A Multi-Threading Architecture to Support Interactive Visual Exploration

Harald Piringer, Christian Tominski, Philipp Muigg, Wolfgang Berger

Abstract—During continuous user interaction, it is hard to provide rich visual feedback at interactive rates for datasets containing millions of entries. The contribution of this paper is a generic architecture that ensures responsiveness of the application even when dealing with large data and that is applicable to most types of information visualizations. Our architecture builds on the separation of the main application thread and the visualization thread, which can be cancelled early due to user interaction. In combination with a layer mechanism, our architecture facilitates generating previews incrementally to provide rich visual feedback quickly. To help avoiding common pitfalls of multi-threading, we discuss synchronization and communication in detail. We explicitly denote design choices to control trade-offs. A quantitative evaluation based on the system VISPLORE shows fast visual feedback during continuous interaction even for millions of entries. We describe instantiations of our architecture in additional tools.

Index Terms—Information visualization architecture, continuous interaction, multi-threading, layer, preview.

1 INTRODUCTION

Exploration of unknown data is an important task in the context of information visualization. Explorative tasks are different from presentation tasks in that they require frequent changes of the view on the data. This includes both, navigation between different data subsets and adjustment of parameters that control the visual mapping. Multiple coordinated views [2], dynamic queries [29], and direct manipulation [28] are key concepts to support visual exploration.

For smooth and efficient exploration, the ensemble of analytical, visual, and interaction methods has to generate results in a timely manner (within 50 – 100 ms [29, 30]). However, even moderately sized data can pose computational challenges. Computing a graph layout of a few hundered nodes or rendering a data set with a few thousand data records as a parallel coordinates plot may take a few seconds on a desktop computer. For *discrete interaction* (e.g., a single click on a button) delays or temporary loss of responsiveness might be acceptable, because interaction occurs at low frequency.

However, research in human-computer-interaction has long been emphasizing the significance of *continuous interaction* as a requirement of interactive systems to support native human behavior [12]. This is in particular true for information visualization, because examining multiple 'what if' scenarios is a key aspect of exploratory data analysis [30]. A scenario could, for example, refer to setting a model parameter to a certain value. For discrete interaction, the user has to explicitly specify scenarios of interest in a successive manner. This approach provides no information about properties between two scenarios and it requires much time to explore parameter ranges. Continuous interaction, on the other hand, allows the user to explore any range in any speed and reduces the risk of losing interesting scenarios. During continuous interaction, two important requirements are to keep the application responsive and to provide a sufficient amount of visual feedback. What 'sufficient visual feedback' refers to depends on the visualization and the purpose, but definitely involves showing a representation of the data.

Many approaches provide a fixed amount of feedback during a continuous user interaction. However, as the available computation time per update can hardly be predicted generically and may vary due to caching and scheduling effects, such approaches suffer from one of two drawbacks: 1) time is left unused and less visual feedback is provided than possible or 2) single updates take longer than the time between consecutive user events. In the second case, the application responsiveness may degrade severely if visualization generation happens in the same thread that is responsible for receiving events.

Therefore, some systems (e.g., IMPROVISE [37]) parallelize these tasks using multi-threading. Although multi-threading makes use of commonplace multi-core technology and is thus desirable, implementing multi-threaded programs is difficult [21, 23] and involves many potential pitfalls which have not sufficiently been addressed in the context of interactive visualization so far. Moreover, multi-threading by itself neither guarantees responsiveness due to potential blocks caused by thread synchronization, nor does it ensure rich visual feedback at interactive rates.

As contribution of this paper, we propose a generic multi-threaded visualization architecture that should help to avoid pitfalls related to multi-threading. It has been designed to meet the following goals:

- Guarantee responsiveness to the user at all times, i.e., avoid perceivable delays of the GUI
- Provide visual feedback as quickly as possible, i.e., keep the latency between interaction and visual feedback below 100 ms [29]
- Provide as much visual feedback as possible
- Scale to data sets with several millions of data items
- Scale with regard to multiple views
- Support most common types of visualizations
- Be applicable regardless of environment or language

Where goals are conflicting, we explicitly outline and discuss particular design choices. This architecture has been shaped based on experiences in implementing several visualization systems and tools, including SIMVIS (C++) [9], VISPLORE (C++) [26], CGV (Java) [34], and VISAXES (C#) [33].

In the next section, we take a look at related work. Section 3 describes our architecture, including details related to multi-threading. We present a quantitative evaluation based on the system VISPLORE in Section 4. We close with a discussion about design choices, further instantiations of our architecture, and ideas for future work in Section 5, and a conclusion in the last section.

2 RELATED WORK

We structure the discussion of related work into non-parallel techniques for achieving rapid visual response, concurrency and parallel programming in general, and multi-threading in interactive visualization in particular.

- *Harald Piringer is with the VRVis Research Center, Vienna, Austria.*
- *Christian Tominski is with the Institute for Computer Science, University of Rostock, Germany.*
- *Philipp Muigg is with the Vienna University of Technology and SimVis Gmbh, Vienna, Austria.*
- *Wolfgang Berger is with the VRVis Research Center, Vienna, Austria.*

Manuscript received 31 March 2009; accepted 27 July 2009; posted online 11 October 2009; mailed on 5 October 2009.
For information on obtaining reprints of this article, please send email to: tvcg@computer.org .

2.1 Non-Parallel Techniques for Rapid Visual Response

Without parallelizing event handling and the generation of visual results, constantly updating the entire visualization during interaction does not scale for large data as both the update frequency and the application responsiveness degrade significantly. Therefore, many systems provide only a fixed (usually minimalistic) amount of feedback during continuous interaction to ensure responsiveness. For example, the commercial system TABLEAU shows only an elastic rectangle during dynamic query operations, whereas the query evaluation is triggered only after releasing the mouse button.

Tanin et al. [31] describe optimizations to dynamic queries. They pre-compute the set of affected items for each pixel position of a slider. During slider movement, newly selected data items are displayed on top of the visualization, whereas removed items are drawn with the background color. Several visualization systems implement this approach (including SPOTFIRE and TREEMAP4). However, as noted by Fekete [13], the restriction to pixel precision is often not tolerable. Fekete also points out that query optimizations alone can not guarantee responsiveness, because the limiting factor is usually the rendering.

One way to speed up rendering is to use abstraction methods, which can operate in data space to reduce data size (e.g., sampling [11]) or in view space to accelerate the rendering (e.g., binning [25]). However, performing costly computations (e.g., clustering) for large data may cause a temporary loss of application responsiveness. Moreover, while abstraction methods can emphasize important information better as compared to indiscriminately showing all items, they necessarily imply a loss of details, which is not always acceptable.

2.2 Concurrency and Parallel Programming

Many real-time graphics applications (e.g., games) exploit the parallelism of modern graphics hardware to achieve interactivity when transforming geometric or volumetric data into images. In information visualization, Fekete and Plaisant [14] investigated methods based on hardware acceleration to interactively visualize a million data items in scatter plots and treemap visualizations. Besides rendering performance, non-standard visual attribute mappings support perception, and appropriate interaction methods are integrated. However, while definitely useful for particular visualization and interaction techniques, transferring all steps of the visualization pipeline to the GPU is not always possible.

Chan et al. [6] developed a client-server system for exploring massive time series. Interactivity is maintained by delegating data queries to eight multi-processor database servers and by applying caching and pre-fetching mechanisms. To guarantee smooth interaction, constraints are derived from the capabilities of the employed hardware and software, and limit the distance that a user is allowed to travel per exploration step. It remains unclear how far such large-scale architectures downscale to desktop PCs. Moreover, concurrency is not mentioned with regard to mapping and rendering steps. Chan et al. argue that the time required to map and render the data is negligible compared to query computation time, which contradicts the aforementioned claim by Fekete [13]. Obviously, the position of the bottleneck depends on the platform, the data size, and the type of both visualization and user interaction. Approaches that assume any of these factors as given can not solve the problem of guaranteeing responsiveness and maximizing feedback in general.

Parallelism and concurrency in a general sense are key topics of computer science and subject to ongoing research. There are numerous highly non-trivial related issues involving synchronization, communication, scheduling, consistency, deadlock prevention, data and task parallelism, performance, and scalability. In case of multi-threading, the advantages like utilizing commonplace multi-core architectures come at the expense of increased system complexity and higher implementation costs [21]. Automatic support (e.g., OpenMP or Intel Threading Building Blocks) provides help for exploiting parallelism for particular computations, but does not scale to parallelizing application-wide tasks like separating user input from generating visualizations. This problem has recently been termed as the *Multicore's Programmability Gap* [23].

Defining design patterns for particular problems has proven a good approach to cope with this complexity. Schmidt et al. [27] describe 17 patterns for concurrent and networked objects, covering event handling, synchronization, and concurrency. Similarly, Mattson et al. [22] define a pattern language for parallel programming, which is structured as dealing with finding concurrency, algorithm structure, supporting structures, and implementation mechanisms. More recently, Herlihy and Shavit [18] summarize the theory when programming for multiple processors and describe practical implementations for concurrent data structures. Many of the patterns and topics described in these books are applicable to systems for visual data analysis. Some patterns are partly related to the architecture as proposed in this paper (e.g., the *Active Object* design pattern [27]). However, the scope of most patterns is very general and none of these books addresses the requirements regarding responses to user interaction nor visualization aspects.

2.3 Multi-Threading in Interactive Visualization

Parallel algorithms and systems play an important role in scientific visualization. Besides approaches tailored towards dedicated graphics hardware or supercomputing environments, multi-threading is frequently used. However, many techniques focus on exploiting data parallelism by parallelizing the processing of data blocks [20]. On a task level, computations in SCIRUN [19] are multi-threaded and do not block the GUI, but are typically not designed for early cancellation due to new input. The system PARAVIEW [5] separates the VTK-based processing engine from the user interface by running both in different processes, and it relies on Tcl scripts for inter-process communication. Due to the design of PARAVIEW to scale to client/server environments and batch processing, it supports only two static levels-of-detail – one during interaction and one for still images –, and does not address early termination due to frequent user interaction. While there are also numerous approaches for progressive visualization, most of them focus on a dedicated visualization technique like volume rendering [4]. For this purpose, most approaches specifically tune the internal representation of the data to maximize performance.

In contrast, information visualization tools typically can not make as many assumptions about the data while offering the user many options to control the visualization pipeline. Unfortunately, little attention has been paid to multi-threading in information visualization literature so far. Heer et al. [16] note that an important issue in implementing the *Scheduler* pattern is to handle concurrency, but no information concerning communication and synchronization is given. Their framework PREFUSE [17] offers a scheduler mechanism to execute costly computations in a separate thread, e.g., to drive animations. The XMDVTOOL uses multi-threading only for asynchronous data pre-fetching [10]. Our review of open source visualization software showed that THE INFOVIS TOOLKIT [13], PROCESSING [15], and MONDRIAN [32] do not employ multi-threading at all.

The visualization system IMPROVISE [35, 37] focuses on a generic approach for coordinating multiple views. It uses shared objects (*Live Properties*), a visual abstraction language (*Coordinated Queries*), and other coordination patterns including containment patterns that are related to semantic layers which will be discussed in Section 3.2. IMPROVISE implements asynchronous displays based on retarding worker threads to allocate as much resources as necessary to the user interface thread (called *throttling* [36]). The authors also propose caching of visualization tiles and other enhancements to improve performance and interactivity during exploration. However, most aspects related to multi-threading are specific to Java. No details are provided on thread synchronization, early termination of updates, or on exploiting multi-threading for maximizing visual feedback. Moreover, the scalability to millions of data records remains unclear as "Interactive Performance" has been listed as future work [37].

To the best of our knowledge, there exists no generic architecture for inherently multi-threaded information visualization of large data, as many details about multi-threading have been left unpublished for information visualization systems. However, we believe that such an architecture could significantly facilitate the development of highly in-

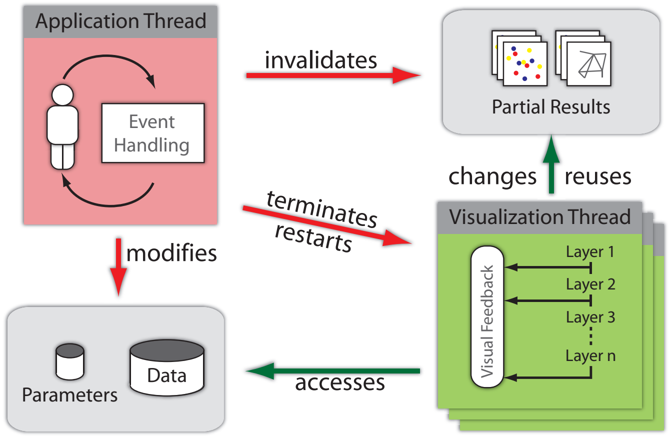

Fig. 1. Overview of our architecture. It shows involved threads and data, how threads access this data, and how the application thread controls the visualization threads.

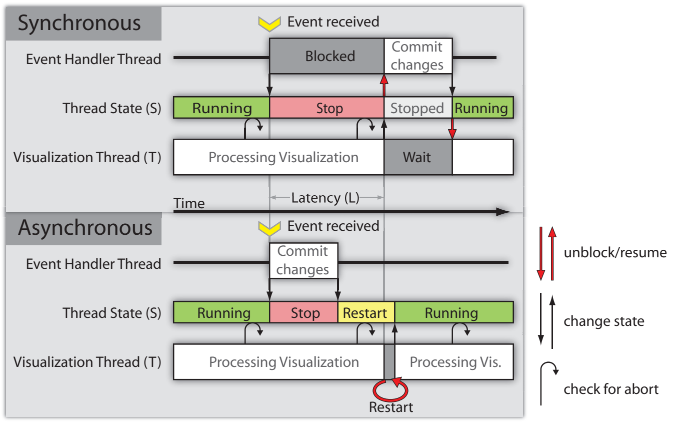

Fig. 2. Comparison of synchronous and asynchronous event handling. Threads communicate by changing the thread state S.

teractive information visualization tools, which combine responsiveness and rich visual feedback even during continuous user interactions. The importance and the current need for reusable architectures for visual data exploration are also documented by the fact that *Visual Analytics Infrastructures* has been established as a dedicated working package in the ongoing European project *VisMaster* [1].

3 Multi-Threading Visualization Architecture

We first provide an overview of our architecture before discussing its details in Sections 3.1 and 3.2. The architecture builds on the separation of the main application thread and visualization threads (see Fig. 1). The application thread is responsible for managing user requests in the event loop using event handlers. To keep this loop alive, event handlers are restricted to perform inexpensive tasks only, i.e., changing visualization parameters and triggering updates. Costly computations are delegated to visualization threads. In a multiple view environment, each view has its own visualization thread.

Especially during continuous interaction, updates in progress will frequently become irrelevant due to the arrival of new events. Therefore, the visualization thread checks repeatedly if it may proceed or should terminate early. For this purpose, we use a thread state object that serves as central point of communication. Depending on the semantics of the event, the execution of event handlers may be concurrent to the execution of the visualization thread (asynchronous), or mutually exclusive (synchronous).

The visualization is subdivided in image space into layers, and the visualization pipeline is processed separately for each layer. We will see later on that the term “layer” is used in a broader sense. Layers serve as partial visual results and can – in addition to partial results in data space – be reused across multiple executions of the visualization thread. Upon (early) thread termination, layers that have been validated so far can be displayed to provide as much visual feedback as possible and as early as possible.

3.1 Early Thread Termination

Our approach to support continuous interaction is to provide dynamic visual feedback by adapting the amount of detail to the available computation time. In general, this time is known only a posteriori, i.e., when it has elapsed due to receiving new input. Receiving this input, however, must be possible and not hindered by generating the feedback itself, which implies performing both tasks in parallel. It thus requires a multi-threaded architecture of each visualization.

According to the *Active Object* design pattern [27], invocation on an object should occur in the client’s thread of control, whereas execution should occur in a separate thread. In our context, the ’object’ is an interactive visualization, ’invocation’ refers to event handlers for processing change notifications which are typically triggered by user input, and ’execution’ means processing the visualization pipeline as widely accepted reference model [7] to generate visual results. Consequently, our visualization architecture maintains a single dedicated visualization thread T per view (maintaining multiple threads per view is discussed in section 5). Changes of parameters along the pipeline affect the final image and thus need to trigger a new execution of the pipeline. In this case, T must abort its current execution (if running) and eventually start processing the pipeline anew. We call this paradigm Early Thread Termination (ETT), as an execution may be aborted before T has finished the final image. During execution, T must repeatedly check for the permission to proceed. Besides necessary clean ups like freeing resources, it must abort once this permission is no longer granted.

The time between requested and actual thread termination incurs a certain latency L. Minimizing L is a central aspect of ETT and requires checking for abort at a high frequency. It is therefore an important requirement that checking is inexpensive, which is generally possible as explained below. When accessing data sequentially, performing a check after every few thousand entries is usually sufficient. In general, T should check at least 10 to 20 times per second to achieve interactive response rates [29, 30], but preferably even much more often. However, it can become impossible to guarantee a high frequency when calling to foreign APIs, which is admittedly a potential limitation of ETT. In order to lessen the practical impact of this problem in particular and to make the responsiveness of the application less dependent on L in general, an important observation is that changes (i.e., events) are critical with a different degree. Some changes require an ordered communication between the handler and T while others do not. We distinguish synchronous and asynchronous handling.

Synchronous event handling (see Fig. 2) enforces a mutually exclusive execution of the handler and T. This implies that handlers need to stop T, and must wait for this stop to occur before proceeding and eventually re-starting T. Synchronous event handling ensures that any subsequent execution of T is aware of the change.

Asynchronous event handling (see Fig. 2) also tells T to stop execution, but does not wait for this to occur. After committing the change, which potentially involves modifying parameters, the handler states that T needs to be restarted as soon as possible and returns. A current execution of T may notice the effects some time afterwards.

Basically, all changes could be handled synchronously. However, the performance of a synchronous handler – and thus the responsiveness of the application – depends directly on L, whereas asynchronous handlers are independent of L and typically do not block the event-handler thread. With regard to responsiveness, asynchronous handlers are therefore preferable and should be used for uncritical changes like modified parameter values. On the other hand, some events require synchronous handling, for example, when objects or data must no longer be accessed (e.g., due to deletion). In practice, visualizations will need both synchronous and asynchronous event handling.

It is a potential problem of ETT, that if an execution is constantly aborted before completing any result, no result will be delivered at all. In general, redundant computation across multiple executions of T should be avoided. It is therefore an important issue to:

1. identify *partial results* along the visualization pipeline, which can be cached and potentially reused across multiple executions,
2. maintain a *state of validity* $V[1..n]$, one for each partial result,
3. minimize the *impact of changes* by invalidating only those elements of V, where the respective result directly or indirectly depends on changed parameters.

Section 3.2 discusses this concept in detail in the context of interactive visualizations. For now, it is important that V is part of the communication between event handlers and T. Moreover, the communication involves the requested state of T, referred to as S. Fig. 2 illustrates, how S is accessed and modified by involved threads over time for both synchronous and asynchronous changes. As a fundamental idea of ETT, T repeatedly checks the state of S. STOP tells T to terminate execution. RESTART also tells T to terminate its current execution, but to immediately restart a new one. Fig. 2 also shows, how L directly affects the duration of synchronous handlers, which are blocked until T has reached the state STOPPED. In order to prevent deadlocks and livelocks, it is generally not recommendable for T to directly or indirectly trigger events itself.

As for all parallel systems, synchronization is important for ETT in order to avoid race conditions. The following points of synchronization can be identified:

- Between event handlers and T, as discussed above.
- Between different event handlers. If changes may occur in more than one client thread, event handlers themselves must be mutually exclusive in order to provide a predictable communication between each handler and T.
- Access to S between all handlers and T. As an important exception, if access to S is atomic (i.e., S is always accessed in one piece as is typically the case for basic data types), checking S for abort – i.e., read access – does not need synchronization, unless S is subsequently written in dependence of the result. This explains why checks for thread termination are usually cheap, meeting a requirement of ETT.
- Access to V between asynchronous handlers and T. For synchronous handlers, V is implicitly synchronized and thus does not require explicit synchronization.
- Access to *local* (i.e., view-specific) parameters along the visualization pipeline which are written by asynchronous handlers and read by T. However, synchronization of access is not sufficient to guarantee that the same state of parameters is used throughout one execution of T. To ensure this, T must maintain a local copy of those parameters which are potentially modified by asynchronous handlers. This is a major disadvantage of asynchronous handlers. Local parameters modified only by synchronous handlers are implicitly synchronized by the mutually exclusive execution. T does therefore not need to maintain a local copy of them. For this reason, modifications of memory-intensive local parameters (e.g., local derived data or a local selection state) typically require synchronous handling.
- Access to *global* (i.e., application-wide) parameters. Such parameters may change outside the execution of handlers of the particular visualization. In a multi-view environment, global parameters refer to the very information linking the views and thus include the data to be visualized itself. However, concurrent read access to global parameters by multiple views is necessary, because a synchronization of read-access to data would otherwise prevent concurrent processing of multiple visualizations, blocking all but one. It would thus eliminate responsiveness. Maintaining a local copy for each view is not practicable for large data. As a solution, changes to global parameters require two notifications: One synchronous notification preceding any modification, which forbids access, and one asynchronous notification permitting access when the modification is finished.

Finally, it is worth mentioning that although ETT is discussed in the context of visualizations in this paper, it is not limited to them. ETT can be applied to the design of any kind of objects that need to combine expensive computations with potentially frequent state changes due to interaction (e.g., ad-hoc queries or derived data columns).

3.2 Layered Visualization

As explained in Section 3.1, identifying and reusing partial results during the execution of the visualization thread is necessary to avoid redundant computation. This section discusses potential approaches to identify such partial results in the context of interactive visualizations, and how partial results help to display a dynamic amount of detail during continuous interaction.

A key idea is to subdivide the final image into separate passes through the visualization pipeline (referred to as "layer"), and to process one layer after the other. Each layer provides additional information and thus increases the amount of detail. It is important that the processing order may be chosen independently of the display order to prioritize important information for previews, as discussed below. In contrast to decomposing work in data space, which is often not possible in information visualization (e.g., computing graph layouts), layering is thus a concept for decomposing results in view space. For most visualizations, it is possible to identify one or more types of layers:

- *Semantic layers* are semantically different parts of the visualization. Typical examples include the background (e.g., an image, a map, a grid, etc.), all visible data items, those items selected by an ad-hoc query, and overlays providing detail-on-demand like labels or precise values [37]. It is reasonable to process semantic layers by decreasing relevance or increasing effort. For example, processing the layer of selected items ("focus") first will typically be less effort than considering all items ("context") and may already provide the most important information.
- *Incremental layers* can be identified in item-based visualizations (like scatter plots or parallel coordinates) by subdividing the data into disjunctive subsets and treating each subset as layer. Each incremental layer contains a sampled version of the data and the accumulation of all layers represents the entire dataset. A desirable feature is to ensure a sampling distribution that conserves important properties of the final image as soon as possible, i.e., in the layers being processed early. Desirable properties could be a size or a relative distribution similar to the final image. This aspect boils down to determining an index that specifies the order in which data entries are to be dealt with.
- *Level-of-detail (LoD) layers* provide visual representations of the same data with different complexity and rendering cost. In contrast to incremental layers, more detailed LoD layers may replace coarser layers, which are consequently not part of the final image. For example, a tree-map showing a hierarchy depth of four might be used instead of one showing only two hierarchy levels [3]. The design space for level-of-detail layers is large and includes abstraction in both view and data space. A *view space-based approach* could be to reduce the rendering quality for early layers, possibly in addition to displaying a sampled version of the data. Examples include disabling anti-aliasing and reducing geometric resolution. As example of a *data space-based approach*, lower levels of details might display features of the data like major trends, clusters, and outliers, or may use aggregation (e.g., bin maps) to reduce the rendering effort [25]. As a special case of LoD layers, *iterative layers* refer to visualizing intermediate results of an iterative algorithm, as for example the computation of a graph layout. In this case, each new layer (i.e., each iteration) typically replaces any previous iteration.

Once the final image could be completed, it is shown to the user. According to the ETT paradigm, the work is aborted whenever relevant parameters have changed. However, it is an important design choice, how visual feedback can be provided even in cases when the visualization thread could not complete.

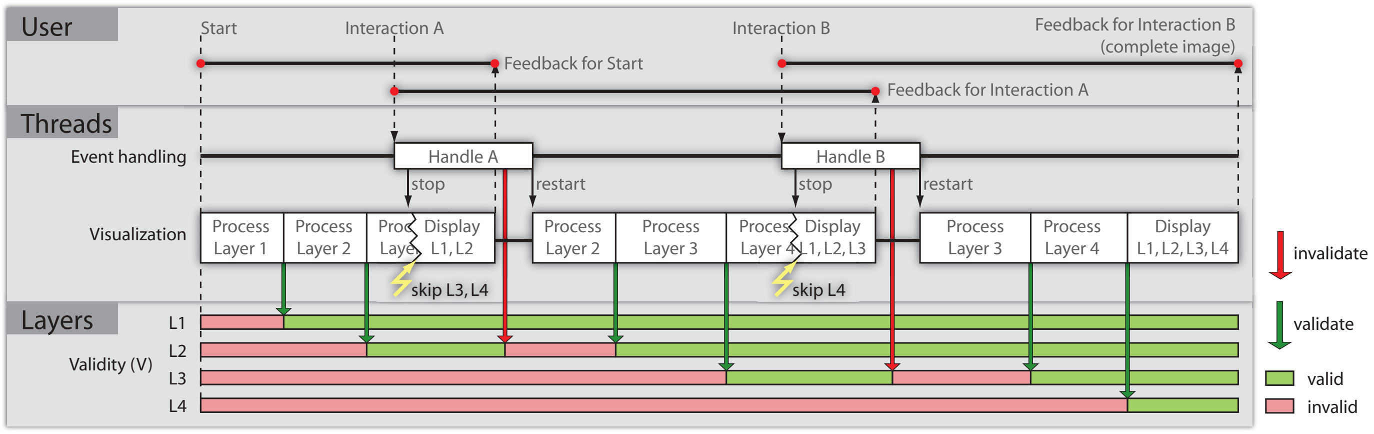

Fig. 3. Caching and early feedback of layers. Two user events (handled synchronously) interrupt the computation and invalidate layers. Visual feedback is provided on thread termination.

Design choice 1: immediate feedback vs. feedback on termination. *Immediate feedback* updates the display whenever a layer could be completed. As advantage, feedback is given early and is guaranteed to be up-to-date. As disadvantage, the composition of each image is exposed to the user and produces potentially disturbing flicker, which could be misinterpreted as data artifacts in extreme cases. In contrast, *feedback on termination* updates the display just before thread termination to show all valid layers, i.e., the highest amount of detail that could be dealt with in between two consecutive user interactions. The advantage is that only one image is generated per execution of the visualization thread, which reduces flicker significantly. As disadvantage, it might take longer until feedback is provided – in particular, if the execution is not aborted. The number and the type of layers and the effort for generating the final image are critical factors in the decision for one approach.

Design choice 2: type, number, and ordering of layers. In general, the number of visual layers increases with the complexity of a visualization. A single layer is most likely sufficient for basic bar charts, whereas a subdivision of parallel coordinates discriminating multiple selections and providing overlays could involve several semantic layers, which could in turn consist of LoD layers. Layers can thus be organized hierarchically. In this case, it is a design decision whether to prioritize level-of-details over semantic layers or vice versa. Apart from semantic dependencies, a processing order of layers may also be implied by internal dependencies between layers. For example, layers showing data items may depend on the layer showing the grid to determine the ranges of all displayed data dimensions.

An important decision for item-based visualizations is whether to provide fine-grained incremental visualization (i.e., a large number of incremental layers), or a fixed – typically small – number of LoD layers. The first case maximizes the average amount of provided detail (e.g., the number of shown items), yet it also increases the variation in the amount of details over time. This might create the impression of flicker even if a single image is shown per execution of the visualization thread. The second case is more stable with respect to the visual feedback, yet also reduces the possibility to adapt the amount of detail to the available computation time. This shows a trade-off between the amount of detail and stability.

Design choice 3: caching concepts. In order to avoid redundant computations, layers also represent reusable partial results. According to the model as proposed by Chi [7], different parameter adjustments affect different stages of the visualization pipeline. For example, changing a color could just require a redraw of already filtered, projected and possibly aggregated data. Performing just the rendering may thus be magnitudes faster than processing the entire visualization pipeline. We refer to this type of reuse as *caching results in data space*, which is related to lazy evaluation and demand-driven pipelines in visualization literature [20]. It is particularly useful for types of visualizations where computing internal representations of the data is relatively expensive as compared to the rendering itself, and where these representations consume a limited amount of memory. Examples include pivoted values of categorical data, aggregated representations as generated by binning continuous data, and the state of iterative algorithms (e.g., for graph-layout and clustering).

On the other hand, some changes affect the entire visualization pipeline, but only for a particular (semantic) layer. For example, ad-hoc queries may require a frequent re-processing of the selected data (“focus”), but may have no impact on the visualization of the entire data (“context”) or other visual elements like the grid. In this case, it is advantageous to *cache results in view space* for each layer independently. Fig. 3 illustrates caching and reuse of layers from the point of view of the user, the involved threads, and the layers as well as their validity. In this example, events are handled synchronously, feedback is provided on abort, and the validity is assumed on a per-layer basis, i.e., not taking partial results along the pipeline into account.

The additional complexity for implementing item-based visualizations using layers as compared to naive implementations can be summarized as:

- Invalidate affected layers instead of redrawing everything.
- Support multiple iterations through arbitrary subsets of the data instead of processing all items in one pass. In the case of multiple selections, for example, iterate through the data once for each selection (and once for all entries), instead of mapping the selection state of each entry to visual attributes like color or size within a single pass. As data records may appear in multiple layers, more significant layers must be shown on top of less significant ones. In particular, it is often desirable – though not required – that the visual representation of a selected item occludes its representation as non-selected item.
- Render layers to off-screen buffers and blend them together instead of drawing directly to screen. In practice, this is more easy to implement for 2D visualizations. In 3D, a composition in view space is generally harder to realize due to the additional depth-information necessary for correct occlusion handling.
- Check for thread termination regularly.

In our experience, these issues apply to all types of item-based visualization (scatter plots, parallel coordinates, time-series views, etc.). Sorting the items by their selection state or grouping them by identical rendering parameters is usually even necessary without explicit layering. The additional complexity imposed by *semantic layers* is thus usually (much) less than 20% in terms of lines of code. For *incremental layers*, the main effort lies in identifying an index for fair sampling (i.e., shuffling rows appropriately). Implementations of incremental layers typically require a single off-screen buffer where new visual output is added. For *LoD-layers*, the additional complexity may range from negligible (e.g., just disabling anti-aliasing) to considerable for cases that require the computation of features of the data like clusters.

4 Evaluation

This section evaluates the proposed architecture. The goal is to demonstrate its applicability and its possibilities to support visual exploration of large data. All tests have been conducted on consumer hardware: Intel Core 2 Quad CPU with four cores at 2.4 GHz, 4 GB of main memory, and an NVidia Geforce 8800 GTS graphics card. Windows XP Professional x64 Edition was used as operating system.

As test dataset, we used a multivariate CFD-simulation of a two-stroke engine. The data table consists of 14.589.282 rows and 50 columns (approx. 5.3 GB), which are mostly physical properties like temperature or pressure. One row in the data table represents one cell of the model geometry at one particular discrete time-step of the simulation. Previous analyses of the dataset have been conducted using the SimVis system [8], which also implements the proposed architecture (see Section 5). The focus of this evaluation is on performance issues with respect to maximizing visual feedback during continuous interaction for data of such non-trivial size.

We performed all tests in a system for visual exploration, termed Visplore. It provides more than 10 different visualizations, which are partly standard (e.g., 2D and 3D scatter plots, parallel coordinates, histograms, etc.) and partly specific to certain application tasks [26]. All views implement the proposed architecture to support continuous interaction and early visual feedback. Multiple views are linked by ad-hoc selections and derived data columns, whose evaluation also utilizes the ETT paradigm. Visplore is written in C++, it uses GTK+ as GUI library and OpenGL for rendering. The system has successfully been applied to analyze data of numerous application domains and is going to be released as part of the software suite of our company partner AVL List GmbH in 2009. However, the focus of this evaluation is to demonstrate the possibilities of our architecture, not to compare Visplore as such against any other system.

We discuss two examples of continuous interaction, which cover important cases: (1) The interaction concerns a single view, yet entails performing the mapping and the rendering stage of the visualization pipeline for the entire data. (2) The interaction concerns multiple views, but affects a single semantic layer. The implementations of the involved views also cover different options for the design choices 1 and 2, as explained below.

For the first example, we drag a slider to restrict the value range displayed on the X-axis of a 2D scatter plot. For the evaluation, we stored the interaction sequence as a macro (which takes 12 seconds) and replayed it with four different implementations to highlight trade-offs in the design space.

- *Case 1.1* A single-threaded implementation as example of a naive approach, i.e., each change entails a redraw of the entire visualization in the same thread as used for handling events.
- *Case 1.2* An ETT-based implementation providing immediate visual feedback for two static LoD-layers as example of a common case in many visualization systems. The first layer consists of a sampled subset of 32.768 data items without point smoothing and without transparency, the second layer is the entire dataset using point smoothing and transparency for visualizing density.
- *Case 1.3* An ETT-based implementation providing early visual feedback of fine-grained incremental layers as example of maximizing visual detail. The visualization pipeline is processed separately in blocks of 4096 rows and the visualization thread checks for abort after each block. The view provides feedback on termination, displaying all data handled so far, and on completion of the entire data set.
- *Case 1.4* An ETT-based implementation without preview visualization, i.e., visual results are only shown if the thread completed the entire visualization. This case has been chosen as example of evaluating the effect of multi-threading without layering.

We use several indicators. Responsiveness is quantified by the average number of user events that could be handled per second during the interaction. The frequency of visual feedback is given by the average and the minimal rate at which the visualization is updated per second. The amount of feedback and its variation – indicating flicker – is given by the minimal, average, and maximal percentage of shown data per update as well as the percentage of data that could at most be shown for 25% and for 75% of the frames (i.e., quantiles). Table 1 shows the results for the time between the start and the end of the interaction.

Table 1. Results for example 1: restricting a slider

	Case 1.1	Case 1.2	Case 1.3	Case 1.4
avg. # events handled / s	4.1	36.3	35.9	35.6
avg. # visual updates / s	4.1	13.2	35.9	0
min. # visual updates / s	3	9	18	0
min. items shown / update	100%	0.2%	0%	0%
q25 of items shown / update	100%	0.2%	3.5%	0%
avg. items shown / update	100%	0.2%	8.8%	0%
q75 of items shown / update	100%	0.2%	10.8%	0%
max. items shown / update	100%	0.2%	97.1%	0%

In case 1.1, feedback is given at a very slow rate. Even worse, the application is hardly responsive during the interaction. All other cases show that multi-threading ensures responsiveness of the application. Comparing case 1.2 to case 1.3 highlights the trade-off between minimizing flicker and maximizing visual feedback. In case 1.2, flicker does not occur at all because the visualization is updated only if the first LoD-layer could finish while the entire visualization (i.e., the second LoD-layer) could never complete. However, both the frequency and the average amount of visual feedback are significantly lower than in case 1.3, where even the minimal update rate of 18 is clearly faster than the desirable frequency of 10 (= 100 ms per update), and where a considerable percentage of the data (8.8%, i.e., 1.2 million items) is displayed on average – the best values are close to showing the entire dataset. On the other hand, the feedback sometimes drops to displaying the grid without data and flicker is generally high in case 1.3. Case 1.4 does not provide any feedback on the data, because at no point during the 12 seconds of interaction, the visualization thread is able to process the entire data in between two consecutive user events. This highlights the importance of early visual feedback. However, even case 1.4 is arguably superior to case 1.1, as it ensures responsiveness (i.e., the slider is updated continuously) and pausing the slider movement without releasing the mouse button would give the visualization thread the time to generate visual feedback. In practice, Visplore uses case 1.3.

For the second example, we drag an ad-hoc selection in a 2D scatter plot and highlight the selected data items in a linked parallel coordinates view showing 5 axes (see Fig. 4). Besides other types of queries, Visplore offers an instant ad-hoc query (referred to as *Focus*) that always selects all data items under the mouse cursor. The *Focus* is precomputed for all possible mouse-positions of a view, which reduces its evaluation to a look-up operation. However, each view needs to update frequently (i.e., on every mouse move) to reflect *Focus* changes. For the evaluation, we again stored an interaction sequence as a macro, this time a continuous mouse movement of 23 seconds, which causes frequent *Focus* updates. The macro has been tested against the following four implementations of parallel coordinates.

- *Case 2.1* A single-threaded implementation without caching any partial results as example of a naive approach where each change necessitates processing the entire visualization pipeline in the application thread
- *Case 2.2* An ETT-based implementation without caching any partial results. However, the *Focus* is processed first and is immediately displayed to provide visual feedback.
- *Case 2.3* An ETT-based implementation caching the image of the *Context* layer, i.e., the semantic layer displaying all data items, and reusing this images as long as the layer stays valid. The comparison of case 2.2 to case 2.3 is intended to emphasize the effect of caching.
- *Case 2.4* Same as case 2.3, but single-threaded to evaluate the effect of caching separately

The average number of events that could be handled per second during the interaction quantifies the responsiveness of the application. The minimum, maximum, and average response time indicate the latency

Table 2. Results for example 2: linked ad-hoc selection

	Case 2.1	Case 2.2	Case 2.3	Case 2.4
avg. events handled / sec.	0.2	20.1	13.5	8.8
min. response time (sec.)	6.7	0.03	0.03	0.03
avg. response time (sec.)	12.1	0.07	0.09	0.09
max. response time (sec.)	13.4	0.23	0.25	0.14
average data shown	100%	0.05%	100%	100%

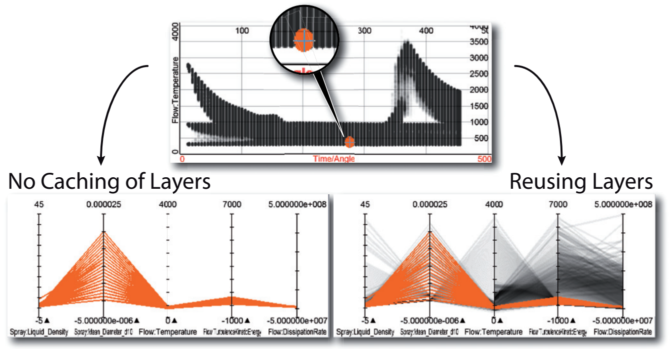

Fig. 4. Example 2: comparison of an ad-hoc selection of entries beneath the mouse cursor in a multiple view setup for two implementations of parallel coordinates. The response time is equally low in both cases, but the amount of detail is much higher when caching and reusing layers.

between changing the *Focus* and providing visual feedback. The average amount of shown data refers to the number of visualized items. In contrast to example 1 where continuous movement triggers updates constantly, the frequency of visual feedback is not a reasonable indicator in example 2, as moving the mouse cursor through empty space does not trigger updates. Table 2 shows the results.

For case 2.1, interaction is practically impossible as the system blocks for several seconds at each mouse move. For the cases 2.2 and 2.3, the system stays responsive and visual feedback is provided quickly. However, case 2.2 only displays the *Focus* most of the time, as illustrated by the left image in Fig. 4 while the entire visualization is only shown when the *Focus* is not updated for some time. Case 2.3 on the other hand always displays the entire visualization due to reusing the cached image of the *Context* as shown by the right image in Fig. 4. The results of case 2.4 are similar to those of case 2.3. This is not surprising considering that by re-using the image of the *Context*, not much work is left to be done. However, interactions invalidating the *Context* degrade the responsiveness as badly as shown for case 2.1.

Concluding, this evaluation demonstrates that the proposed architecture successfully preserves responsiveness of the application while providing visual feedback during continuous user interactions even for a dataset of 14.5 million items. It also shows that ETT, previews, and caching must work together to achieve this goal. Although not shown in this evaluation, our architecture also scales with respect to a large number of views. For informal evidence we refer to publications related to the systems implementing the architecture (for example [9]).

5 Discussion and Future Work

Three important yet contradicting objectives of our architecture are:

- to *minimize the latency* between interaction and visual feedback, which is equivalent to maximizing the frequency of updates
- to *maximize the amount of detail* shown upon ETT
- to *minimize the variation* of the amount of shown detail in order to provide a stable image.

The design choices 1 and 2 have been explicitly denoted, because they allow for trading off these objectives against each other: (1) Providing immediate feedback for each completed layer minimizes latency while it maximizes flicker – especially in the case of fine-grained layering. (2) Utilizing many layers allows for minimizing latency and maximizing detail, but the flicker is usually significant.

Another option for trading off latency against preview details concerns the handling of asynchronous events. As requests by asynchronous handlers are less critical than those issued by synchronous handlers, they can be ignored for some time. For example, it seems reasonable to finish and to display costly visual results which are almost complete when receiving a request for thread termination.

Design choice 3 is essential for adapting the architecture to a wide range of visualizations. Caching of results in view space is important where rendering is expensive, as for item-based visualizations. Caching results in data space is suitable in case of expensive computations yet potentially cheap rendering. For example, visualizing a large data warehouse may require aggregating billions of data rows for generating little output like a bar chart. Although less obvious than for costly rendering, early visual feedback is also possible in this case: the final results could repeatedly be estimated and displayed during the computation based on already considered data.

The most important limitation of our architecture is the need to frequently check for termination. As already mentioned, this may become impossible when passing control over to foreign APIs for a long time. This is particularly critical for synchronous changes as it compromises responsiveness much like a single-threaded architecture. Asynchronous changes preserve responsiveness, but the latency of visual feedback may still be disturbing.

It may seem reasonable to spawn a new visualization thread for each asynchronous event (synchronous changes must wait for thread termination anyway). However, we decided against this option, because our practice has shown that gains are small compared to a significant increase in complexity. While synchronization is complex for a single visualization thread, it becomes worse for multiple threads. Redundancy increases too, as each thread requires a copy of all local view parameters. Furthermore, it is not reasonably possible for graphics APIs that do not support concurrent access to the same rendering context (e.g., OpenGL). In such cases, maintaining a single thread per view avoids the significant overhead caused by context switches, which is incurred when using a common thread pool for multiple views.

The proposed architecture has been implemented in several systems besides VISPLORE (see Section 4). The SIMVIS visualization framework [9] is mainly used in the context of 3D or 4D simulation data. Multiple linked views are provided to the user, allowing for interactively selecting and viewing data in different attribute spaces. Multiple of these views implement ETT to maintain responsiveness even when visualizing hundreds of millions of data entries. Attribute views such as scatter plots or time series visualizations [24] all support asynchronous as well as synchronous thread termination and use cached background layers to provide feedback to the user during continuous interaction. The 3D visualization capabilities of SIMVIS also rely on concepts presented in this work to perform progressive rendering as well as level of detail rendering during continuous interaction using a multi-resolution approach. When dealing with very large data, a common approach is to access data in blocks [20] which are guaranteed to be in memory while the rest may be swapped to disk (known as out-of-core visualization). Both VISPLORE and SIMVIS perform active memory management, which shows that out-of-core visualization is compatible with our architecture. Switching blocks even provides a dedicated point to check for thread termination.

CGV is a system for interactive exploration of graphs [34]. It uses multiple linked views to show different aspects of clustered graphs. Early thread termination is used for instance in the graph splatting view. Because the performance of graph splatting depends not only on the number of data items, but also on pixel resolution, the view uses a level-of-detail layering and renders the splat progressively at increasing resolutions. Continuous interactions as for instance dragging a noise level slider or a threshold slider are guaranteed to stay responsive and feedback is provided quickly.

Axes-based visualizations map data values to positions relative to well-arranged visual axes. The TIME WHEEL [33], for example, allows among other interactions for continuous rotation of data axes around a central time axis. Interactive visual feedback is crucial in this case to help users maintain the mental map. Therefore, ETT and layering are applied for the TIME WHEEL. It is subdivided into semantic layers: axes layer, labels layer, preview layer, and data layer, which are drawn in this order. As a result, the basic shape of the visualization (i.e., the axes), labels, and a sampled version of the data are visualized early, while the entire data set is processed in the background.

We see multiple directions for future work. First, the design space potentially involves more than three design choices as discussed in this paper, and a systematic coverage would be very helpful. Second, the aspect of flickering needs more thorough research, including a user-evaluation about how much flickering is considered acceptable. New approaches could strive for minimizing flickering while still providing much visual feedback, e.g., by ignoring asynchronous changes for some time or by fading the images of consecutive updates. Third, it still remains a challenge to achieve rich visual feedback during continuous interaction in a distributed environment.

6 Conclusion

Continuous user interaction is important in information visualization to support smooth data exploration. A key concern is to preserve responsiveness and to provide rich visual feedback at the same time. Realizing this in practice is difficult, however, as it requires parallelism of application tasks which involves many non-trivial details.

We proposed a generic multi-threaded architecture to support continuous interaction and to help avoiding pitfalls. As illustrated by the evaluation, our architecture scales with respect to data size and the number of views. It is applicable to many types of visualizations regardless of a particular platform, programming language or graphics API, as instantiations in several visual analysis systems and tools show. GPU-based rendering is supported, but not required.

We identified and discussed three major design choices to allow others to adapt the architecture to particular visualization needs and to trade off latency against the amount of detail and the stability of visual feedback during continuous interaction. We also discussed in detail communication and synchronization aspects of our architecture as key issues of any multi-threaded program. We believe that our architecture will facilitate the development of highly interactive information visualization tools, and that it will help to promote rich visual feedback during continuous user interactions.

Acknowledgements

This work has been supported by the Austrian Funding Agency (FFG) within the scope of the projects VISCOMP, AutARG (No. 819352), and AVISOM (No. 818060) as well as by our industry partner AVL. Thanks go to M. Eduard Gröller, Heidrun Schumann, Helwig Hauser, and Krešimir Matković for constructive comments and discussion.

References

[1] EU Coordination Action VisMaster. `http://www.vismaster.eu`.

[2] M. Q. W. Baldonado, A. Woodruff, and A. Kuchinsky. Guidelines for Using Multiple Views in Information Visualization. In *Proc. of the Working Conf. on Advanced Visual Interfaces (AVI)*, pages 110–119. ACM, 2000.

[3] R. Blanch and E. Lecolinet. Browsing Zoomable Treemaps: Structure-Aware Multi-Scale Navigation Techniques. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1248–1253, 2007.

[4] S. Callahan, L. Bavoil, V. Pascucci, and C. Silva. Progressive Volume Rendering of Large Unstructured Grids. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1307–1314, 2006.

[5] A. Cedilnik, B. Geveci, K. Moreland, J. Ahrens, and J. Favre. Remote Large Data Visualization in the ParaView Framework. In *6th Eurographics Symp. on Parallel Graphics and Visualization*, pages 163–170, 2006.

[6] S. Chan, L. Xiao, J. Gerth, and P. Hanrahan. Maintaining Interactivity While Exploring Massive Time Series. In *Proc. of IEEE Symp. on Visual Analytics Science and Technology*. Computer Society, 2008.

[7] E. Chi. A Taxonomy of Visualization Techniques Using the Data State Reference Model. In *Proc. of the IEEE Symp. on Information Visualization*, pages 69–75. IEEE Computer Society, 2000.

[8] H. Doleisch. SIMVIS: Interactive Visual Analysis of Large and Time-Dependent 3D Simulation Data. In *Winter Simulation Conference*, pages 712–720. WSC, 2007.

[9] H. Doleisch, M. Gasser, and H. Hauser. Interactive Feature Specification for Focus+Context Visualization of Complex Simulation Data. In *VISSYM '03: Proc. of the Symp. on Visualization*, pages 239–248. Eurographics Association, 2003.

[10] P. Doshi, G. Rosario, E. Rundensteiner, and M. Ward. A Strategy Selection Framework for Adaptive Prefetching in Data Visualization. In *SSDBM '03: Proc. of the 15th Intl. Conf. on Scientific and Statistical Database Management*, pages 107–116. IEEE Computer Society, 2003.

[11] G. Ellis and A. Dix. Enabling Automatic Clutter Reduction in Parallel Coordinate Plots. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):717 – 724, 2006.

[12] G. Faconti and M. Massink. Continuous Interaction with Computers: Issues and Requirements. In *Vol. 3 of the proc. of HCI International 2001*, pages 301–305. Lawrence Erlbaum, 2001.

[13] J. Fekete. The InfoVis Toolkit. In *Proc. of the IEEE Symp. on Information Visualization '04*, pages 167–174. IEEE Computer Society, 2004.

[14] J. Fekete and C. Plaisant. Interactive Information Visualization of a Million Items. In *Proc. of the IEEE Symp. on Information Visualization '02*, pages 117 – 124. IEEE Computer Society, 2002.

[15] B. Fry. *Visualizing Data: Exploring and Explaining Data with the Processing Environment*. O'Reilly Media, Inc., 2008.

[16] J. Heer and M. Agrawala. Software Design Patterns for Information Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):853–860, 2006.

[17] J. Heer, S. Card, and J. Landay. Prefuse: a Toolkit for Interactive Information Visualization. In *CHI '05: Proc. of the SIGCHI Conf. on Human Factors in Computing Systems*, pages 421–430. ACM, 2005.

[18] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[19] C. Johnson, A. Parker, C. Hansen, G. Kindlmann, and Y. Livnat. Interactive Simulation and Visualization. *Computer*, 32(12):59–65, 1999.

[20] C. Law, W. Schroeder, K. Martin, and J. Temkin. A Multi-Threaded Streaming Pipeline Architecture for Large Structured Data Sets. In *Proc. of the IEEE Conf. on Visualization '99*, pages 225–232. IEEE Computer Society, 1999.

[21] E. A. Lee. The Problem with Threads. *IEEE Computer*, 39(5), 2006.

[22] T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, 2004.

[23] B. S. Michel and H. Zima. SC'08 Workshop: Bridging Multicore's Programmability Gap, 2008.

[24] P. Muigg, J. Kehrer, S. Oeltze, H. Piringer, H. Doleisch, B. Preim, and H. Hauser. A Four-level Focus+Context Approach to Interactive Visual Analysis of Temporal Features in Large Scientific Data. *Computer Graphics Forum*, 27(3):775–782, 2008.

[25] M. Novotný and H. Hauser. Outlier-Preserving Focus+Context Visualization in Parallel Coordinates. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):893–900, 2006.

[26] H. Piringer, W. Berger, and H. Hauser. Quantifying and Comparing Features in High-Dimensional Datasets. In *Proc. of the Intl. Conf. on Information Visualisation (IV08)*, pages 240–245. IEEE Computer Society, 2008.

[27] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects*. John Wiley & Sons, 2000.

[28] B. Shneiderman. Direct Manipulation: a Step beyond Programming Languages. *IEEE Computer*, 16(8):57–69, 1983.

[29] B. Shneiderman. Dynamic Queries for Visual Information Seeking. *IEEE Software*, 11(6):70–77, 1994.

[30] R. Spence. *Information Visualization: Design for Interaction (2nd Edition)*. Prentice-Hall, Inc., 2007.

[31] E. Tanin, R. Beigel, and B. Shneiderman. Incremental Data Structures and Algorithms for Dynamic Query Interfaces. *SIGMOD Rec.*, 25(4):21–24, 1996.

[32] M. Theus. Interactive Data Visualization using Mondrian. *Journal of Statistical Software*, 7(11):1–9, 11 2002.

[33] C. Tominski, J. Abello, and H. Schumann. Axes-Based Visualizations with Radial Layouts. In *Proc. of ACM Symp. on Applied Computing (SAC'04)*, pages 1242–1247. ACM Press, 2004.

[34] C. Tominski, J. Abello, and H. Schumann. CGV – An Interactive Graph Visualization System. *Computers & Graphics*, 2009. (to appear).

[35] C. Weaver. Building Highly-Coordinated Visualizations in Improvise. In *Proc. of the IEEE Symp. on Information Visualization*, pages 159–166. IEEE Computer Society, 2004.

[36] C. Weaver and M. Livny. Improving Visualization Interactivity in Java. In *Proc. of Visual Data Exploration and Analysis*. IS&T/SPIE, 2000.

[37] C. E. Weaver. *Improvise: A User Interface for Interactive Construction of Highly-Coordinated Visualizations*. PhD thesis, University of Wisconsin - Madison, 2006.