Visually Enriching and Comparing Runtime Performance of Visualization Pipelines

Hagen Tarner[1*], Patrick Gralka[2], Guido Reina[2], Fabian Beck[3], Steffen Frey[4*]

[1*]University of Duisburg Essen, Schützenbahn 70, Essen, 45127, Germany.
[2]Visualization Research Center, University of Stuttgart, Allmandring 19, Stuttgart, 70569, Germany.
[3]University of Bamberg, An der Weberei 5, Bamberg, 96047, Germany.
[4*]University of Groningen, Nijenborgh 9, Groningen, 9747, Netherlands.

*Corresponding author(s). E-mail(s): hagen.tarner@paluno.uni-due.de; s.d.frey@rug.nl;
Contributing authors: patrick.gralka@visus.uni-stuttgart.de; guido.reina@visus.uni-stuttgart.de; fabian.beck@uni-bamberg.de;

Abstract

In many graphics and visualization frameworks, directed acyclic graphs are a popular way to model visualization pipelines. Pipeline editors provide a visual interface to modify the underlying graph and use, e.g., a node-link diagram to visualize the data and program flow. This paper proposes an interactive tool for post-mortem performance analysis and comparison of pipeline variants. We extend the node-link representation of a visualization pipeline and enrich it with fine-grained runtime performance metrics. Annotating this static structure with dynamic performance information lets the developer evaluate performance characteristics in depth. Our approach further supports a visual comparison of two user-selected states of the graph. The comparison allows us to identify the impact of (topological) changes on the graph's performance. We demonstrate the utility of our approach with different scientific visualization use cases and report expert feedback.

Keywords: Software Visualization, Visualization of time-varying data, Scientific visualization, Visual models and representations

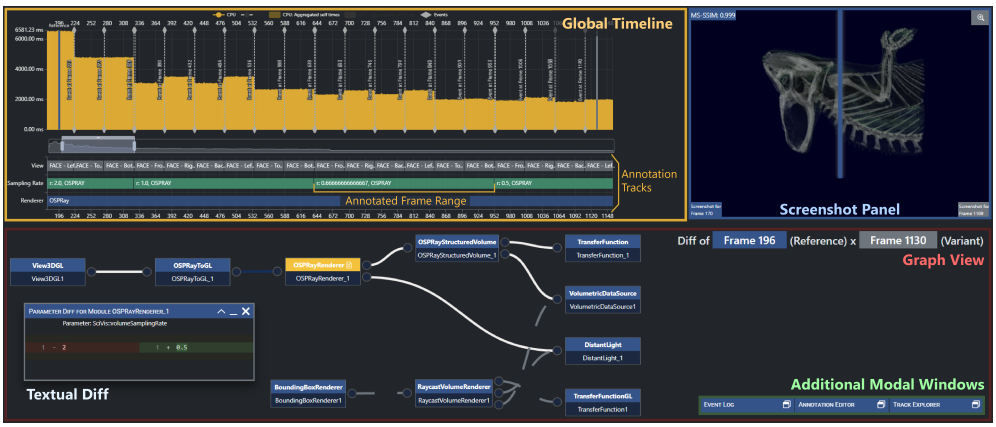

Figure 1: Our proposed approach allows fine-grained performance analysis of visualization pipelines. The *Global Timeline* depicts frame times and user-generated annotations of frame ranges, the *Screenshot Panel* presents the pipeline's rendered output (showing the "Veiled Chameleon" data set[1] as example), and the *Graph View* encodes performance information in the visual attributes of the node-link representation of the pipeline. This screenshot shows the topological and performance diff of two frames in the *Volume Rendering* application example (see Sec. 5.1).

1 Introduction

In scientific visualization, faster rendering speeds and improved interactivity of a visualization system allow researchers to visualize and navigate complex data sets more swiftly. Common metrics to judge the runtime performance of a visualization pipeline are *frames per second* or the time it took to generate the last frame (*frame time*). The aggregated performance of the stages of a visualization pipeline for a single frame provides a good overview. However, the per-frame aggregations lack the detail to explain individual parts of the rendering process and their contribution to the overall performance. Hence, this work takes a more fine-grained approach to analyzing the runtime performance of a visualization pipeline and applies meaningful levels of aggregation to allow users a drill-down into the data at its finest granularity, while still enabling frame-of-interest identification across the whole data set.

In a visualization pipeline, the steps to generate an image are conceptually organized consecutively: data from a source is filtered, mapped to a renderable representation, and rendered to a display. Each of these fundamental steps is typically implemented using a combination of individual components executed in a specific order, conceptually forming a data flow network. In this work, we call each basic component of one such network a *module*, and one step of a visualization pipeline can comprise multiple modules (see the graph in Fig. 1). Data passes through the network via *connections* between modules. The network can be effectively modeled as an acyclic graph and visualized as a node-link diagram. This diagram consists of nodes (also known as *vertices*) connected via

[1] Dr. Jessie Maisano, 2003, *Chamaeleo calyptratus* (On-line), Digital Morphology: https://digimorph.org/specimens/Chamaeleo_calyptratus/whole/, last accessed 2025-03-13.

links (also known as *edges*) to depict the connections or associations between them. Mapping the concrete architecture of a visualization tool to this concept, we can record fine-grained performance data on the connection level and encode this information in the visual attributes of the node-link diagram. This allows for the combined evaluation of the pipeline topology and the dynamic runtime data of its performance.

A crucial step for evaluating a pipeline's runtime performance is contextualization by comparing it to other pipeline variants regarding certain parameter settings or module connectivity. Comparing different states of a pipeline allows reasoning about the impact the changes have on the pipeline's runtime. For this, we enrich the combined display of the pipeline's fine-grained performance data and static structural data with a unified diff view of two variants of the same pipeline. This way, we can visualize changes in topology alongside changes in runtime performance and link these two dimensions of analysis. Changes to the structure or parameters of a pipeline do not only affect its runtime but can also affect the rendered output. The computed visual output of a pipeline serves as a check for the pipeline author (*Does my pipeline work as intended?*) while being the final product of a pipeline's execution and the main reason for authoring a pipeline in the first place. This makes the rendered output an essential dimension of analysis when evaluating a pipeline's execution. For this, we include the rendered image plus image comparison possibilities to our proposed approach.

Following these ideas, we propose a novel post-mortem analysis tool (see Fig. 1) for fine-grained performance data depicted on a graph representation based on visualization pipeline editors. After describing related work (Sec. 2), we discuss the user requirements, design choices, and implementation of our tool, which we collectively consider to be our main contributions:

- The design of our tool is generic; while we describe performance data collection and results with the visualization framework MegaMol (Grottel et al, 2015), it could easily be used with any visualization software modeling visualization pipelines as acyclic graphs (like VTK, Paraview, Inviwo, etc., Sec. 3.1).
- Our task analysis (Sec. 3.2) targets developers and pipeline authors, identifying three main relevant kinds of features for these users: overview, single pipeline state analysis, and pipeline state comparison.
- Our tool improves upon prior work in various regards: it provides a variety of views for different perspectives on states (module graph, screenshots, performance bar/line charts, etc.), facilitates comparison of arbitrary states, and supports the analysis of both parameter changes and graph structure changes (Sec. 4).
- The tool was designed and developed in close collaboration with domain experts. Sec. 5 describes the interactions and outcomes of this collaboration.

We discuss these contributions together with ideas for extension in Sec. 6 and conclude our work in Sec. 7. The tool, the used data sets, and a video demonstration are publicly available at https://doi.org/10.18419/DARUS-4115.

2 Related Work

Our work relates the usage of node-based editors for visualization pipeline modeling to (visual) performance analysis. While we discuss relevant examples of node-based editors,

performance analysis methods, and performance visualizations, we are not aware of any works that address the combination of these and target a similar scenario to ours.

2.1 Node-based Editors for Visualization Pipelines

As established before, a pipeline can be modeled as a graph, typically visualized as a node-link diagram. In scientific visualization, using visual editors to create and edit such node-based pipelines is a common practice (Moreland, 2013). Each available editor depicts the visualization pipeline in its own way and focuses on providing users with specific features. As our approach tries to relate changes in performance to changes in a pipeline's topology, the depiction of pipeline performance and possible comparison of pipeline variants are relevant. MegaMol (Grottel et al, 2015) is a full-fledged visualization system for data from different scientific fields, e.g., astronomy, architecture, and flow visualization. It does not offer a comparison of pipelines and reports pipeline performance on a per-frame basis (fps and frame times). VisTrails (Bavoil et al, 2005) separates pipeline specification and execution to provide reproducibility and provenance data on pipeline creation. VisTrails has visual comparison features and focuses on parameter space exploration, versioning, and provenance, but lacks performance visualization. Inviwo (Jönsson et al, 2020) represents the visualization pipeline in different levels of abstraction, giving users with various levels of experience full flexibility to create highly optimized pipelines. It captures few statistics regarding execution times per module, but does not provide means for analyzing them. VisIt (Childs et al, 2012) presents a distributed infrastructure for processing visualization pipelines on petascale HPC platforms. It offers three kinds of comparison: image-level, data-level, and topological-level (topology of the data sets). Fine-grained performance data in combination with the visualization pipeline is missing. Lastly, ParaView (Ahrens et al, 2005) has been extended (Petersen et al, 2023) to allow visual programming of the underlying pipeline by adding a visual node-link editor for pipeline authoring. It offers execution time logging and generates statistics from the logs.

2.2 Benchmarking and Performance Modeling

Numerous papers advocate for an empirical approach to performance evaluation. Runtime performance of visualization applications typically varies significantly due to factors such as data, parameters, and hardware, making it challenging to model and predict. This has been well-studied especially for volume rendering as a fundamental method in scientific visualization. Bethel and Howison (2012) examined various settings, algorithmic optimizations, and memory layouts for the Intel Nehalem, AMD Magny Cours, and NVIDIA Fermi architectures. They found that optimal configurations vary across platforms in unexpected ways. This analysis of performance characteristics is crucial for parameter tuning, hardware selection and developing improved algorithms (Bentes et al, 2014). Bruder et al (2020) reviewed 50 scientific visualization papers on performance analysis along with guidelines for performance evaluation.

Thorough performance analysis is essential for predicting and modeling application performance in computer architecture and high-performance computing. Researchers have proposed both generic performance models (Hong and Kim, 2009; Deakin et al,

2016) and models tailored to specific architectures, such as NVIDIA GPUs (Zhang and Owens, 2011). These approaches often use micro-benchmarks: small, specific code segments to test particular characteristics. While these models generally target broad computing applications, significant work focuses on scientific visualization, especially in high-performance computing. For instance, for GPU clusters, Rizzi et al (2014) developed an analytical model to predict the scaling behavior of parallel volume rendering, and Tkachev et al (2017) used neural networks to predict the impact of different compute hardware on runtime performance.

2.3 Visual Performance Analysis

Performance visualization techniques are useful for evaluating a software system's behavior and runtime characteristics. Numerous approaches exist to visualize performance based on the structure of the underlying data and the application's domain (Isaacs et al, 2014). Examples include memory performance (*MemAxes* (Giménez et al, 2018)) and I/O performance analysis (*Gauge* (Del Rosario et al, 2020)) in an HPC context.

Petersen et al (2023) also provide a method for visualizing performance data of a visualization pipeline on the node graph representation. In contrast to our method, it does not support considering and comparing changes to pipelines but only parameter changes inside a static set of employed filters.

Tracy[2] is among the most popular free, fine-grained profilers. It includes a *flame graph* (Gregg, 2016), one of the most widely adopted hierarchical visualizations for performance data. This technique is also present in notable closed-source alternatives like Nvidia NSight, Intel VTune, Microsoft Visual Studio / PIX. Tracy further allows the basic comparison of two traces. Tracy also shows a source code diff to correlate performance changes to source code changes. Tracy is, however, not meant to analyze visualization pipelines or any other graph-based structure specifically.

NVIDIA Nsight Graphics[3] also allows the profiling of single frames for calls to graphics APIs like OpenGL, Vulkan, and Direct3D. If code decorates the OpenGL rendering with (nested) *DebugGroups*, these are employed to generate a flame graph in the tool's timeline, providing context for the actual OpenGL command stream, and allowing the detection of costly parts of the pipeline for a single frame. To the best of our knowledge, there is no support for comparing multiple frames with respect to their performance and no way of tying visualization pipeline parameters into the analysis, let alone pipeline changes.

The evaluation of performance data requires contextualization. This includes a comparison of a reference execution to one or more variants. Visual comparison (Gleicher et al, 2011) has been used to support this contextualization in different application scenarios, e.g., guide the user to find the best-performing image processing pipeline (Ikarashi et al, 2021), visualizing the evolution of software systems (Tarner et al, 2020), comparing different rendering techniques in a benchmark suite (Tarner et al, 2022), or visualizing the evolution of graphical software models (Pietron et al, 2022). The latter also includes the visual comparison of node-link diagrams and constitutes a starting point for the design of our Pipeline Diff Mode (see Sec. 4.3). Visual

[2]Tracy code repository: https://github.com/wolfpld/tracy, last accessed 2025-03-13.
[3]NVIDIA Nsight Graphics: https://developer.nvidia.com/nsight-graphics, last accessed 2025-03-13.

comparison in the context of performance visualization often also includes comparing hierarchical data (Zhuang et al, 2008; Sandoval Alcocer et al, 2013; Trümper et al, 2013; Bezemer et al, 2015). Sandoval Alcocer et al (2019) integrated multiple comparison facets (fine-grained runtime metrics, call graphs, code changes) into a matrix-like representation for tracing performance across several versions.

3 Data Model and Analysis Tasks

Established profiling tools such as *Tracy* and *flame graph* provide fine-grained performance data. Granularity can be down to the level of individual function calls or even single lines of code. While this level of detail is crucial for performance analysis, it can be challenging to interpret in the context of complex visualization pipelines. To improve the usability of such data, we leverage the abstraction level of node-based editors—an established technique for visualizing pipelines, commonly used in scientific visualization (see Sec. 2.1) and industry-standard software such as Blender,[4] Unreal Engine,[5] and Houdini.[6] Integrating the fine-grained performance data with the structured visualization capabilities of node-based editors, we want to ease certain performance analysis tasks. This includes providing an overview perspective across time as well as supporting a detailed inspection of certain steps in time. Additionally, visual comparison mechanisms are required, allowing users to contextualize performance data more effectively and identify optimization opportunities.

3.1 Data Model and Metric Recordings

The data sets for the prototype comprise two primary types of data. First, a profiling log, stored in CSV format, which encompasses fine-grained performance data on a per-frame basis (timestamps). Second, a set of screenshots, with each screenshot capturing the rendered output and containing the graph structure as meta-data (Reina, 2023). To increase reproducibility, we used benchmark scripts to generate the data we discuss here. The approach, however, also allows recording data during interactive sessions and is not limited to pre-defined executions.

Timestamps are recorded at the invocation and completion of functional code units representing the nodes of the acyclic module graph that describes the visualization pipeline. MegaMol models the pipeline as a graph consisting of *Modules* and *Calls*. All Calls derive from a single class, providing a minimally intrusive entry point for collecting performance data (`Call`). Timers measure the inclusive runtime of modules and downstream modules, covering any visualization pipeline in MegaMol. A central profiling subsystem collects and processes all measurements. CPU timings are captured using the standard C++ steady clock. GPU timings are collected using query objects that record timestamp events in the OpenGL command stream. Timer invocations are explicitly associated with their respective frames, allowing for delayed evaluation. The timer implementation is generic and reusable with other visualization systems, requiring only a change to the parent (owner) pointer.

[4] Blender product website: https://www.blender.org, last accessed 2025-03-13.
[5] Unreal Engine product website: https://www.unrealengine.com, last accessed 2025-03-13.
[6] Houdini product website: https://www.sidefx.com/products/houdini/, last accessed 2025-03-13.

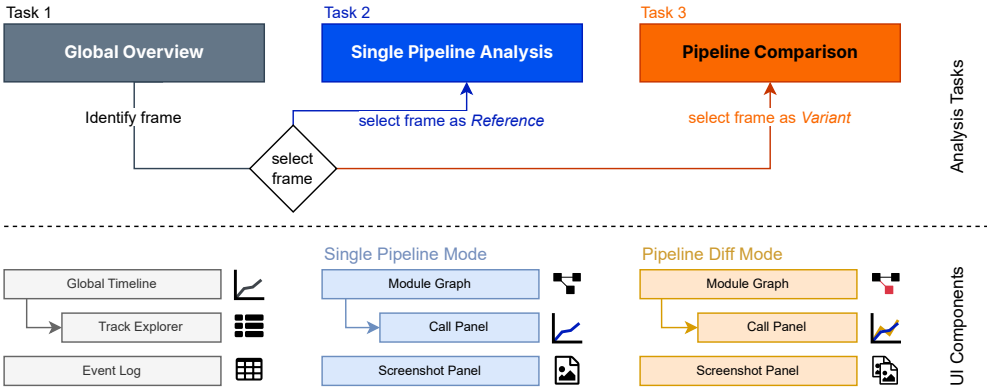

Figure 2: Supported analysis tasks. Task 1 (*Global Overview*): identify the frame for in-depth analysis in the set of all recorded frames. Task 2 (*Single Pipeline Analysis*): load pipeline state, render output, and performance metrics for the selected frame. Task 3 (*Pipeline Comparison*): activate diff mode when a second frame is selected and display the explicitly encoded structural and behavioral diff of the pipeline and rendered outputs. Each task has a set of associated UI components.

In addition to Call-based timings, we record whole-frame costs between consecutive frames by capturing timestamps on both CPU and GPU. These timestamps include potential overhead beyond graph evaluation, such as frame buffer management. We synchronize rendering to frame boundaries by flushing the OpenGL pipeline at the end of each frame, ensuring accurate whole-frame costs. Without this synchronization, OpenGL command execution may extend beyond the frame buffer swap request, leading to inconsistent frame costs and noticeable discrepancies after parameter changes.

Performance logs also contain every parameter and graph topology change. This list of graph change events is used to re-run each benchmark script to collect screenshots for these frames without influencing the profiling logs. The screenshots store metadata comprising the current graph state, and our analysis tool extracts these graph states for use in diffs and the Graph View.

3.2 Analysis Tasks

The target group of our proposed approach comprises domain experts, performance analysts, and pipeline authors. They all share expertise in rendering and are familiar with modeling a visualization pipeline as an acyclic graph.

The main goal of our approach is to enable members of our target group to generate fine-grained insight into the performance of their pipeline. We want to allow them to inspect performance not only per frame but also to dig deeper and visualize what contributed to each frame's runtime cost at an abstraction level consistent with the framework they are working with. Based on the findings of Petersen et al (2023) and our own experience from visualization framework development, we identified the following

list of tasks as relevant. Fig. 2 gives an overview of the analysis tasks and maps each task to a set of UI components, which will be discussed in more detail in Sec. 4.

Analysis Task T1, *Global Overview*: analyze the performance profile across the entire data set. The pre-recorded performance measurements that serve as the data sets for our approach can contain an arbitrary number of frames, and each frame contains several fine-grained runtime performance metrics depending on the complexity of the pipeline. Hence, a global overview of the data set is needed for the user to navigate the recorded metrics. To not overwhelm the user with too much information, the overview should show performance data aggregated on a per-frame basis instead of on a per-module basis. Given this overview, the user can identify interesting frames for further inspection and analyze them in more detail.

Analysis Task T2, *Single Pipeline Analysis*: closely inspect the pipeline at a single frame wrt. performance-critical characteristics. For fine-grained analysis, the approach should mimic the pipeline's visualization, which the user already knows from the pipeline editor. This way, we can reuse the user's mental model of the pipeline and annotate the familiar node-link diagram with our fine-grained performance metrics. This also offers the benefit of presenting the metrics on a granularity level already known to the user, as it is the same level at which they authored the pipeline. The final level of detail (per module invocation) for the recorded metrics should be accessible from within the node-link diagram. This requirement subsumes T1 and T2 from Petersen et al (2023). We aim for comparison across one complete pipeline as well as the detection of outliers (i.e., the slowest module).

Analysis Task T3, *Pipeline Comparison*: compare the pipeline at two different frames to evaluate the impact of graph changes on changes in performance and rendered output. To contextualize the analysis, we give the user the option to load two recorded frames and compare them with respect to topology, parameter settings, runtime performance, and rendered output. This diff view should support the visual comparison of all the mentioned dimensions of analysis. We build upon T4 from Petersen et al (2023) by analyzing the influence of parameters within modules. Additionally, we examine the effects of modifying or replacing parts of a pipeline, such as changing the rendering backend or adding views, and explicitly analyze these differences.

4 Visualization Approach

To address the analysis tasks **T1**–**3**, as described in Sec. 3.2, we created an interactive analysis tool for post-mortem analysis and comparison of fine-grained performance data described in Sec. 3.1. The tool is built using *ECharts* (Li et al, 2018), *Vue.js*,[7] and *Electron.js*.[8] Fig. 2 maps the analysis tasks to specific UI components of the tool.

4.1 Global Overview

To give a general overview of all recorded metrics in a loaded data set (**T1**), we offer three linked visualizations. The Global Timeline (see Fig. 8 top) shows different

[7] Vue.js product website: https://vuejs.org/, last accessed 2025-03-13.
[8] Electron.js product website: https://www.electronjs.org/, last accessed 2025-03-13.

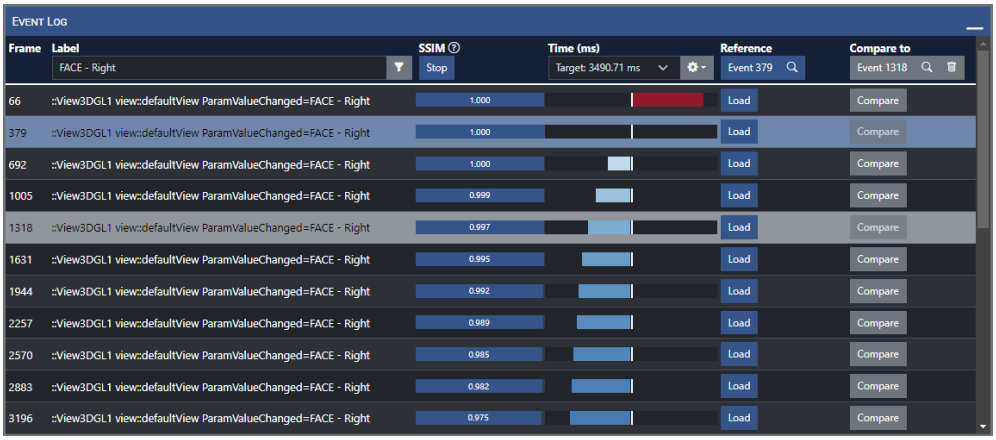

Figure 3: Detail view of the *Event Log* UI component. It uses a visual table to present information about programmatically recorded graph change events. Each table row includes the frame number, a textual diff, SSIM values for image comparison, and a visual representation of the performance diff, relative to the selected frame (379).

frame times for the whole data set in an annotated timeline, the Track Explorer (see Fig. 8 bottom) uses boxplots to generate statistics over user-defined frame ranges, and the Event Log (see Fig. 3) summarizes recorded graph changes.

The Global Timeline serves as a general starting point for an analysis session. Its main goal is identifying frames for the detailed performance analysis in task **T2**. It is an interactive timeline that annotates global frame times with recorded graph change events. It includes multiple metrics: frame times in milliseconds for each frame in the data set (whole-frame costs), separated by the used hardware (*CPU* or *OpenGL*) as a line chart, and one bar per API that shows the sum of all module self times per frame. A gap between bars and line charts shows that MegaMol spent time outside the graph to complete rendering a frame (see Fig. 4).

For further orientation in data sets with a high number of frames and to create an intuition for the performance profile of the data set under analysis, the Global Timeline offers grouping frames into frame ranges and providing textual annotations for these ranges. Each annotation has a label, start and end frame, and is associated with one *Annotation Track* (see Fig. 8). A typical use case for the annotations in a benchmarking scenario would be to create one track per parameter (e.g., data set, resolution) and one annotation per parameter value (e.g., *Resolution: 1080p*, *Resolution: 2160p*). The annotated frame ranges can then be further analyzed via the Track Explorer (see Fig. 8 bottom). Each track annotation generates a pair of boxplots, separated by employed hardware, and provides descriptive statistics for the whole-frame costs of all frames in the selected range. This gives the analyst a quick way to correlate parameter settings to overall frame times. Annotations and tracks are user-defined and editable via a simple UI (see Fig. 4). They can be persisted for re-use in a later analysis session.

The third UI component to address **T1** is the Event Log (see Fig. 3). It is a visual table that displays the automatically recorded graph change events. It displays the

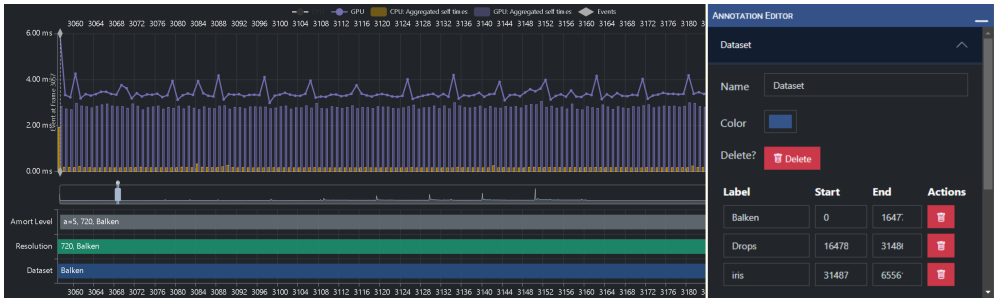

Figure 4: Beam data set rendered at 720p, amortization level 5. Performance spikes are not reflected by the intra-graph GPU costs, but suggest a certain regularity. Right side shows the editor for the annotation tracks.

frame number at which the change occurred and a textual summary of the parameter changes for every event. It also visualizes the structural similarity in rendered outputs for all events by calculating the pair-wise MS-SSIM (Wang et al, 2003) values for the selected event and all other events (see the third column in Fig. 3). The fourth column of the Event Log contains an inline visualization to quickly compare performance characteristics across events. It uses a diverging color scale with the currently selected event's frame as the reference to show increased or decreased whole-frame costs for all other events. Scrolling through the Event Log thus gives the analyst a quick overview of performance differences and rendered output of the whole data set wrt. the currently selected event. Events are also marked in the Global Timeline via a vertical line (see dashed gray lines in Fig. 8 top).

4.2 Single Pipeline Analysis

For the drill down into the fine-grained performance data of a single frame, we offer three main UI components: the *Graph View*, the *Call Panel*, and the *Screenshot Panel*. When only a single frame is selected, these components operate in *Single Pipeline Mode*, in contrast to the *Pipeline Diff Mode* (see Sec. 4.3), where also a second frame is to be selected.

Clicking a frame in the Global Timeline or an event in the Event Log loads the pipeline state of the respective frame into the Graph View for further analysis (**T2**). The Graph View shows the topology of the graph and the recorded performance data for the selected frame. The graph is visualized as a node-link diagram at the center of the screen. Each Module is represented as a box, and each Call is depicted as a line between modules. As connections can be typed and modules can support multiple inputs and outputs, we add connectors for each type of incoming (left side) and outgoing (right side) connection to each module. We use the *dagre*-layout to calculate the positions of modules and connections. This type of layout is commonly used for directed graphs. It focuses on preserving ranks across the paths through the graph and

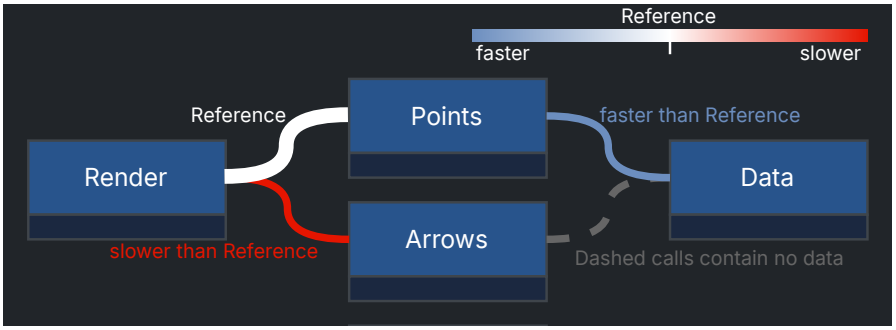

Figure 5: Color encoding in single view mode: a diverging color scale (blue, white, red) depicts runtime performance of calls relative to a user-selected reference call. Calls without performance data in the selected frame are shown in gray.

effectively avoids overdrawing modules on top of each other. For the computation, we use the JavaScript-based implementation `dagre-layout`.[9]

To encode the performance metrics inside the Graph View, we use a diverging color scale (blue, white, and red) for the links (see Fig. 5). The scale's center point (white) is set to the performance metric of the selected reference call. Calls with a runtime metric below the reference are colored blue, and values above are red. In its initial state, the most expensive call is selected as the reference. This results in all other calls being colored in shades of blue. Calls in the data set that do not contain performance metrics for the selected frame are visualized with a dashed gray line. In Single Pipeline Mode, the reference call is interactively selected. Coloring of all calls updates when the analyst selects a call for further analysis.

To inspect the recorded performance data in its finest granularity, the analyst can open the *Call Panel* by clicking a Call in the Graph View. The Call Panel contains the timings for the selected call, separated by Callback (the requests a call can invoke) and executing hardware. Each Callback is visualized as a line chart, with the frame number on the X-axis and the frame time on the Y-axis. The charts are linked so that the analyst can inspect frame times for a single frame across all callbacks for the selected API. An example of the Call Panel can be seen in Fig. 6.

To evaluate the rendered output of the pipeline execution, the tool shows the resulting image in the top-right corner. In Single Pipeline Mode, the tool also prints information about when the image was rendered (frame number) and the file's location and allows a full-screen display of the image.

4.3 Pipeline Comparison

We provide context to the performance runtime analysis by allowing the user to compare different variants of a single visualization pipeline (**T3**). This is useful for highlighting the impact changes in parameter settings have or how changes to the pipeline structure behave wrt. runtime performance and rendered output. Pipeline comparison is activated when the analyst selects two frames (Reference/Variant) for comparison in the Global

[9]dagre-layout product website: https://github.com/hikerpig/dagre-layout, last accessed: 2025-05-20.

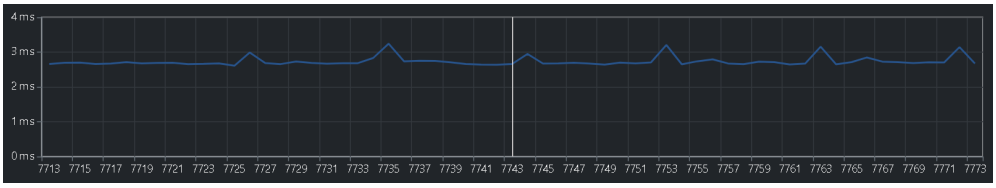

Figure 6: Detail view of the *Render* callback of the (amortized) parallel coordinate renderer exhibiting periodic spikes.

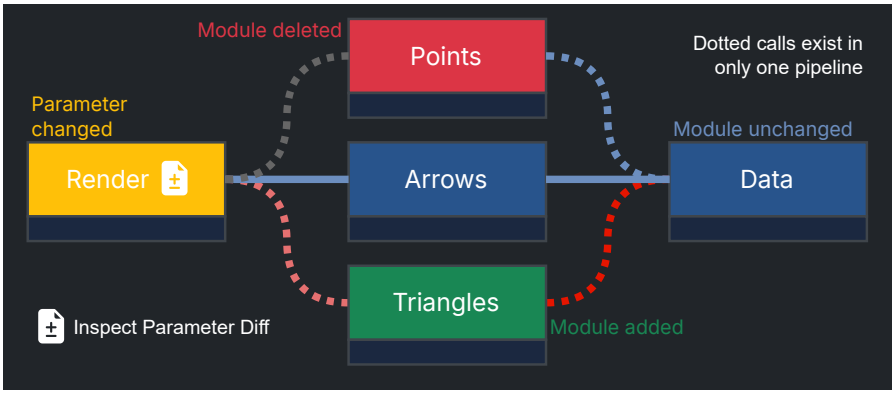

Figure 7: Color encoding used in diff mode: a new module was added (`Triangles`) and an old one deleted (`Points`). The `Render` module has at least one changed parameter. Calls existing in both variants are colored like in single view mode.

Timeline or the Event Log. In this Pipeline Diff Mode, we alter the UI components to show the difference between the frames. This mainly includes a textual diff of parameter changes on the module level, a visual diff of runtimes on the connection level in the Graph View and Call Panel, and finally, we use MS-SSIM inside the Screenshot Panel and Event Log to show the similarity of the rendered output.

The graph layout of the Graph View in Performance Diff Mode is calculated using the same layout algorithm as in Single Pipeline Mode. To position Modules only existing in one of the selected frames, we first construct a supergraph of all Modules and Calls from both frames and use this supergraph for layout calculation. Differentiation on what Module or Call belongs to which frame is done via color encoding.

In contrast to the Single Pipeline Mode, the coloring of the Graph View in the Pipeline Diff Mode is non-interactive. Each call is colored by the absolute diff value between the selected frames (see Fig. 7): blue means the Variant is cheaper than the Reference, white symbolizes no difference in performance, and red means the call is more expensive in the selected Variant. Calls in only one of the graphs are visualized with a dotted line, and the missing value is set to zero for diff calculation. This results in newly added calls appearing as red dotted lines and deleted calls as blue dotted lines. In Pipeline Diff Mode, the Modules of the graph are colored to indicate whether they were changed in the selected graph variant (see Fig. 7). Modules that only exist

in one of the selected frames are colored green when they only exist in the Variant (*Module was added*) or red when they only exist in the Reference (*Module was deleted*). Modules in both graphs are colored blue when identical and yellow if at least one parameter was changed. In the latter case, an icon is added that, when clicked, shows a textual diff of the set of changed parameters and their values (see Fig. 1). To color addition and deletion of nodes and parameter values, we use the established red/green color encoding from traditional diff tools like *diff* or *git*, but are aware that this choice is not ideal for users with a red–green vision deficiency.

The Call Panel adapts to the Pipeline Diff View by drawing the timings as separate lines superimposed on top of each other. Callback and API are still separate values.

In Pipeline Diff Mode, the Screenshot Panel shows both rendered images at once (see Fig. 1). The images are drawn on top of each other with a horizontal divider to show/hide the images. MS-SSIM values are displayed above the images for the quantitative analysis of image differences.

5 Design Process and Results

The design and functionality of the tool were developed in close collaboration with domain experts. This section reports the experts' feedback, how it shaped the development of the prototype, and concludes with a discussion of the diff views.

5.1 Expert-Driven Design of the Performance Analysis Tool

Our tool was designed through an iterative, expert-driven process that integrated domain-specific insights into its development. By involving MegaMol framework experts during the design phase, we ensured that feedback directly influenced the tool's capabilities and usability. MegaMol's acyclic graph structure (*Module Graph*) served as a foundation, enabling performance instrumentation of *Calls* and generating fine-grained timing visualizations for module callbacks (Sec. 3.1).

To guide the tool's design, we created four test scenarios: (I) Scatterplot Matrices and (II) Parallel Coordinate Plots (PCPs) with amortized rendering (Becher et al, 2023), (III) volume rendering, and (IV) a SciVis Contest 2019 pipeline (Schatz et al, 2019). Each scenario included annotated datasets to facilitate expert exploration. We interviewed three MegaMol experts (A, B, and C) during the design process. Each interview provided feedback that shaped subsequent tool iterations, emphasizing usability, clarity, and feature relevance. By involving experts during the design process, we continuously refined the tool's features and usability. Their insights ensured that the tool effectively addressed domain-specific performance analysis needs, such as identifying bottlenecks, debugging anomalies, and confirming expected behaviors. This iterative, expert-driven approach highlights the importance of early engagement with end users to inform design decisions and optimize analytical workflows.

Expert A, with extensive experience in amortized rendering, guided the design for analyzing scatterplot matrices and PCPs. Early sessions highlighted critical improvements to benchmarking consistency, leading us to replace wall-clock-based timing with frame-based timing for reproducibility. Suggestions for improving annotation clarity, frame numbering, and color coding were implemented to enhance usability. Feedback

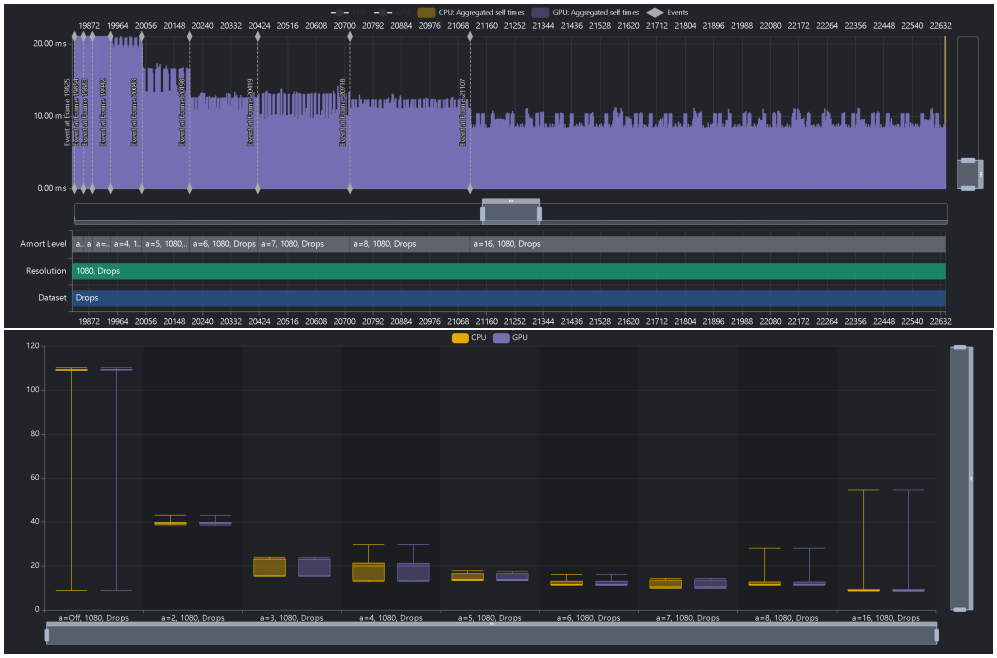

Figure 8: Amortization for the parallel coordinate renderer after selecting the 1080p region. Top (Global Timeline): diminishing returns for the acceleration become evident, especially given the increasing number of frames required for convergence. Bottom (Track Explorer): increasing frame cost jitter for the high amortization levels.

during subsequent sessions drove the addition of features to visualize performance trends, such as diminishing returns (see Fig. 8) at higher amortization levels. Expert A's analysis of periodic patterns and spikes in performance prompted design changes to capture previously untracked overhead, including timers for graph entry points. Recommendations to enhance visual cues for complex graphs, such as refined color mappings, further improved analytical workflows.

Expert B's feedback focused on usability and the overall user experience. He systematically assessed each component, leading to refinements in the Track Explorer and annotation editor to improve flexibility and ease of use. His preference for interactive aggregation tools inspired enhancements to filtering functionality and annotation-based aggregations. By testing volume rendering scenarios, he verified the plausibility of performance metrics and identified opportunities for improved interactivity, such as parameter adjustments during analysis. His input directly shaped user interface elements, such as timeline annotation linking, tooltip consistency, and parameter mappings. For the SciVis Contest data, he highlighted the tool's ability to emphasize temporal performance trends and debug anomalies effectively.

Expert C evaluated the PCP amortization scenario with a focus on correlating CPU/GPU costs. His feedback led to the inclusion of separate visualizations for overall frame timings and graph costs, enabling clearer identification of framework overhead.

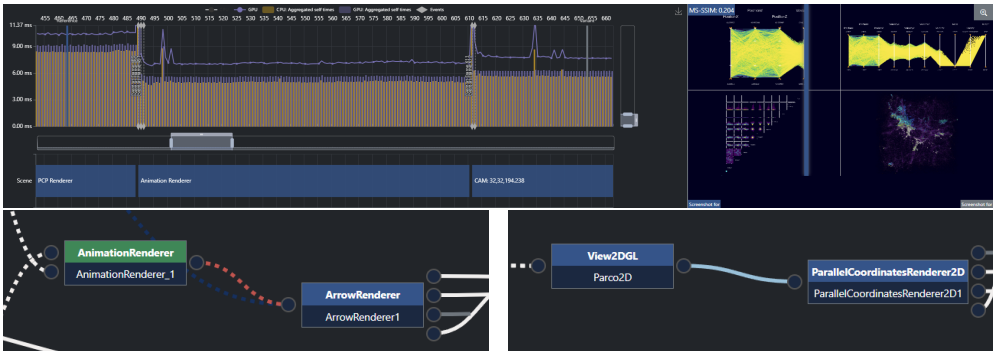

Figure 9: Example from the SciVis contest data set. The Timeline shows that rendering becomes cheaper, despite the new subwindow in the top left and the `ArrowRenderer` attached there. The call to the `ParallelCoordinateRenderer2D` shows that reducing its screenspace footprint makes it so much cheaper we can easily afford the overhead of another renderer.

He also suggested tighter linking between views, such as synchronized timeline and graph visualizations, which were incorporated into later iterations. Additionally, his input influenced design decisions for features like preview-augmented scroll bars and annotation-based navigation.

5.2 Diff Views

As indicated above, the experts were encouraged to explore our tool freely, and we did not direct the expert's attention or the exploration process itself. We contribute a discussion of the Pipeline Diff Mode, which was not analyzed in detail in the context of the expert feedback above.

First, we exemplify the interaction between the Event Log, the Graph View, and the Screenshot Panel in Pipeline Diff Mode. In the Event Log (see Fig. 3), which summarizes graph changes, we can gain insights regarding the performance profile (**T1**), and here we particularly notice a significant change in performance that is also associated with a small decrease in image quality according to MS-SSIM. The Graph View in Pipeline Diff Mode allows users to compare variants (**T3**). Here, we highlight the impact changes of a parameter setting have, specifically the change of the `volumeSamplingRate` on the timing of the `OSPRayRenderer`. The impact on quality can be investigated in the Screenshot Panel (see Fig. 1) by comparing the similarity of the rendered output. This is important as quantitative measures such as MS-SSIM can only indicate quality; they cannot assess the nature of produced artifacts/distortions. In the considered example, we can see only minor deviations across the whole image, which, combined with the substantial performance gain, suggests that the parameter change might be considered beneficial overall for typical application scenarios.

The second example shows two unexpected effects at once (see Fig. 9): In the reference frame, the output is split top/bottom, with the top spanned by a parallel coordinate renderer and the bottom split again among a scatterplot matrix and a

volume renderer augmented with arrow glyphs (**T2**). The variant frame additionally splits the top, reducing the output size of the parallel coordinates and adding a second, independent view of the glyphs. Here, the additional rendering invocation appears to make the overall rendering less expensive by about 30% (**T3**). From experience, we would also expect that parallel coordinates suffer significantly from overdraw, so compressing such a renderer's output can have adverse effects, as heavy overdraw serializes framebuffer writes. However, the Graph View in Pipeline Diff mode shows clearly that the parallel coordinates become significantly cheaper (**T3**), in fact, so much that we can easily afford to add the second invocation of the glyph renderer.

6 Discussion and Future Work

The proposed approach and its implementation still have certain limitations that can be addressed in future work.

Generalizability and Evaluation. In our ongoing work, we implemented the proposed approach as a prototype within the MegaMol framework, driven by continuous feedback from domain experts. To evaluate fine-grained performance data effectively, we recorded data in great detail, requiring only minimal changes to the considered framework (see Sec. 3.1). Given that visual editors for creating node-based pipelines are widely used in scientific visualization, we believe that our approach can be seamlessly applied to other frameworks as well. In future work, we aim to demonstrate its generalizability by applying it to additional well-established frameworks such as ParaView.[10] This step also broadens the possible user base, making it easier to perform a larger evaluation.

Live Monitoring and Visual Debugging. The proposed prototype is a post-mortem analysis tool. A meaningful extension of this approach would be to integrate it into a running visualization pipeline and enable live monitoring of fine-grained performance data. A step further would be to extend the live monitoring approach to live debugging. The user could define performance thresholds per module or connection. As soon as these are hit, the pipeline's execution would be suspended to allow inspection of the pipeline in its current state. These changes would, however, require a significantly tighter integration with the analyzed framework, abandoning the current approach of minimal changes to the hosting application.

Whole-frame Performance Given our observations in current graphics drivers (see Sec. 3.1), we see two choices for looking at point-to-point full-frame performance measurement: either one accepts costs *slipping* between frames, making the analysis of parameter influence extremely hard in the surrounding of the change itself, or one can force-flush the pipeline. The first option led to performance spikes in different frames for CPU and GPU in our tool, which was confusing to the experts. We verified this using a Tracy log of the same benchmark and could also observe extremely costly frames accompanied by unrealistically cheap ones. It breaks the association of queued GPU work with a particular frame, so performance can only be reliably inspected in terms of multi-frame aggregations, contrary to our scope. The second option is similar

[10] Using `VtkAlgorithm::ProcessRequest` as an entry point for recording fine-grained performance data, analogous to extending MegaMol's `Call` class (see Sec. 3.1).

to the stable power discussion when inspecting performance (Gralka et al, 2024): the results are easy to interpret and reliable but do not reflect a real-world scenario. This strategy also makes splitting CPU and GPU costs per frame unnecessary in the Global Timeline since they never differ significantly. However, they enable the analysis of rendering costs per frame, and in contrast to stable power, the performance numbers do not change significantly; only their temporal distribution does.

7 Conclusion

In this paper, we presented a post-mortem analysis tool for fine-grained performance data of visualization pipelines. It uses an annotated node-link diagram to simultaneously depict the topology of the underlying graph, runtime performance data, and render output. Our approach offers multiple views on states and allows comparisons across states, potentially involving topology changes. Feedback from scientific visualization experts during development demonstrated applicability and utility across different application examples. In future work, we aim to incorporate additional metrics, like memory usage or power consumption (Gralka et al, 2024).

Funding. This work was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)—Project-ID 251654672—TRR 161 and—Project-ID 327154368—SFB 1313, the Bundesministerium für Bildung und Forschung (BMBF, Federal Ministry of Education and Research)—Project-ID 16ME0608K—WindHPC, and by Mercator Research Center Ruhr—Project "Vergleichende Analyse dynamischer Netzwerkstrukturen im Zusammenspiel statistischer und visueller Methoden." Open Access funding enabled and organized by Projekt DEAL.

Data availability. Supplemental material is available at https://doi.org/10.18419/DARUS-4115.

References

Ahrens J, Geveci B, Law C (2005) ParaView: An end-user tool for large data visualization. In: Visualization Handbook. Elsevier, Oxford, UK, chap 36, p 717–732, https://doi.org/10.1016/B978-012387582-2/50038-1

Bavoil L, Callahan S, Crossno P, et al (2005) VisTrails: Enabling interactive multiple-view visualizations. In: Proceedings of IEEE Visualization. IEEE, New York, NY, USA, pp 135–142, https://doi.org/10.1109/visual.2005.1532788

Becher M, Heinemann M, Marmann T, et al (2023) Accelerated 2D visualization using adaptive resolution scaling and temporal reconstruction. J Vis 26:1155–1172. https://doi.org/10.1007/s12650-023-00925-3

Bentes C, Labronici BB, Drummond LMA, et al (2014) Towards an efficient parallel raycasting of unstructured volumetric data on distributed environments. Clust Comput 17:423–439. https://doi.org/10.1007/s10586-013-0244-0

Bethel EW, Howison M (2012) Multi-core and many-core shared-memory parallel raycasting volume rendering optimization and tuning. Int J High Perform Comput Appl 26(4):399–412. https://doi.org/10.1177/1094342012440466

Bezemer CP, Pouwelse J, Gregg B (2015) Understanding software performance regressions using differential flame graphs. In: Proceedings of the IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering. IEEE, New York, NY, USA, pp 535–539, https://doi.org/10.1109/saner.2015.7081872

Bruder V, Müller C, Frey S, et al (2020) On evaluating runtime performance of interactive visualizations. IEEE Trans Vis Comput Graphics 26(9):2848–2862. https://doi.org/10.1109/tvcg.2019.2898435

Childs H, Brugger E, Whitlock B, et al (2012) VisIt: An end-user tool for visualizing and analyzing very large data. In: High Performance Visualization—Enabling Extreme-Scale Scientific Insight. Chapman and Hall/CRC, New York, NY, USA, chap 16, p 357–372, https://doi.org/10.1201/b12985

Deakin T, Price J, Martineau M, et al (2016) GPU-STREAM v2.0: Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models. In: High Performance Computing. Springer, Cham, CH, pp 489–507, https://doi.org/10.1007/978-3-319-46079-6_34

Del Rosario E, Currier M, Isakov M, et al (2020) Gauge: An interactive data-driven visualization tool for HPC application I/O performance analysis. In: Proceedings of IEEE/ACM Fifth International Parallel Data Systems Workshop. IEEE, New York, NY, USA, pp 15–21, https://doi.org/10.1109/pdsw51947.2020.00008

Giménez A, Gamblin T, Jusufi I, et al (2018) MemAxes: Visualization and analytics for characterizing complex memory performance behaviors. IEEE Trans Vis Comput Graphics 24(7):2180–2193. https://doi.org/10.1109/tvcg.2017.2718532

Gleicher M, Albers D, Walker R, et al (2011) Visual comparison for information visualization. Inf Vis 10(4):289–309. https://doi.org/10.1177/1473871611416549

Gralka P, Müller C, Heinemann M, et al (2024) Power overwhelming: the one with the oscilloscopes. J Vis 27:1171–1193. https://doi.org/10.1007/s12650-024-01001-0

Gregg B (2016) The flame graph. Commun ACM 59(6):48–57. https://doi.org/10.1145/2909476

Grottel S, Krone M, Müller C, et al (2015) MegaMol—a prototyping framework for particle-based visualization. IEEE Trans Vis Comput Graphics 21(2):201–214. https://doi.org/10.1109/tvcg.2014.2350479

Hong S, Kim H (2009) An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In: Proceedings of the 36th Annual

International Symposium on Computer Architecture. ACM, New York, NY, USA, pp 152–163, https://doi.org/10.1145/1555754.1555775

Ikarashi Y, Ragan-Kelley J, Fukusato T, et al (2021) Guided optimization for image processing pipelines. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing. IEEE, New York, NY, USA, pp 1–5, https://doi.org/10.1109/vl/hcc51201.2021.9576341

Isaacs KE, Giménez A, Jusufi I, et al (2014) State of the art of performance visualization. In: EuroVis - STARs. The Eurographics Association, Eindhoven, Netherlands, pp 141–160, https://doi.org/10.2312/eurovisstar.20141177

Jönsson D, Steneteg P, Sundén E, et al (2020) Inviwo—a visualization system with usage abstraction levels. IEEE Trans Vis Comput Graphics 26(11):3241–3254. https://doi.org/10.1109/tvcg.2019.2920639

Li D, Mei H, Shen Y, et al (2018) ECharts: A declarative framework for rapid construction of web-based visualization. Vis Inform 2(2):136–146. https://doi.org/10.1016/j.visinf.2018.04.011

Moreland K (2013) A Survey of Visualization Pipelines. IEEE Trans Vis Comput Graphics 19(3):367–378. https://doi.org/10.1109/tvcg.2012.133

Petersen M, Lukasczyk J, Gueunet C, et al (2023) Extended visual programming for complex parallel pipelines in ParaView. In: Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization. The Eurographics Association, Eindhoven, Netherlands, pp 39–49, https://doi.org/10.2312/pgv.20231084

Pietron J, Funk L, Tichy M (2022) Improving the comprehension of evolving graphical models. In: Proceedings of the Working Conference on Software Visualization. IEEE, New York, NY, USA, pp 96–107, https://doi.org/10.1109/vissoft55257.2022.00018

Reina G (2023) Can image data facilitate reproducibility of graphics and visualizations? toward a trusted scientific practice. IEEE Comput Graph Appl 43(2):89–100. https://doi.org/10.1109/mcg.2023.3241819

Rizzi S, Hereld M, Insley J, et al (2014) Performance modeling of vl3 volume rendering on GPU-based clusters. In: Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization. The Eurographics Association, Eindhoven, Netherlands, pp 65–72, https://doi.org/10.2312/pgv.20141086

Sandoval Alcocer JP, Bergel A, Ducasse S, et al (2013) Performance evolution blueprint: Understanding the impact of software evolution on performance. In: Proceedings of the First IEEE Working Conference on Software Visualization. IEEE, New York, NY, USA, pp 1–9, https://doi.org/10.1109/vissoft.2013.6650523

Sandoval Alcocer JP, Beck F, Bergel A (2019) Performance evolution matrix: Visualizing performance variations along software versions. In: Proceedings of the Working Conference on Software Visualization. IEEE, New York, NY, USA, pp 1–11, https://doi.org/10.1109/vissoft.2019.00009

Schatz K, Müller C, Gralka P, et al (2019) Visual analysis of structure formation in cosmic evolution. In: Proceedings of the IEEE Scientific Visualization Conference. IEEE, New York, NY, USA, pp 33–41, https://doi.org/10.1109/scivis47405.2019.8968855

Tarner H, Frick V, Pinzger M, et al (2020) Visualizing Evolution and Performance Metrics on Method Level as Multivariate Data. In: Proceedings of the 13th Seminar Series on Advanced Techniques & Tools for Software Evolution. CEUR-WS.org

Tarner H, Bruder V, Frey S, et al (2022) Visually comparing rendering performance from multiple perspectives. In: Proceedings of Vision, Modeling, and Visualization. The Eurographics Association, Eindhoven, Netherlands, pp 115–125, https://doi.org/10.2312/vmv.20221211

Tkachev G, Frey S, Müller C, et al (2017) Prediction of distributed volume visualization performance to support render hardware acquisition. In: Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization. The Eurographics Association, Eindhoven, Netherlands, pp 11–20, https://doi.org/10.2312/pgv.20171089

Trümper J, Döllner J, Telea A (2013) Multiscale visual comparison of execution traces. In: Proceedings of the 21st International Conference on Program Comprehension. IEEE, New York, NY, USA, pp 53–62, https://doi.org/10.1109/ICPC.2013.6613833

Wang Z, Simoncelli E, Bovik A (2003) Multiscale structural similarity for image quality assessment. In: Proceedings of the Thirty-Seventh Asilomar Conference on Signals, Systems & Computers. IEEE, New York, NY, USA, pp 1398–1402, https://doi.org/10.1109/acssc.2003.1292216

Zhang Y, Owens JD (2011) A quantitative performance analysis model for GPU architectures. In: Proceedings of the IEEE 17th International Symposium on High Performance Computer Architecture. IEEE, New York, NY, USA, pp 382–393, https://doi.org/10.1109/hpca.2011.5749745

Zhuang X, Kim S, Serrano MI, et al (2008) PerfDiff: a framework for performance difference analysis in a virtual machine environment. In: Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization. ACM, New York, NY, USA, pp 4–13, https://doi.org/10.1145/1356058.1356060

Appendix: UI Overview

The following figures both show the same data set: parallel coordinate plots of the ”Balken” data set, rendered at 720p with an amortization factor of 5. The first figure shows the application in its single pipeline analysis mode of frame 3831. All available UI components are visible. The second figure shows the pipeline diff mode comparing frames 3831 and 4887. This screenshot contains only the UI components that change when switching to pipeline diff mode.