitives and rendering of said primitives using OpenGL. Primitives are categorized into chunks based on their bounding box, and only chunks visible within the current viewport are rendered. This optimization improves rendering performance and enables interactivity, even with very large diagrams.

The computation of positions, sizes and linking points of nodes is delegated to tree renderer subclasses.

Indented Tree Renderer

One of the specialized tree renderers lays out nodes to form an indented tree plot. Starting from the root, the hierarchy of graphical node descriptions is traversed in pre-order. A

depth parameter, increased by one for each recursive function call, keeps track of the current nesting level, while an index counter increases by one with each node passed. The values of these two counters translate to the position of graphical primitives representing the node.

For a top-down diagram, the coordinate on the X-axis of each node depends on the recursion depth at the time of traversal, while the coordinate on the Y-axis depends on the value of the index counter. A "dimensions" data structure supplied to the renderer determines node spacing, indentation size, label padding and diagram orientation. Different orientations are produced by rendering the diagram top-down, then applying a transformation to the result.

No links are rendered in an indented tree plot. Therefore, link mapping attributes are discarded by this renderer.

Node-Link Renderer

The node-link renderer generates its hierarchy layout in two passes. The first pass computes the required space for each node, which is the width in a top-down or bottom-up diagram, or the height in a left-to-right or right-to-left diagram. Required space is computed recursively, taking the maximum of the node's own size (based on the label's bounding box) or the cumulative required space of its children (plus spacing). As each node's required space depends on the space taken by its child nodes, the renderer's first pass operates depth-first on the hierarchy.

Once all space requirements have been computed, a second pass lays out all nodes, starting at the root. Each node is centered within its own required space, with a fixed distance from their parent and their siblings (if present). The axis along which nodes are centered depends on the diagram's orientation as specified in a "dimensions" data structure, similar to the indented tree renderer. The graphical primitives representing the node are placed at the computed location. Additionally, a line connecting the node to its parent is emitted for each node, with the exception of the root node. The line touches the node's bounding box at opposing sides, with the edge being dependent on the diagram orientation. Because each node's location depends on the position of its parent, this pass traverses the hierarchy in pre-order.

5 Results

5.1 Examples

To demonstrate and evaluate the functionality and usefulness of the application, an example datasets was derived from the master branch of the SFML source code repository on GitHub (https://github.com/SFML/SFML). SFML is an open-source multimedia library with multiple active developers and a history of around 2000 commits at the time of writing. The repository contains about 3000 unique file paths that have been checked over the course of the library's development.

Fig. 5.1 depicts a the result of visualizing a pair of adjacent revisions using the node-link renderer. A left-to-right diagram displays the paths to files that have been changed in the selected commit. The fill color of each node represents how many bytes have been changed for each node. As the color scale in the bottom left corner indicates, a yellow color indicates that relatively few bytes changed, while a darker blue indicates a greater quantity of changes. Inner nodes are colorized by the sum of their own children. A filter has been applied to remove all nodes that have not received any changes, drastically minimizing the size of the diagram compared to rendering the full repository.

Each node's label corresponds to the name of the file or directory it represents. The root node receives the name of the checked out repository branch (which defaults to "master" for Git repositories) at the time of data acquisition.

In fig. 5.2, a complete, zoomed-out overview of all nodes that have received changes across a larger range of revisions is shown. The standard deviation of the size (in bytes) of each node is used for colorization. Files that received large quantitative changes throughout the selected range of commits are highlighted as blue rectangles, while ones with smaller or fewer changes receive a yellow color. These colors are interpolated depending on the actual standard deviation for each file.

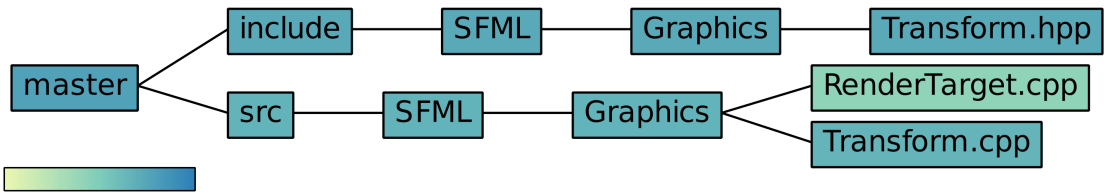

Figure 5.1: Node-link diagram showing the files changed in a single commit.

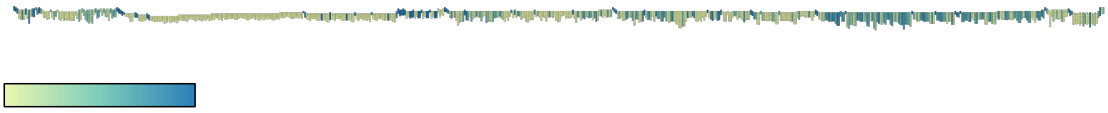

Figure 5.2: Indented tree plot displaying an overview of the changes made to the repository over a series of revisions.

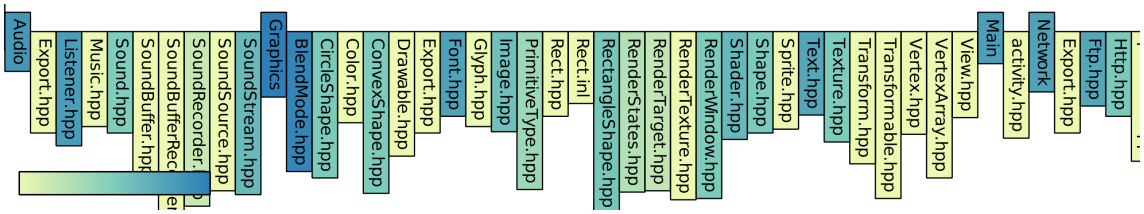

Figure 5.3: Detailed view of fig. 5.2.

While the compact structure of an indented tree plot makes it useful for visualizing large quantities of nodes, individual labels cannot be read at a far zoom level. Nevertheless, interesting structures or trends may be identified through color. Upon such identification, specific sections of the diagram can be explored in more detail by zooming interactively, enabling the user to view those sections in more detail and determine which files have undergone a certain amount of changes (fig. 5.3).

As larger volumes of data are analyzed and the resulting supertree grows in size, node-link diagrams begin to become less practical. Large amounts of whitespace and long link lines cause the visual coherence between parent and child nodes to be lost, making interactive exploration difficult when linked nodes no longer fit within the screen at a zoom level sufficiently close to read node labels. This issue is demonstrated in fig. 5.4, where the distance between child nodes and parent nodes grows significantly, particularly in close proximity to the root. This issue increases in severity as the tree continues to grow in size, as shown in fig. 5.5: parent-child links are stretched to great lengths, and almost appear to be intersecting at far zoom levels.

Indented tree plots allow for a more consistent visual traversal of the ancestor chain: rather than having to follow a link line until the parent node is identified, the parent of any node can be located by ascending one indentation step in the hierarchy and moving in the direction of the root until a node at the current indentation level is found. The tight packing of nodes in indented tree plots ensures that the wasteful use of whitespace is minimal in comparison to node-link diagrams.

In addition to visualizing the aggregated amount of changed bytes per file, the application can be configured to analyze the number of revisions in which a given file was changed, disregarding the actual volume of each change. This constructs a type of heatmap and allows for the identification of "hotspots", frequently changed files that appear in multiple

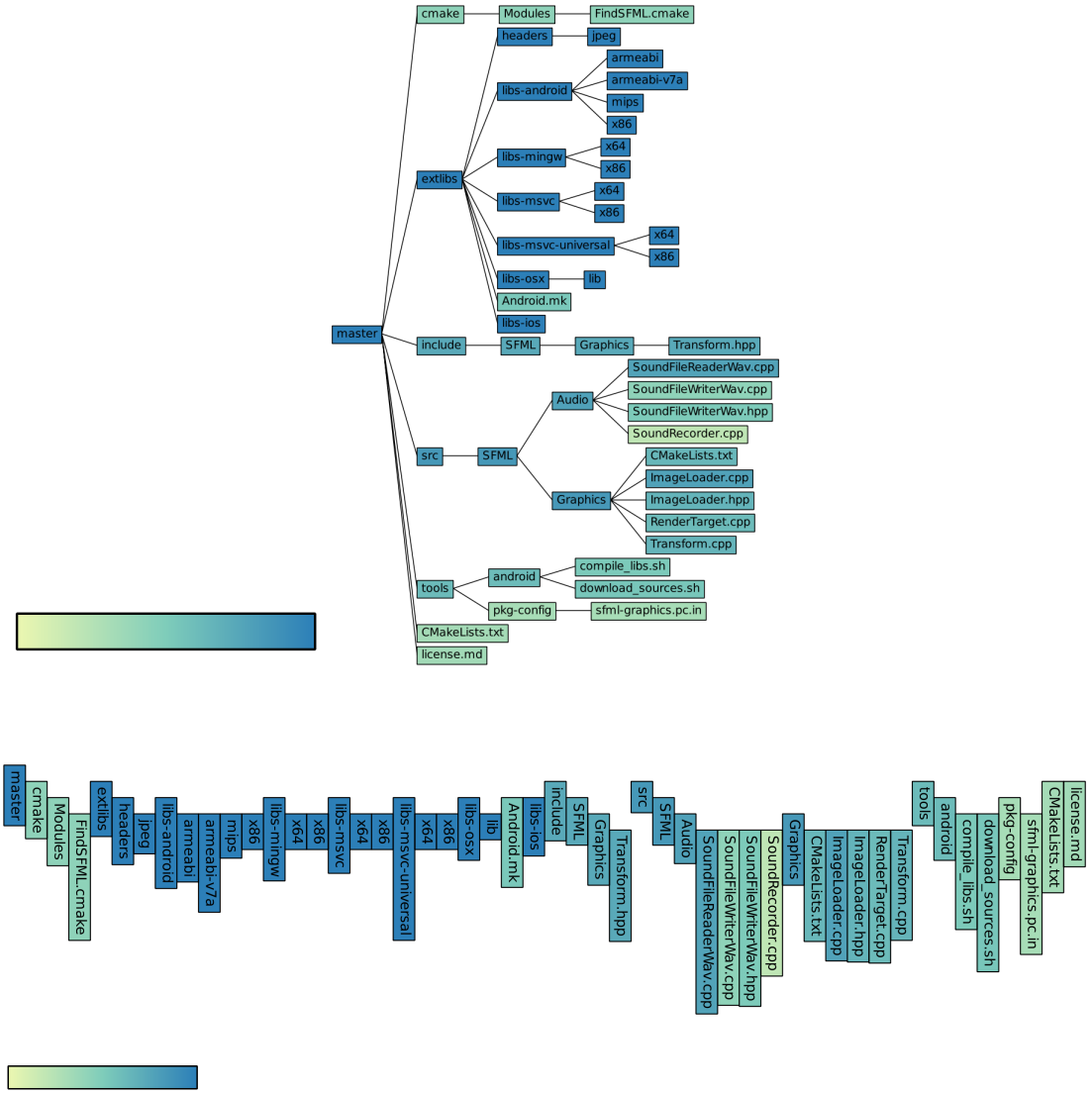

Figure 5.4: Comparative view of the same hierarchy as a node link diagram (top) and an indented tree plot (bottom).

commits within the selected timespan (fig. 5.6). This also demonstrates the use of a different color scale: rarely changed (or unchanged) files are colored white, while an increasing number of revisions containing changes to a file cause it to assume an increasingly dark red color. Directories are treated as "unchanged", as otherwise, each directory with several changed files contained within it would end up as a false hotspot.

At the default zoom level, in which all nodes are visible, labels are not readable. However, once a hotspot is located by its characteristic red coloring, the view can be zoomed in interactively, allowing for the exact node to be identified, showing which file has received a large number of changes across revisions (fig. 5.7).

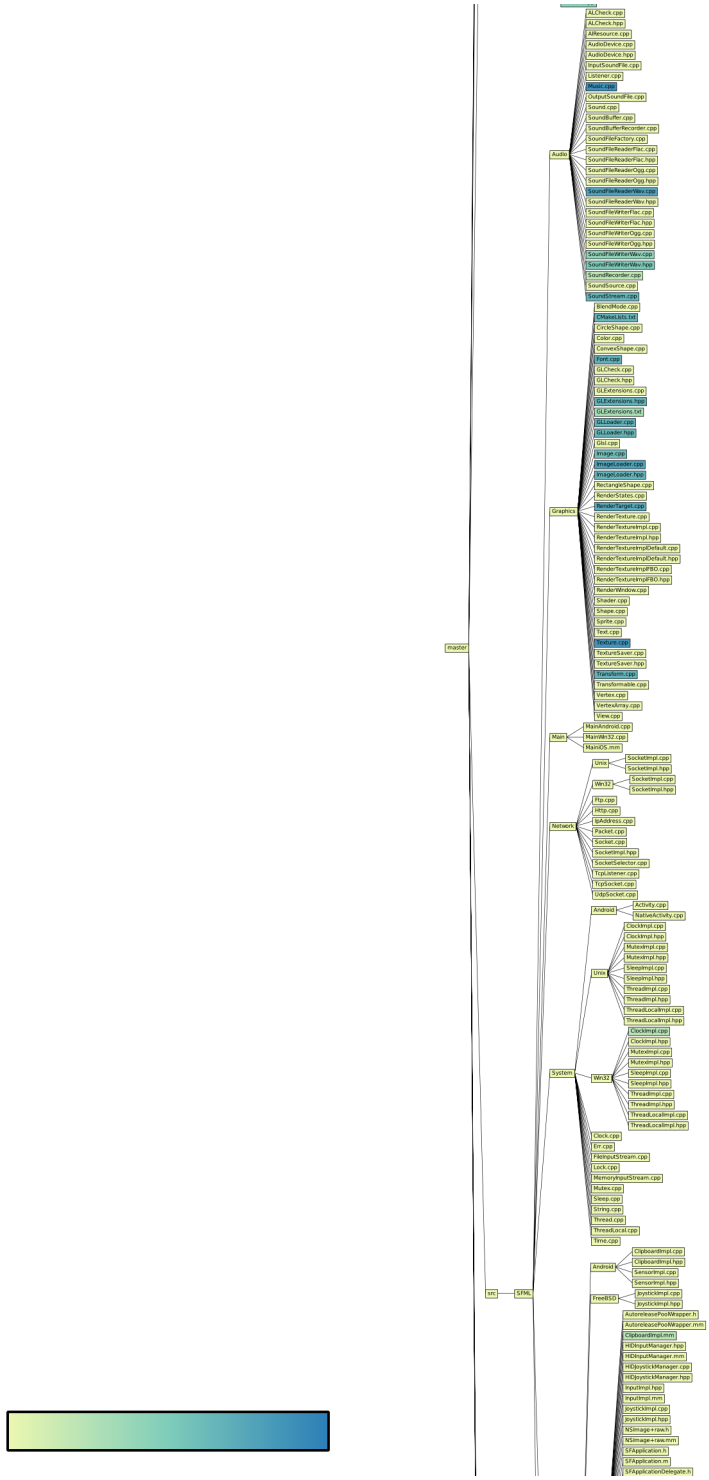

Figure 5.5: Cropped example of a node-link diagram that has exceeded a manageable size.

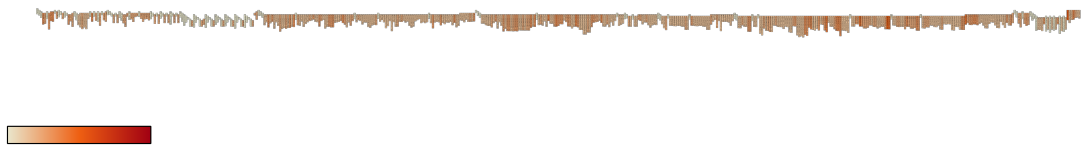

Figure 5.6: Indented tree plot displaying an overview of the number of revisions of each file within a certain range of commits.

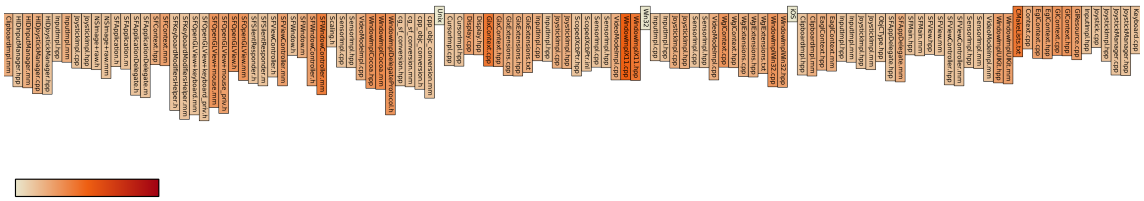

Figure 5.7: Detailed view of fig. 5.6.

5.2 Performance

In order to measure the application's performance, a series of increasingly large revision range selections was made on the same SFML dataset used for the examples. For each visualization benchmark, the hierarchies of the most recent N revisions were merged and visualized, where N is varied across benchmarks to identify performance characteristics as the scale of input data grows. The merge was repeated 3 times per benchmark and the average time of all samples was taken. Timing was performed using a nanosecond-precision system timer integrated into the application, which profiles certain time-intensive steps and prints the elapsed time to the console.

The benchmarks themselves followed the same mapping and filtering rules seen in fig. 5.2: nodes that have not been changed in any selected revision are removed, metrics are aggregated using the standard deviation formula and nodes are colorized by this aggregated metric, as well as receiving the file name as a label. The resulting hierarchy was then laid out and rendered as an indented tree plot.

The hardware used for this benchmark is a 4-core Intel Core i5-2500 CPU clocked at 3.30GHz, 12 GB of random access memory, as well as a GeForce GTX 560 Ti. Testing was performed in the application's 64-bit release build with maximum optimizations (`-O3` flag for GCC) and the resulting binary was executed on Linux Mint 18 (64-bit) with proprietary GPU drivers.

5.2.1 Data Pre-Processing

Before the repository can be visualized, a pre-processing step that iterates through the repository's history and collects metrics must be performed. As this process only has to be performed once for each repository, and is interruptible interactively, its performance is not of critical importance. Nevertheless, traversing the entire history of a Git repository, particularly one with a great number of commits and files, can take a considerable amount of time.

The repository parser took 84.68 seconds to iterate the 2033 commits of the SFML repository's `master` branch, and produced 98 MiB of persistent data. Only the byte count metric was collected from files for this benchmark, which is constant across different types and sizes of files. Actual performance depends on a variety of factors: disk read speed, file system type, type and amount of metrics collected, commit count, file count, file size, number of changed files per commit, etc.

5.2.2 Data Acquisition (Deserialization)

When visualizing a range of commits, the first step is to read the hierarchy for each revision within the selected range from the file system. This step takes a non-negligible amount of time for sizable hierarchies, and depends on the linear read performance of the hard drive used to store the parsed repository data. As each revision file contains the entire hierarchy at that point in time, the read time is roughly linear to the number of revision read, regardless of the actual quantity of changes within each commit (fig. 5.8).

5.2.3 Merging

After the in-memory tree for a given revision has been acquired, it needs to be merged into the existing supertree. This needs to be done for all revisions, again resulting in linear performance characteristics with regard to the number of revisions selected. This step is currently the slowest for the SFML example dataset, taking multiple seconds for large revision selections (fig. 5.9).

5.2.4 Mapping

Once the supertree is has been generated, it needs to be mapped to a group of graphical primitives representing each node. The performance of this transformation depends on the final node count of the tree, as each (visible) node needs to be mapped. Using filters to remove nodes from the supertree reduces the total time taken for this step. Therefore, in this performance benchmark, a filter was used to remove nodes from the hierarchy if they were not changed in any of the selected revisions, thus yielding a somewhat greater distribution of node counts to test the mapping performance of (fig. 5.10).

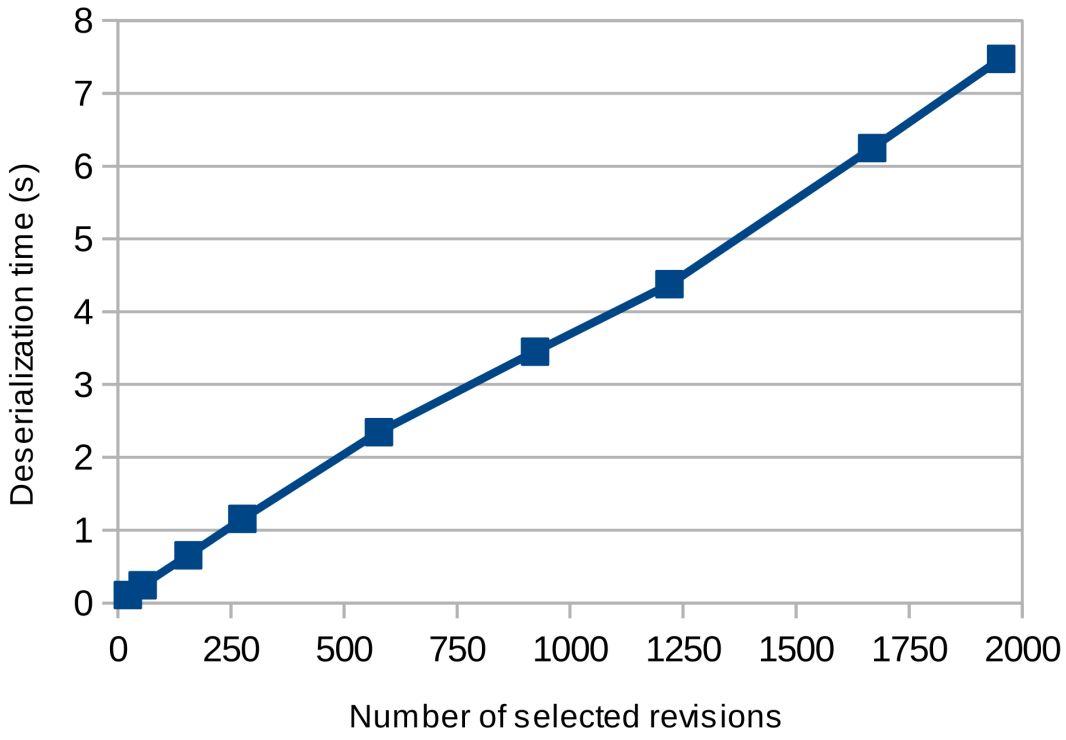

Figure 5.8: Number of seconds taken to read and deserialize hierarchy data for revision selections of varying sizes.

The most computationally complex part of this step is the construction of character polygons for label rendering. As such, performance improves for shorter labels, or if labels are omitted entirely.

5.2.5 Rendering

The final and most performance-critical step of visualization is the rendering step. While the aforementioned three steps make up the majority of waiting time from selecting a range of revisions to being able to explore the visualized merged hierarchy, this only needs to happen once per selection. The rendering step, however, is executed for each user interaction, and its performance directly influences the perceived interaction smoothness. A considerable amount of effort was spent optimizing the rendering step to produce sufficiently fast results. The target performance for these optimizations was a stable 60 frames per second during view panning interactions, as this is the refresh rate of most computer monitors, which translates to a maximum rendering time of $\frac{1}{60}s \approx 0.0167s$.

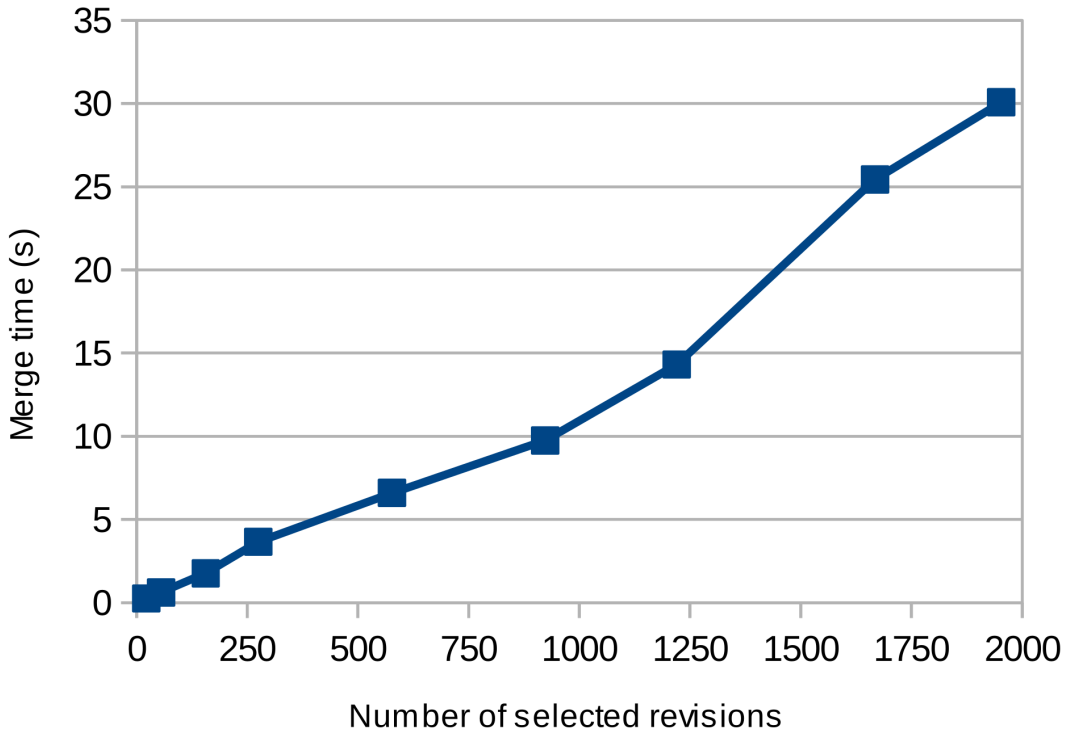

Figure 5.9: Number of seconds taken to merge hierarchy data into a supertree for revision selections of varying sizes.

Even for moderately large indented tree plots with about 3000 nodes, this performance goal was met successfully on the machine used for this benchmark (fig. 5.11). As zooming in causes only visible chunks to be rendered, even larger hierarchies can be rendered at a comparable performance when zoomed in sufficiently. This ensures that interaction with the visualization is smooth. An issue with Qt's OpenGL renderer causes occasional stutters when zooming. This is further described in section 5.3.

5.2.6 Raw Benchmark Results

The average measurement results for hierarchy visualization are given in the following table.

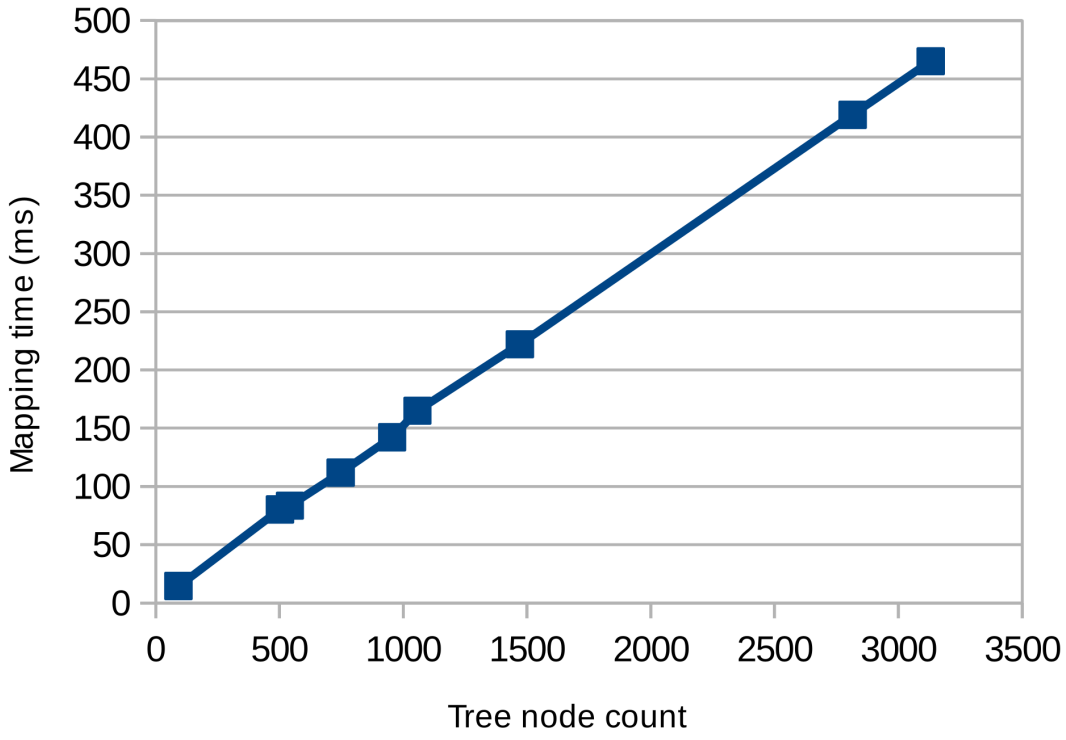

Figure 5.10: Number of milliseconds taken to map node attributes to graphical primitives for hierarchies with varying node counts.

Revisions	Node count	Deserial. (ms)	Merge (ms)	Map (ms)	Render (ms)
22	92	108.85	255.72	14.56	0.22
55	502	243.85	618.89	80.37	1.35
156	542	655.32	1768.03	83.49	1.54
275	747	1154.47	3662.85	111.81	2.00
577	955	2344.67	6602.80	142.36	2.31
923	1057	3452.45	9746.98	165.03	2.36
1220	1471	4377.20	14309.68	222.05	3.33
1668	2815	6248.50	25443.15	419.06	6.09
1953	3130	7474.62	30071.23	465.04	6.18

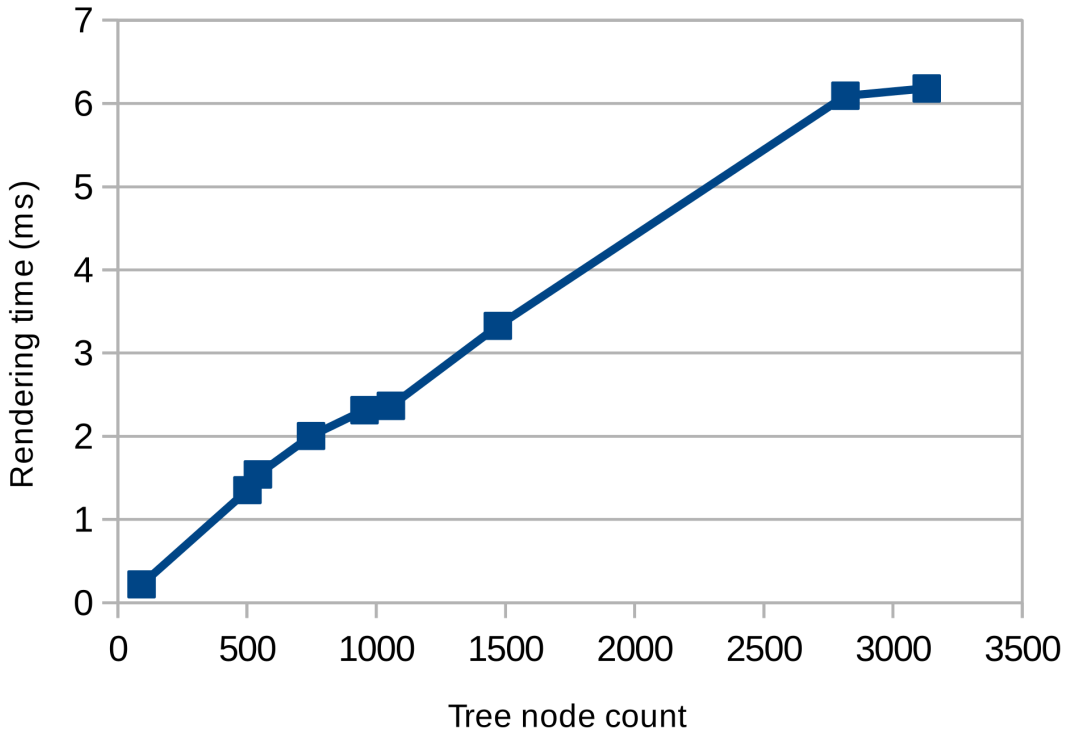

Figure 5.11: Number of milliseconds taken to map node attributes to graphical primitives for hierarchies with varying node counts.

5.3 Problems and Limitations

While the application produces reasonable results for the tested input data, the underlying merging algorithm has a few flaws that can cause misleading features to appear in the final visualization.

The Jaccard-based tree merging algorithm does not identify renamed leaves, meaning that renamed files (which are represented as leaves in the hierarchy) will appear as entirely different files starting from the revision that renamed them. This not only causes the files to appear twice if such a revision is included in the merge input range, but also falsifies the resulting metrics. To mitigate this issue, the fact that renaming detection function correctly for inner nodes can be taken advantage of: attaching content-dependent data to each file, such as its abstract syntax tree if the file contains source code, would allow the merging algorithm to identify mergable files based on the similarity of nodes contained within the file's own subtree.

Another problem related to the merging algorithm is its inability to detect file and directory relocations. Moving a subtree from one node within the hierarchy to another will cause the whole subtree to be duplicated and its metrics decoupled, as only sibling nodes are taken into consideration as merging candidates. Solving this issue would require the use of a more sophisticated algorithm, such as a variant of Tree Edit Distance that allows backtracing the steps to transform one hierarchy into another, granting information on which nodes should be merged together.

Additionally, the current implementation of the Jaccard-based tree merging algorithm exhibits poor performance for very large directories containing large numbers of immediate files. As the pairing algorithm to find the most similar sibling has a worst-case quadratic number of iterations with respect to the number of sibling nodes, multiplied with each node's total leaf count for actually computing the Jaccard similarity coefficient, this results in a large performance degradation as the number of files within a directory grows. This issue can be mitigated by immediately matching nodes with identical labels, before applying the measure of Jaccard similarity to match any remaining nodes, therefore minimizing the number of expensive operations required for the common case of few changed nodes with a large number of unchanged siblings.

Another potential issue is the memory usage of the supertree's various intermediate representations: the abstract tree itself, auxiliary data for layout purposes and the graphical primitives for nodes and links all have to fit into main memory. In particular, very large numbers of mapped nodes can cause high amounts of random access memory to be used for storing label polygons. A way to reduce memory usage is to implement lazy mapping, where node labels are not mapped until the view itself is sufficiently zoomed in that only a subset of nodes are visible in the cropped view, and then only generating text polygons for those visible nodes.

While the process of data pre-processing displays a progress bar and is interruptible at any point, this does not apply to the revision selection process. Once a range of commits is selected, the application becomes unresponsive until the full acquisition, deserialization, merging, filtering, mapping and rendering steps have been completed, which may take several seconds (as discussed in section 5.2). To solve this issue, all algorithms would have to include a call to an interruption check function, which would be controlled by a progress window with a "Cancel" button that sets an interruption flag, similar to the repository parser's interruptible implementation.

Qt itself appears to suffer from a performance issue that causes occasional stutters when very large indented tree plots or node-link diagrams are zoomed. Certain functions within Qt's OpenGL backend will sometimes hang for several seconds at a time while drawing diagrams with roughly 3000 nodes, causing the application become unresponsive. The cause of this bug is currently unknown.

Finally, the repository parser operates under the assumption that commit timestamps are unique. However, in some cases, two or more commits may share the same timestamp, therefore overwriting one another's revision files. A solution to this problem would

be naming intermediate revision files by their commit hash, rather than the revision timestamp.

6 Summary

As software projects are worked on over the course of several years of development time by multiple contributors, the structure and contents of their source code undergo significant changes, which are typically tracked using version control system. The history of such a software repository, modeled as a time-indexed list of hierarchies and enriched with metrics at file level, can serve as a data source for an explorative visualization, granting insights into the evolution of the software system and identifying trends among metrics.

The goal of this thesis was the design and development of a scalable application for visualizing hierarchies acquired from software system source code as they undergo changes over time. A specific focus was placed on visualizing the hierarchy over time rather than only looking at isolated individual snapshots at a specific point in time. By implementing the full visualization pipeline, including pre-processing a repository and acquiring metrics, as well as providing customizable methods of temporally and spatially aggregating attributes, the application developed as part of this thesis aims to fulfill the use case of interactively exploring a range of hierarchies as one merged "supertree".

Given a valid Git repository, the application allows the user to pre-process the entire history of the repository into an intermediate format while collecting metrics from each file at each of its revisions. The pre-processed hierarchy is persistently stored and can be loaded into the application for visualization. The user can then select any range of revisions from the repository, the working trees of which will be merged into a single hierarchy that is then visualized. Aggregated metrics are mapped to visual attributes on a per-node basis, and a variety of options for customization is provided, enabling the user to tweak parameters at each stage of the visualization pipeline. The rendered hierarchy, laid out either as a node-link diagram or indented tree plot at the user's choice, can be interactively explored, ranging from a complete overview of the whole structure to a detailed look at individual nodes of interest.

The tool itself was developed as a C++ application using the Qt library and implements most relevant requirements specified in section 3.1. It was tested using the open-source library SFML as a data source, which provided a dataset consisting of around 2000 revisions. Testing showed that the visualization of aggregated metrics is functional from a technical perspective, and that various points of interest, such as clusters of files with high amounts of changes in a certain timespan, can be visually identified and inspected closely. The performance of the visualization is also sufficient to enable smooth user interaction.

6.1 Future work

The application's modular and extensible design allows for the future implementation of more repository parsers (such as Subversion or Mercurial), enabling a greater range of source code repositories to be used as a dataset for visualization.

Additionally, implementing the full range of metrics specified in section 3.1 with the help of an abstract syntax tree parser (such as libclang) will allow for more thorough and specific insights to be made into the code quality of C and C++ projects, unifying static analysis and big data visualization.

Moreover, the concept of hierarchy evolution visualization can be applied not only to metrics derived using static analysis, but also measurements taken at runtime, such as the results of performance benchmarks. Adapting the tool to collect runtime performance metrics and visualizing their quantitative changes over time, while maintaining logical integrity if such metrics are added or removed throughout the software lifecycle, could enable the visual identification of performance regressions and trends over the target software's course of development.

Bibliography

[Bil05] P. Bille. “A survey on tree edit distance and related problems. Theor Comput Sci 337(1-3):217-239.” In: 337 (June 2005), pp. 217–239. DOI: 10.1016/j.tcs.2004.12.030 (cit. on p. 12).

[BRW10] M. Burch, M. Raschke, D. Weiskopf. “Indented Pixel Tree Plots.” In: *Advances in Visual Computing: 6th International Symposium, ISVC 2010, Las Vegas, NV, USA, November 29-December 1, 2010. Proceedings, Part I*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 338–349. ISBN: 978-3-642-17289-2. DOI: 10.1007/978-3-642-17289-2_33 (cit. on p. 14).

[GS62] D. Gale, L. S. Shapley. “College Admissions and the Stability of Marriage.” In: *The American Mathematical Monthly* 69.1 (1962), pp. 9–15. ISSN: 00029890, 19300972 (cit. on p. 17).

[HM81] W. A. Harrison, K. I. Magel. “A Complexity Measure Based on Nesting Level.” In: *SIGPLAN Not.* 16.3 (Mar. 1981), pp. 63–74. ISSN: 0362-1340. DOI: 10.1145/947825.947829 (cit. on p. 24).

[Jac12] P. Jaccard. “The distribution of the flora in the alpine zone.” In: *New Phytologist* 11.2 (1912), pp. 37–50. ISSN: 1469-8137. DOI: 10.1111/j.1469-8137.1912.tb05611.x (cit. on p. 11).

[Sch11] H. J. Schulz. “Treevis.net: A Tree Visualization Reference.” In: *IEEE Computer Graphics and Applications* 31.6 (Nov. 2011), pp. 11–15. ISSN: 0272-1716. DOI: 10.1109/MCG.2011.103. URL: http://treevis.net/ (cit. on p. 12).

[ZMC05] S. Zhao, M. J. McGuffin, M. H. Chignell. “Elastic hierarchies: combining treemaps and node-link diagrams.” In: *IEEE Symposium on Information Visualization, 2005. INFOVIS 2005.* Oct. 2005, pp. 57–64. DOI: 10.1109/INFVIS.2005.1532129 (cit. on p. 13).

All links were last followed on January 4, 2018.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature