An Enhanced Visualization Process Model for Incremental Visualization

Hans-Jörg Schulz, Marco Angelini, Giuseppe Santucci, and Heidrun Schumann

Abstract—With today's technical possibilities, a stable visualization scenario can no longer be assumed as a matter of course, as underlying data and targeted display setup are much more in flux than in traditional scenarios. *Incremental visualization* approaches are a means to address this challenge, as they permit the user to interact with, steer, and change the visualization at intermediate time points and not just after it has been completed. In this paper, we put forward a model for incremental visualizations that is based on the established Data State Reference Model, but extends it in ways to also represent partitioned data and visualization operators to facilitate intermediate visualization updates. In combination, partitioned data and operators can be used independently and in combination to strike tailored compromises between output quality, shown data quantity, and responsiveness—i.e., frame rates. We showcase the new expressive power of this model by discussing the opportunities and challenges of incremental visualization in general and its usage in a real world scenario in particular.

Index Terms—Visualization pipeline, data state reference model, progressive visualization, proactive visualization.

1 Introduction

In its most common form, the visualization process follows a pipeline of operators. These operators can be adjusted—e.g., exchanged or reparametrized—before or after being applied to the data, but not while they are used to generate the visualization. This monolithic form of visualization works well for traditional visualization scenarios, in which input and output constraints to the visualization remain stable. For example, it is suited in cases where the input data is static and available in full and where the display setup for which the visualization is generated stays fixed.

Yet with today's technical possibilities, such a stable scenario can no longer be assumed as a matter of course, as these constraints are much more in flux and harder to pinpoint for the duration of the visualization process. To stick with the examples of input data and display setup, the input may be streaming data of continuously changing quantity and quality, and the display setup can be flexibly changed during the process as displays become ubiquitous. For such unstable scenarios, monolithic visualization fails to address their specific needs. In the case of an infinite data stream, adjusting the visualization pipeline cannot be done after the visualization is completed, as it never will be. This calls for an *incremental visualization process* [1] instead, which permits the user to change and interact with a visualization at intermediate time points and not just after it has been completed.

While incremental visualization solutions are increasingly being developed and used these days, the same cannot be said for the modeling of the incremental visualization process. Existing visualization models, such as the *data state reference model (DSRM)* [2] or the *visualization pipeline* [3], [4] date back to the 1990's and are still in use as formal underpinnings for dataflow-oriented visualization software, such as AVS [5] and VTK [6]. For incremental visualization, such a model is still needed in order to capture its particular opportunities and challenges.

With this paper, which builds upon ideas first presented as a poster at IEEE VIS '14 [7], we address this need by taking a first step towards a visualization model that is able to capture the intricacies of incremental visualization. We do so specifically by extending the DSRM, so that we are able to model traditional monolithic and more recent incremental visualization processes alike. This is achieved by replacing selected monolithic visualization operators with subdivided operators yielding intermediate results and monolithic datasets with streams of data chunks, or both.

We then use the new expressive power of this model to highlight the opportunities and challenges particular to incremental visualization. One of these opportunities is, for example, that through an appropriate user interface, incremental visualization allows a user to view and interact with early "draft visualizations" that may still miss parts of the data (e.g., showing just a few first samples) or that may still miss visualization layers (e.g., showing just data points but no labels yet) or both. This does not only support a quick readjustment of inadequate visualizations without having to wait for the visualization to

- *H.-J. Schulz is with the Fraunhofer IGD Rostock, Rostock, Germany. E-mail: hans-joerg.schulz@igd-r.fraunhofer.de.*
- *M. Angelini is with the Sapienza University of Rome, Italy. E-mail: angelini@dis.uniroma1.it.*
- *G. Santucci is with the Sapienza University of Rome, Italy. E-mail: santucci@dis.uniroma1.it.*
- *H. Schumann is with the University of Rostock, Germany. E-mail: schumann@informatik.uni-rostock.de.*

be rendered in full, but the intermediate visualization also facilitates a better understanding of how the final visual representation came about.

Among the challenges of incremental visualization is that it is more complex to set up (e.g., deciding when, how, and from where to gather the intermediate visualization results) and to steer (e.g., deciding when has the user seen "enough" to draw a conclusion from the visualization). Hence appropriate user interfaces are needed that support these tasks. We demonstrate such a UI in the context of our application scenario. This scenario entails a data analysis application in which the input data is chunked and streamed to provide the user with a responsive interactive visualization system for *fluid interaction* [8].

The remainder of this paper is structured as follows: Section 2 will further detail and define the notion of incremental visualization, relate it to existing approaches, and outline the challenges it entails. In Section 3, we propose our model for describing incremental visualization scenarios in a more general and abstract way. Considerations for using this model are reviewed in Section 4 and then exemplified for the concrete use case in Section 5. Finally, Section 6 concludes this paper by summarizing the key points of the proposed model for incremental visualization.

2 Incremental Visualization

This section sets out to further discuss and assess the concept of incremental visualization. This is done from three angles. First, we define the principal ways in which visualizations can be turned into incremental visualizations—i.e., by subdividing the visualization operators and/or chunking the data. Then we relate these principal ways to a number of existing visualization scenarios from the literature considering their differences and commonalities. Finally, we highlight some of the challenges arising from the notion of incremental visualization.

2.1 Background and Definitions

For the remainder of this paper, we utilize the *state/operator model* of the visualization process [2], [9]. We do so without loss of generality, as it has been shown by Chi [10] that the state/operator model has the same expressiveness as dataflow-oriented models, commonly known as *visualization pipelines* [3], [11].

Data chunks: We define that a dataset of size n (i.e., containing n *data items*) is given in i *data chunks* with $1 \leq i \leq n$. $i = 1$ denotes the traditional visualization of the whole dataset in one pass. $i = n$ denotes a stream of individual data items.

Processing steps of a subdivided operator: We define that a visualization operator consisting of m *processing operations* is given in j *processing steps* with $1 \leq j \leq m$. $j = 1$ denotes the traditional way of generating the visualization in one monolithic procedure and $j = m$ steps denote its most fine-grained modularization into individual processing operations.

Increasing both, the granularity of the data chunks and the granularity of the processing steps, was suggested early on in visualization research [12]. Since then it has been used, for example, to facilitate parallel architectures through chunking the data and subdividing the visualization process in order to run them concurrently [13]. For many visualization scenarios, the chunking of data and the subdivision of visualization operators are independent of each other.

Incremental visualization: Based on this model of chunked data and subdivided operators, we define *incremental visualization* as an iteratively refining visualization process that adds or updates visualization details in a stepwise effort to complete a dataset's visual representation. In this context, *completeness* is understood with respect to the dataset and to its processing—i.e., the visualization is complete when all i data chunks have been processed by all j processing steps. Once completeness is achieved, the visualization process terminates. When completeness cannot be achieved (e.g., infinite streaming data or non-converging computations), the process can also terminate when it is guaranteed that none of the remaining unprocessed data and none of the remaining iterative processing steps would visibly change the outcome any further—i.e., the visualization is sufficiently stable. If stability cannot be achieved, incremental visualization becomes an infinite loop that continuously modifies the visual representation.

Such an iteratively refining visualization provides the user with a preliminary visualization result after each iteration as intermediary stages. This is not a mere one-way-street, as it gives the user a visual output with which to interact while the final output is still being generated. By doing so, the user effectively sends early feedback to the visualization process that can adjust input data and/or processing parameters at intermediary stages. These adjustments can be the specification of regions of interest within the (streaming) data space or the reparametrization of processing steps. This general notion of incremental visualization does not only relate to a number of existing visualization approaches, but the interaction throughout the visualization process—i.e., the steering of this very process—leads to novel challenges.

2.2 Related Work

Going through the literature, it appears that a number of seemingly disjoint visualization approaches can be understood as incremental visualization. These approaches can be fundamental visualization architectures or specific visualization solutions, as well. The following gives a few examples for both and discusses thereafter, in which ways incremental visualization differs from them.

2.2.1 Fundamental Visualization Architectures

Out-of-core visualization [14], [15] is used when the size of the dataset to be visualized exceeds the available memory space. In order to nevertheless generate a visualization, the dataset is partitioned into smaller chunks, which are then sequentially passed through the visualization process to reduce its memory footprint. This idea of processing chunked data instead of the whole dataset can be traced back as far as 1993 [16]. It thus works in the same spirit as the data chunking in our concept of incremental visualization.

Progressive visualization aims to continuously show visual updates even through long-running data transmission [17], data loading [18], and rendering procedures [19] for whose completion one would otherwise have to wait. It aims to produce and show partial, yet meaningful results from running processes. This architectural paradigm makes use of data chunking—e.g., chunking and reordering the data to be transmitted, or sampling the data to be rendered into chunks of increasing size. A similar notion is *progressive visual analytics*, which provides partial results from long-running analysis operations [20]. The partial results are generated by a recursive algorithm, whose recursion steps map to a subdivision of the process. In both cases, the chunking of data and the subdivision of the process, progressive visualization can be mapped to incremental visualization.

Parallel visualization [21], [22], [23] is used to distribute the visualization process across multiple processing units and to run it concurrently. Parallelism can be achieved (among other options) by chunking and distributing the data (*data parallelism*) and/or by subdividing and distributing the visualization process (*pipeline parallelism*) [13], [23]. Additional caching mechanisms [24] and scheduling mechanisms [25] can further enhance the parallel execution by allowing for reusing stored results from previous computations and omitting unnecessary computations, respectively. It is thus closely related to our concept of incremental visualization, which additionally supports ongoing interaction throughout the visualization process.

Computational steering is the "interactive control over a computational process during execution" [26]. This control is usually facilitated by a visualization of the progress or an intermediary outcome of the computation with which the user can interact. The information about the running process is extracted and displayed at so called *checkpoints*, *breakpoints*, or *sync points* during the computation. These points along the running process can be either predefined or interactively set by the user. As they clearly subdivide the computational process, this corresponds to forming processing steps in an incremental visualization.

2.2.2 Specific Visualization Solutions

Streaming data visualization [27], [28] is used mainly for transient dynamic datasets that are continuously generated and possibly infinite. Such data is generated, for example, in the form of news feeds or sensor data. Apart from that, it is also employed for transmitting large but finite datasets across slow network connections. In both cases, the data comes as a sequence of smaller chunks and is processed as such. Hence, this scenario matches the idea of partitioned data in our incremental visualization concept.

Layered visualization [29] is a concept borrowed from the field of geographic information systems (GIS) where cartographic representations are assembled from independent layers showing different aspects of a map—rivers, streets, buildings, vegetation, etc. This can be applied to visualization as well, where we can likewise define independent layers, such as the actual *data layer* showing the data items, a *reference layer* showing coordinate axes and color legend, an *annotation layer* showing labels, etc. Each of these layers can be generated individually and shown independently, building up the visualization layer by layer and thus effectively subdividing the visualization process in the sense of incremental visualization.

Online dynamic graph drawing [30], [31] is applied to time-varying network-structured data, which is not yet known in full at the time of its layout. It stands in contrast to the typical dynamic graph drawing approaches for which information about the entirety of the dataset is already known a priori—for example, the complete nodeset and edgeset over all time points (*supergraph*, *union graph*). As the data is dealt with in an incremental fashion, it maps directly to the stepwise procedure that forms the essence of our concept of incremental visualization.

2.2.3 Putting Incremental Visualization in Context

The approaches discussed above have in common that they are *reactive* or problem-driven. By this, we mean that they are either motivated by the demands of the input data—known to be too large for the memory (cf. out-of-core visualization) or for the network connection (cf. streaming data), or even of unknown size (cf. online dynamic graph drawing). Or they are motivated by unacceptably long runtimes of high-end visualization techniques leading to sophisticated interruption techniques (cf. computational steering), to concurrency mechanisms (cf. parallel visualization, layered visualization), or to subdivision strategies (cf. progressive visualization).

In contrast to that, we suggest a *proactive* perspective on incremental visualization that is output-driven—i.e., driven by its potential for the user rather than by the requirements of data size or processing times. This means that we propose incremental visualization as a highly flexible process that can and often should be applied even when it would not be necessary from a data or runtime point of view. Anyone who ever watched a force-based network layout unfold step by step knows that observing

the construction of a visualization "live" is a powerful visual feedback that instills confidence in the visualization result. Recent research shows that such iterative refinement can speed up data exploration by allowing for interaction with unfinished visualization stages [32], [33], unlike any progress bar.

Yet, its benefits go well beyond the mere speed-up of producing the first preview and delivering a nice animation of the visualization process. For example, the chunking and reordering of the data according to the user's wishes also allows for a reprioritization of what shall be shown first. This mechanism can be utilized for a "tour through the data", as it was already envisioned by Rosenbaum et al. [17] for progressive visualization. Such a tour could be used for a configurable, stepwise build-up of information that would convey a sequential order in the data and automatically draw attention to the data items drawn first. It is output-driven in the sense, as it computes a chunking and ordering of the dataset that reflects what a user is interested in seeing first (analysis scenario) or showing first (presentation scenario).

The incremental visualization process model proposed in this paper, thus not only aims to provide a generalized conceptual foundation for the diverse approaches listed above by incorporating chunking of data and subdivision of operators alike. It also aims to capture such proactive approaches by making process information like different metrics explicit, which in turn helps the user to make informed changes to the process and its parameters. It is thus not the focus of our incremental visualization process model or this paper to discuss or to introduce particular approaches for subdivision on a technical level, as multiple such approaches already exist—for example, *incremental querying* [34], [35] or *database sampling* [36], [37]. Instead, we abstract from these issues that present research fields in their own right by modeling the generic parameters connected to such subdivision mechanisms and allowing to instantiate them with any particular mechanism. The challenges resulting from our generic view on incremental visualization are outlined in the following section.

2.3 Incremental Visualization Challenges

For concrete scenarios of incremental visualization, like the ones given in the previous section, design decisions are often already narrowed down by the setup of the scenario or by commonly known experiences with such scenarios. Yet when considering incremental visualization as a generic concept, which permits a user to facilitate any of the possible subdivision scenarios, one needs to address the challenges of authoring and using such a flexible approach.

When **authoring incremental visualizations**, the author has to decide on the necessary chunking of data and subdivision of operators. The simplest scenario is to have no chunking for both ($i = 1, j = 1$) and thus a traditional monolithic visualization. Then there are the two cases of either chunking the data ($i > 1, j = 1$) or subdividing the operators ($i = 1, j > 1$). We call the first case the *quality-first strategy*, as it rather trades the amount of the shown data (i.e., only a few chunks instead of the full dataset) than the image quality of the result, as the visualization process is still carried out in full. Likewise, we call the second case the *quantity-first strategy*, as it rather makes concessions to the quality of the output visualization (i.e., a quick and dirty preview instead of a full-fledged high-res result) than to show anything short of the full dataset. Yet the most interesting case is certainly the combination of these two strategies to produce a custom visualization compromise. While this allows for negotiating and tailoring the design decisions for every visualization problem anew, it also requires extra consideration for which currently no precedence exists. A more detailed discussion of the considerations that go along with these subdivision strategies will be given in Section 4.

When **using incremental visualizations**, the user has to be aware that he or she is seeing and interacting with an unfinished visualization. It is thus important for the user to be able to judge the reliability of the shown intermediate results—data-wise (i.e., how much of the full dataset is already shown) and process-wise (e.g., how many layers of the visualization are already shown). If the user interacts with an incremental visualization too early, the interaction has a high likelihood of being inappropriate or not targeted well enough. Yet if it is done too late, the user wasted valuable time waiting for further changes to the visualization that did not occur. Since this cannot be determined simply from looking at a current visualization state, the incremental visualization must provide meaningful and sufficient information about the progress of the process.

These principal challenges are rarely described, let alone addressed in the related work. The reason could be, that the different scenarios subsumed by our incremental visualization concept have so far been considered in isolation. As a framework and joint nomenclature for discussing these principal challenges and possible solution strategies for them, we introduce an extension of the DSRM to incremental visualization.

3 An Incremental Visualization Process Model

While the definition of incremental visualization in Section 2.1 gives an understanding of what we mean by this term, this section models incremental visualization in a manner that allows capturing its particularities that go beyond traditional monolithic visualization. Section 3.1 thus starts from an established process model for this traditional case and enhance it where necessary for using it in the incremental case.

We then discuss further considerations that govern the specifics of data chunking in Section 3.2 and the particularities of operator subdivision in Section 3.3, before bringing them together in Section 3.4.

3.1 Extending the Data State Reference Model for Incremental Visualization

A visualization pipeline models the visualization process by encapsulating algorithmic steps in operators that are usually depicted as rectangular nodes. The passing of data between these algorithmic steps is modeled through transitions that connect the nodes in the form of directed edges. Each operator is responsible for transforming data from one data state into another. These states are commonly termed: *incoming (raw) data*, *derived data*, *geometry data*, and *image data*. The transformatory processes between them are usually called *filtering* (incoming data ⇒ derived data), *mapping* (derived data ⇒ geometry data), and *rendering* (geometry data ⇒ image data). This base model is extended by the DSRM to also permit operators that do not only transform between data states, but also operate within a data state.

We extend the DSRM, so that its original form becomes a subset of the extended DSRM. It is thus able to model traditional monolithic visualization and more recent incremental visualization approaches. Hence, our model consists also of operators and transitions between them for passing the data. In addition, we explicitly model the sources (i.e., datasets) and the sinks (i.e., resulting views) in our model (Figure 1a), as it is not only common nowadays that multiple data sources are visualized in multiple views, but also that these data sources are heterogeneous—e.g., some of them are available in full, whereas others are available as a stream of data chunks. With this addition to the model, these sources can be clearly represented and distinguished. Operators and transitions are further enhanced to model their subdivision.

Enhanced operators model the subdivision of the visualization process. They are represented through a number of white to black marks at the bottom of an operator node. Each of these marks symbolizes an (intermediate) result: the black mark represents the complete result, the gray marks represent partial results, and the white mark represents the possibility to simply pass the unchanged input as an output, effectively making the operator optional. A traditional monolithic operator is thus depicted as a node with a black mark—or if it is optional, a white and a black mark. Whereas an incremental visualization operator is shown as a node with additional gray marks. The different kinds of nodes according to how they are marked are listed in Figure 1b. The number of the gray marks can either denote the number of partial results produced, or it can simply be used in a metaphorical way to denote that partial results are available independent of their number.

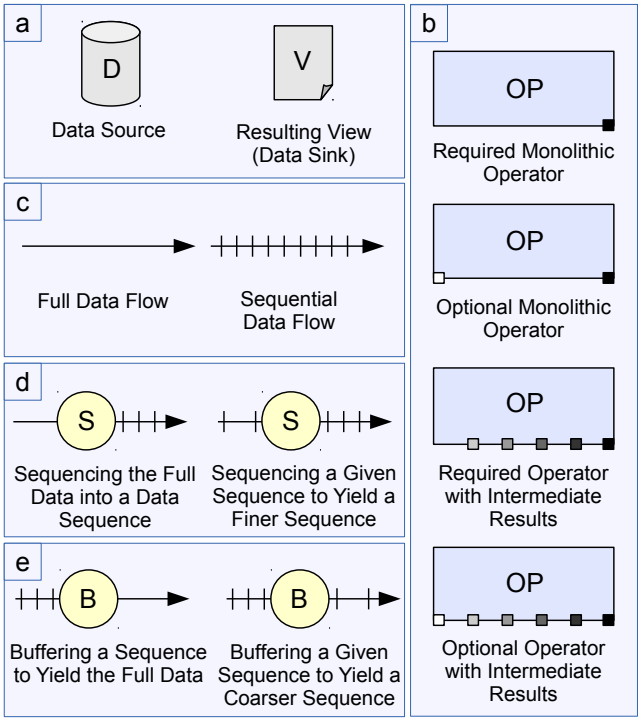

Fig. 1. Our extended model of the visualization process adds explicit data sources and data views (a), operators that make intermediate results available (b), sequential data flow (c), as well as sequencing and buffering mechanisms (d+e).

Executing such a subdivided operator simply means that it returns a series of results—first a result identical to the input if the operator is optional, then partial results of increasing quality, until finally the complete result is produced. Each produced result (identity, partial result, complete result) must be a valid input to all operators that follow immediately after. This entails, for example, that a transformatory operator that processes data from one state and outputs data from another state can never be optional. It must process at least part of the data to serve a valid input to the next operator that expects data from a different state—e.g., geometry data instead of derived data. How many partial results *can* be produced is operator-dependent, as different operators permit access to different intermediate stages—e.g., iterations/recursions steps or levels of approximation. How many partial results *should* be produced is user-dependent, as different tasks require different update rates as the resulting view is stepwise refined. This means that it may not be sensible to show every possible partial result, but maybe only every k^{th} result or to update the view only when a partial result differs at least by a given Δ from the previously shown one.

Enhanced transitions model the chunked data being passed among operators. They are represented through a number of marks that cut across the connector line. These marks denote the data flow to be

a stream of data chunks, as opposed to passing the full dataset at once, which is denoted by an unmarked connector (Figure 1c). In this case, we use the distance between the marks along the connector line as a visual cue to denote different sizes of the data chunks—i.e., the smaller the space between consecutive marks, the smaller the chunk size. A stream of chunked data can originate natively from a data source that delivers its contents in this manner or from an incremental visualization operator that sends its output in the form of subsequent versions of increasing completeness. It can also be generated by transforming a full dataset into a stream of data chunks or simply larger data chunks into smaller ones by sequencing the data (Figure 1d). Conversely, a sequence of data can be buffered to yield the full dataset again or simply to produce larger data chunks out of smaller ones (Figure 1e).

As a result of these two enhancements, subdivision and chunking are modeled independently: the subdivision of operators is modeled by enhancing the nodes and the chunking of data is modeled by enhancing the edges. A full model of an incremental visualization process can be assembled from these independent parts very much in the same spirit as the original DSRM does: data sources, operators, and final views are linked with each other through connectors. The resulting model differs only in the aspect that it explicitly shows in which parts of the visualization process chunking of data, subdivision of operators, or even both are utilized. In the following two sections, we use this base model to reason about incremental visualization and its more intricate details of chunking the data and/or subdividing the operators.

3.2 Chunking Data

Sequencing and buffering data are highly data type dependent operations. Hence, we give here an abstract discussion of the different factors that influence these operations in general and that can then be translated to the specifics of a particular data type, as for example a chunking of the incoming (raw) data has different requirements than a chunking of the image data that is to be displayed.

Sequencers chunk the incoming data and dispatch the resulting data chunks in a sequential order. This process can be configured via the parameters of *Data chunk size* and *Ordering strategy*.

The *Data chunk size* $s \in [1 \dots n]$ defines how small to chunk the data. Since different scenarios and sometimes even different operators of the same scenario ask for different data chunk sizes, this parameter is rarely set to a fixed value, but can instead be adjusted to cope with changing requirements or data properties. Such data properties can for instance be defined by different data levels, such as hierarchy levels, abstraction levels, and contextual levels [38]. As a result, chunks may have different sizes—e.g., when chunking the data by geopolitical regions.

The *Ordering strategy* defines the order in which the chunked data is dispatched. Since the user's interest in the data may change, different chunks of data may be of particular interest at different time points—e.g., sometimes the user may wish to see overall trends first, whereas at other times he or she may want to jump ahead to outliers or other features. Given that the chunks have been partitioned in a way that permits for quantifying their relevance with respect to the user's interest, they can also be ordered and thus prioritized according to that relevance to satisfy the user's interest as early as possible [17], [39].

Buffers merge a number of smaller data chunks into larger ones—either for efficacy reasons (smaller data chunks may not constitute meaningful input for the next operator) or for efficiency reasons (many small data chunks might produce more computation overhead than a few larger chunks). Buffering parameters determine when the buffer has accumulated enough chunks to dispatch them as one larger chunk. This can be defined, for example, through a desired *Data chunk size* or a given *Time interval*.

The *Data chunk size* $s \in [1 \dots n] \cup [\infty]$ sets the threshold to an output size that must be reached before dispatching the data chunk. Similar to the data chunk size defined for sequencers, data properties can be taken into account as well, in order to group the data on particular data levels that do not necessarily break down into partitions of equal size. For infinite data streams or data streams of unknown length, $s = \infty$. This will accumulate the incoming data forever, output the full data up to the current chunk, and repeat this for every incoming chunk.

The *Time interval* $t \in [0 \dots \infty]$ sets the threshold to include any data that arrives within the given interval at the buffer. For $t = 0$, a chunk is passed on as soon as it arrives without any accumulation, whereas $t = \infty$ has the same function as described above for the data chunk size. This parameter can be viewed as a frame rate and is used, for example, to prevent flicker caused by too frequent updates/redraws of the resulting view. Note that a time interval threshold can also be combined with a given data chunk size. These two thresholds can either be used in conjunction (triggering only when both are met) or in disjunction (triggering when either of the two is met).

Sequencers and buffers with different parametrizations can further be combined to produce a wide variety of chunking behavior. For instance, by using the sequencer as it is, the data gets chunked into disjoint data chunks, with the union of all data chunks forming the entire dataset. We call these *partition chunks*, as they contain only a part of the complete data. By simply adding a buffer with $s = \infty$ right after a sequencer, we can also generate data chunks that subsume all previous data chunks, with the last data chunk being equal to the entire dataset. We call these *revision chunks*, as each chunk represents a more

up-to-date/refined version that replaces any earlier chunk. The partial results of increasing quality originating from a subdivided operator, as described in the previous section, are examples of revision chunks. Note that these two types of chunks correspond to the chunking strategies for data identified by Mühlbacher et al. [40]: Their "Strategy S3: Divide and Combine" produces *partition chunks*, whereas their "Strategy S1: Data Subsetting" utilizes *revision chunks*. Yet differently configured sequencers and buffers can also be cascaded to yield more complex behavior. For example, it is possible to combine multiple sequencers to realize multi-level partitioning schemes, such as first chunking with respect to a spatial data property (e.g., by country) and then further chunking with respect to a temporal data property (e.g., by year).

3.3 Subdivided Operators

Operators that potentially lend themselves to subdivision are those that produce the final outcome in multiple stages or over a number of iteration/recursion steps. Both of these possibilities tie in with the two prevalent strategies for subdividing operators, as identified by Mühlbacher et al. [40]: Their "Strategy S2: Complexity Selection" applies *different algorithms* of increasing accuracy to yield intermediate results in stages until the exact computation is finished. Whereas their "Strategy S4: Dependent Subdivision" iteratively/recursively applies the *same algorithm* and outputs the intermediate results at each step. For the sake of clarity, we focus our discussions mainly on Strategy S4, but we show in the use case that our model applies to Strategy S2 in the very same manner.

In principal, an operator's subdivision has two aspects: the input side that configures the operator's subdivision via appropriate **parameters** (e.g., how many and/or how often intermediate results should be generated), as well as the output side that derives **metrics** about the quality and completeness of the (partial) results. We denote parameters and metrics in our graphical notation as triangles oriented towards or away from the operator, respectively (Figure 2).

The metrics are essential for various purposes, such as making the user aware of the performance of an operator (e.g., how much longer until the final result is produced?), allowing her or him to better understand the process behavior (e.g., is it converging or stabilizing?), as well as to judge the trustworthiness of the outcome (e.g., how representative is the current result of the overall result?). On top of communicating the metrics to the user, they can also be used to steer the process internally by using them as input parameters for the same or other operators—for example, using an error metric for step size control.

While the choice of suitable metrics depends on the actual use case, some metrics are quite general and useful in a broad range of applications. Specifically,

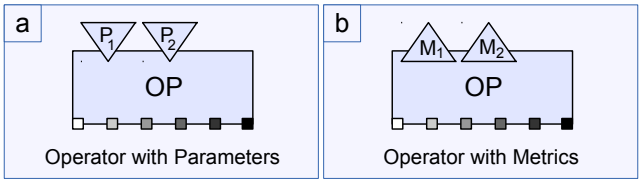

Fig. 2. Our graphical notation for operators with parameters (triangles pointing towards the operators) and metrics (triangles pointing away from the operators).

we determine three families of metrics to be of particular use for controlling and driving the process:

- for individual data chunks, *quality metrics* can be used to assess the quality of an element traveling the pipeline [41] (e.g., number of processing steps this data has already passed through);
- for two subsequent data chunks, *delta metrics* can be computed at data level and at view level to measure the difference between them;
- for the entirety of all data chunks having been processed so far, *error metrics* can be used to estimate the error of the current partial result as compared to the exact result (e.g., convergence of an iterative algorithm, error introduced by numerical approximations).

These metrics may and should be used in conjunction. For example, an output of high visual quality (i.e., a fully processed, polished high-res visualization) is meaningless if its error is too high, because the data subset it shows is too small to be representative. The same holds true for the opposite case of an accurate output (i.e., no error as it shows all the data) of extremely low visual quality that is unreadable for all practical purposes.

3.4 Combining Chunked Data and Subdivided Operators

There are two principal cases for such a combination, which can then be combined to form more complex scenarios: either the data chunking is performed before or after a subdivided operator is applied. In the following, we assume that the data chunking produces i chunks and the subdivided operator produces j (partial) results and thus j chunks.

In the case where the chunking is performed prior to the subdivided operator, the operator has different options of how to deal with the sequence of chunks that constitutes its input. These options can be modeled using a sliding window metaphor that assumes the parameter *Mem* as its window size. This parameter can take three principal values:

1) $Mem = 1$: the operator processes one chunk at the time and outputs at most j revisions for this one chunk before the next chunk arrives and the operator starts to process it instead;
2) $Mem > 1$: the operator considers a sequence of *Mem* consecutive chunks as input and also

outputs its j revisions for blocks of *Mem* chunks. This is used for operators requiring a minimal input size that is larger than the average chunk size, e.g., to guarantee statistical significance;

3) $Mem = \infty$: the operator uses all incoming chunks as an input. As a new chunk arrives, it is added to all the previous chunks and the output of the j revisions is updated accordingly.

The number of resulting chunks is at most $i \times j$ in the cases of $Mem \in \{1, \infty\}$, assuming that the operator computes all j stages before the next chunk arrives.

In the second case, a subdivided operator is executed before the data is chunked. The number of resulting chunks is at most $j \times i$, assuming that each intermediate result is fully sequenced into i chunks before the next intermediate result arrives. Note that in the same way as sequencers and subdivided operators multiply the number of chunks, buffers divide it and monolithic operators leave it unchanged.

To steer whether an operator is to be terminated early upon the arrival of a new data chunk (quantity-first strategy) or gets to finish the current chunk (quality-first strategy), we suggest to add a parameter called $Quality$ to subdivided operators. $Quality$ can be defined in terms of an operator's input by specifying how many new chunks have to arrive before a running refinement of the current result is aborted to take into account this new set of data chunks. Likewise, it can also be defined in terms of an operator's output by specifying how good the result must be at least (e.g., as measured by a quality or delta metric) before the operator can be interrupted with new data chunks. Using the input-oriented quality definition, a $Quality = 1$ would mean that the operator pursues a pure quantity-first strategy that restarts the operator as soon as a single new data chunk becomes available. This would push as much data through the pipeline as possible, regardless of the quality of the output. If $Quality = \infty$, the operator would pursue a pure quality-first strategy that lets the operator run through all j processing steps until the final result is produced, before restarting it for any new data that has arrived in the meantime. Any value in between denotes a compromise between these two extremes. Note that input-oriented and output-oriented $Quality$ definitions can also be used in conjunction to formulate more complex prioritization strategies.

4 Considerations for Applying the Extended Model

This section highlights strategies for working with the presented model. For this, we revisit the two challenges introduced in Section 2.3: the *authoring of incremental visualizations* through modeling the visualization process and the *use of incremental visualization* through steering the modeled visualization process at runtime—both at a *conceptual* and at a *practical* level.

4.1 Authoring Incremental Visualizations

At design time, the most critical question is to decide whether to chunk the data, to subdivide the operators, or both. This issue can be discussed from three different angles: from an *input perspective*, from an *output perspective*, and from a *processing perspective*.

The **input perspective** is basically what we have termed the reactive view on incremental visualization in Section 2.2. It subsumes considerations, such as:

- If the dataset does not fit the memory, it speaks for chunking the data.
- If the dataset is highly structured, it speaks against chunking the data.
- If the runtime to produce the desired output is rather long, it speaks for subdividing the long-running operator.
- If a long-running operator produces merely an optional embellishment and can easily be left out, it speaks against subdividing this operator.

Note that these are only a few examples of such considerations and each application scenario will require its own considerations from the input perspective.

The **output perspective** is in line with our proactive view on incremental visualization. It basically aims to identify the requirements of the output of the incremental visualization, rather than of the input. These requirements can be split in the two aforementioned strategies of *quality-first* and *quantity-first*. These can either be mapped onto the prioritization parameter $Quality$, or they can be "hardwired" into the pipeline with quality-first being realized through a data chunking, so that at least part of the data can be rendered in full as early as possible. Whereas quantity-first would be realized as a subdivision of the operators, so that the entirety of the data can be shown in some preliminary way as early as possible.

The **processing perspective** aims to streamline technical parameters, such as minimizing the number of data chunks, as each chunk may carry substantial data overhead or lead to a runtime overhead [29].

Once this decision has been made, it requires a number of practical considerations to realize the chosen subdivision in an implementation. For this, we suggest a multi-threaded software architecture that mirrors the process model with one asynchronous *worker thread* per operator. These threads have access to a priority queue of data chunks to work on. This setup is sketched in Figure 3. It generalizes similar architectural approaches, such as PIVE [32] that relies on two threads for the whole process.

In our setup from Figure 3, a thread basically works on a data chunk in an internal processing loop (c) that carries out refinement operations in an iterative fashion until the result matches a given break condition. In case this break condition is a delta value, the thread stops if two subsequent iteration steps only change the outcome by less than the given delta

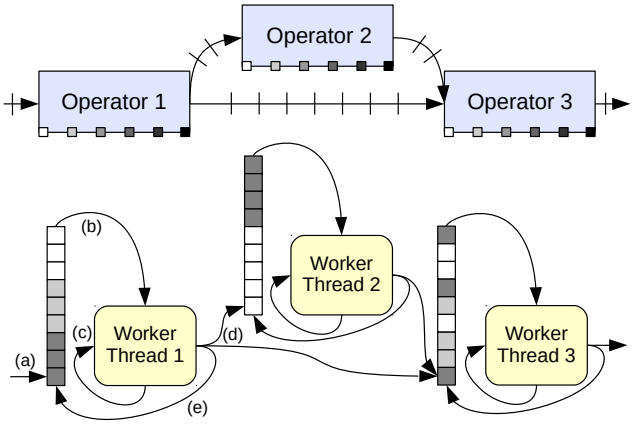

Fig. 3. Architecture using priority queues and asynchronous threads to realize our conceptual model. Incoming data chunks are added to the queue (a) and retrieved by the worker thread when idle (b). The worker thread computes better solutions in an internal loop (c) until a good enough result according to a given break condition is generated and passed on to subsequent threads (d). If the result is not yet good enough to be a final result, it is again added to the queue for further processing (e). Different shades of gray denote how close the queued data chunks are to the final result. These correspond directly to the marks at the bottom of the enhanced operators: white = unprocessed input chunk, gray = intermediate result, black = final result. As final results are not queued-up again, no queue contains a black data chunk.

value—i.e., the solution has become sufficiently stable with respect to that value to serve as an intermediate result and thus to be passed to the threads of the next operators in the pipeline (d). If the resulting data chunk does not yet fulfill a final condition—i.e., it is not yet stable enough to be considered the final result—it is again added to the job queue to be worked on further (e). This way, we can vary three parameters for each thread: the size of the data chunks, the prioritization of the data chunks in the queue, and the break condition—e.g., the delta value. These three parameters can be used to express the various ways of chunking:

The **size of the data chunks** directly expresses the amount of data chunking—i.e., data chunks of the size of the whole dataset correspond to no chunking and data chunks of the size of singular data items correspond to maximal chunking.

The **prioritization strategy of the queue** directly mirrors the processing strategy—e.g., in Figure 3, the first queue is prioritized by a quantity-first strategy that puts newly incoming and yet unprocessed data (white cells) at the top of the list, whereas the second queue uses a quality-first strategy that aims to finalize intermediate results (gray cells) first before starting on new data chunks (white cells), and finally the third queue uses a simple, unordered first-in/first-out strategy without any prioritization. It can thus be used to reflect the $Quality$ parameter set for each operator.

The **break condition** can be used to steer the degree of operator subdivision—i.e., if the delta value is set to 0 only stable and thus final results will be produced and the operator is effectively not chunked. Yet the larger the delta value is, the more intermediate results are produced as good enough approximations.

4.2 Using Incremental Visualization

Incremental visualization like any other visualization is first *configured*, then *run*, and finally interactively *observed* by a user. Depending on which of these three aspects the particular focus lies, we distinguish three scenarios of increasing user involvement.

Inspecting the visualization process focuses on *running&observing* the visualization as it unfolds on the screen. This scenario is all about witnessing the visualization process, in the sense that the observer gets to see step-by-step how the final visualization came about. It is not only common in the field of network visualization where iteratively refined layouts are subsequently displayed to illustrate the "untangling" of the graph, but, for example, also for animating (pre-)processing steps, such as clustering or sampling the data. Overall, it serves to better understand the visualization process and thus to be more confident that the visualization truthfully shows the data.

Tailoring the visualization process focuses on *configuring&running* the visualization in a way that conforms closely to the user's needs. This scenario is all about making sure that the visualization process is aligned with the goals of the user, in the sense that the visualization author carefully prioritizes what to process in which order. Prioritization relates here to data (i.e., chunk the data and order the chunks by decreasing importance) as well as to operators (i.e., subdivide the operators so that visualization layers of importance are finished first). For example, in a monitoring scenario, data chunks that simply *"confirm the expected"* can be held off until later, as they are unimportant, but chunks of *unexpected data* should immediately be taken into account and shown. By defining what constitutes *expected data*, the incremental visualization can easily be tailored by the visualization author to such an *unexpected-first strategy*.

Interacting with the visualization process focuses on *observing&configuring* the visualization according to what has been observed. This scenario is all about interactively adjusting the visualization's configuration to tune the visualization process towards the incoming data. For example, an adjustment of the $Quality$ parameter can easily resolve temporary "congestions" of the visualization pipeline, if the load of input data chunks rises unexpectedly. In practice, we suggest to utilize the graphical notation of our conceptual model for providing interaction handles for the various parameters of the subdivided operators, as well as for

the chunked data. Showing the incremental visualization process model alongside the generated view permits to inspect and adjust process details, such as operator metrics and parameter settings, respectively, through a common and consistent interface. Metrics are an important aspect of this scenario, as they give insight into the process if no visible changes occur in the visualization (Where is the hold-up?) and into its trustworthiness (How much of all data is already shown?). The information conveyed through the metrics can prompt the user to make on-the-fly adjustments to steer the running visualization process.

The following use case illustrates this latter scenario of the highest degree of user involvement.

5 Use Case

We applied our incremental visualization model to a concrete visualization scenario for the NTHSA FARS (Fatality Analysis Reporting System) dataset that covers all US car crashes between 2001 and 2009 [42]. This data is too large to be displayed in interactive frame rates, which makes it necessary to turn it into an incremental visualization of appropriate quality and with a justifiable error. The process of finding the proper configuration that fulfills these constraints is also demonstrated in the accompanying video.

5.1 Data Description and Requirements

The FARS dataset contains each car crash's latitude and longitude, as well as a number of characteristic attributes. These include the hour of the day, the number of involved people, the number of fatalities, the speed limit, the weekday, and the number of involved drunk people with an active role. This number can include pedestrians, as well as drivers of cars, trucks, motorcycles, or bicycles. In this scenario, it was chosen to explore the dataset by generating a density map representing the crashes that are similar to a chosen reference crash, i.e., showing the 5% most similar crashes. The similarity is computed using the Euclidean distance on a subset of normalized crash attributes. These crash attributes can be set by interacting with six sliders, which then issues a query for crashes with very similar attributes and plots them on a density map. A possible visualization pipeline realizing this procedure is modeled by the schema shown in Figure 4. Each time the user changes a query value through the sliders, the visualization pipeline starts a similarity search, produces a scatterplot of the resulting crashes, and computes a density map using a nonlinear mapping that optimizes the number of pixels that are assigned to each color.

However, the dataset contains more than 370,000 fatal crashes and processing them across the whole pipeline requires a processing time PT of about 4 seconds on an Intel i7 quad-core processor. This delay hampers a fluid exploration: each time the user moves

Fig. 4. Monolithic visualization pipeline for the crash similarity scenario.

a slider, he or she has to wait about 4 seconds before the result can be explored or the slider can be further adjusted. This makes it tiresome to compare visualizations for different queries and makes the interaction with the slider slow and unresponsive.

Hence, the designer of the system is looking for a solution that incrementally produces the visual result at a rate of ≥ 5 frames per seconds (FPS), i.e., $PT \leq 0.2$ seconds. With incremental visualization, it is possible to chunk data and subdivide operators on one side, and to monitor the PT value and the image quality of the output on the other side. Chunking the data increases the speed inversely to the size of the chunks—e.g., processing a chunk of 5,000 crashes takes about 0.05 seconds, allowing for reasonably fluid interaction. However, the result being the union of the 5% most similar data points that are individually determined per data chunk introduces a statistical error that is driven by the chunk size. The bigger the chunk, the smaller the error and the larger the PT. Yet, smaller chunks also introduce computational overhead and present fewer data in the first iterations.

To alleviate this problem, we can not only chunk the data, but in addition also subdivide the process, e.g., by introducing quicker density map algorithms than the original nonlinear mapping to compute early visual results. The designer of the system has now to find a suitable tradeoff between data chunking and process chunking that balances data quantity (i.e., minimizing the error) and image quality (i.e., maximizing the faithfulness of the output), while maintaining interactivity (i.e., keeping the visualization responsive). The following sections illustrate how to arrive at such a well-configured tradeoff and how to use it. This procedure follows three steps:

1. Modeling the incremental visualization pipeline. This entails to select the model components that capture the data and operator characteristics, focusing on how and where to chunk them. Some general model parameters, such as *Data chunk size* and *Quality* can be preset to suitable default values, while it is up to the user to correctly set semantics-dependent parameters, such as *Mem*.

2. Selecting process specific metrics. This step is about selecting metrics that are able to measure errors, quality, and differences in a way that it supports the user in steering the visualization process. This selection is dependent on the input data, the user's goals, and on the availability of metrics from the implementation of the involved operators.

3. Parametrizing for a quality/quantity tradeoff. This means to set the quality constraints and the data chunk size so that the incremental visualization allows for the desired interactive exploration by balancing the speed of the interaction against the quantity and quality of the data that is shown. This ties together the other two steps by observing the selected metrics and setting the model parameters accordingly.

5.2 Modeling the Incremental Visualization Pipeline

An incremental visualization pipeline that realizes the aforementioned data chunking of the FARS dataset and the subdivision of the density mapping operator is shown in Figure 5. It introduces a sequencer right at the beginning of the pipeline and each data chunk is sent to the `Similarity Search` operator that takes as parameters the characteristics of the actual reference crash and a similarity threshold. It processes each chunk independently ($Mem = 1$) and thus subsequently yields for each chunk the top 5% similar crashes to the chosen characteristics, with the percentage set by the threshold parameter. These most similar crashes are then sent to the `Scatterplot` operator that cumulates them in a scatterplot ($Mem = \infty$). The scatterplots are then sent to the `Density Map` operator that is subdivided by means of a "Complexity Selection" (cf. Section 3.3). It outputs three results generated by three algorithms of increasing computation time: a monochromatic mapping (a), a linear mapping (b), and the nonlinear mapping (c) as the original operator did. The monochromatic mapping is about 20% faster than the nonlinear mapping that requires to inspect the collision distribution to compute the color coding. In addition, it can simply pass on the incoming scatterplot, practically omitting the operator altogether. Chunking the data and subdividing the process allows for speeding-up the first output of a draft visualization using intermediate results.

Our prototype system features a UI that is tailored for investigating and confirming the efficacy of the incremental visualization pipeline (Figure 6). It displays the pipeline in the top-right view and selecting one of its elements opens a window at the bottom-right that shows its parameter settings and allows for changing them. Whereas the middle-right view shows metrics that measure the effects of any change.

5.3 Selecting Process-specific Metrics

The operators provide different metrics, useful to tune and monitor the process. The `Scatterplot` operator produces the quality metric $Error$ that estimates the percentage of incorrect data that is present in the final result due to computing the top 5% of similar crashes on each chunk individually and not on the whole dataset. The inspection of this metric, allows for evaluating the representativeness of the processed

Fig. 5. Incremental visualization pipeline for exploring the FARS dataset through similarity search.

data as compared to all data, looking for a tradeoff between speed and correctness of the displayed data.

The `Density Map` operator computes the quality metrics Q_x (with $x \in \{a, b, c\}$) of the different density maps based on the amount of processed data: $Q_x = C_x * \sqrt{data}$, where C_x is a quality coefficient of mapping x. Moreover, with the goal of optimizing our visualization for a certain degree of completeness in the visual output of the *Density Map*, we define the $\Delta vis(k, k+1)$ metric that compares two consecutive visualizations k and $k+1$ at pixel level:

$$\Delta vis(k, k+1) = \Sigma_{\forall n \in N} \frac{|color(d_{k+1,n}) - color(d_{k,n})|}{|N|}$$

where N is the set of pixels and $color(d_{k,n})$ is the color associated with pixel $n \in N$ in visualization k. Δvis can be used as a convergence indicator: When it is below a given threshold, it is possible to skip the rendering of this density plot and wait for one that differs more from the previous one. We might even decide to stop the overall process, assuming that the actual visualization is representative of the final one, as no more significant changes occurred.

In the screenshot in Figure 6, the three quality metrics of the `Density Map` operator are clearly visible in the detail view by their triangles pointing away from the operator block. The metric Q_a is currently selected, which triggers it to be plotted over the course of the running visualization process in the middle-right view. The "hold" checkboxes allow for freezing the selected metrics across different runs to compare their behavior for different parameter values.

5.4 Parametrizing for a Quality/Quantity Tradeoff

It is the user's goal to produce ≥ 5 FPS while at the same time achieving a visual quality and a data quantity that remain representative of the whole dataset. To do so, the user starts the system with data chunks composed of 1,000 tuples, in order to have a responsive system that allows for quickly inspecting different design choices. He or she starts inspecting the quality metrics Q_x (with $x \in \{a, b, c\}$) of the subdivided `Density Map` operator, to compare monochromatic, linear, and nonlinear density mappings. According to this strategy, the user starts tuning the process, optimizing the speed/quality-ratio by increasing the $Quality$ parameter, initially set to 1 (see Figure 6). This allows the `Density Map` operator to fully process each chunk. Hence, both a and c are computed on all chunks and the highest quality is achieved by output c. Yet at some point, the amount of data that contributes to mapping a is greater than

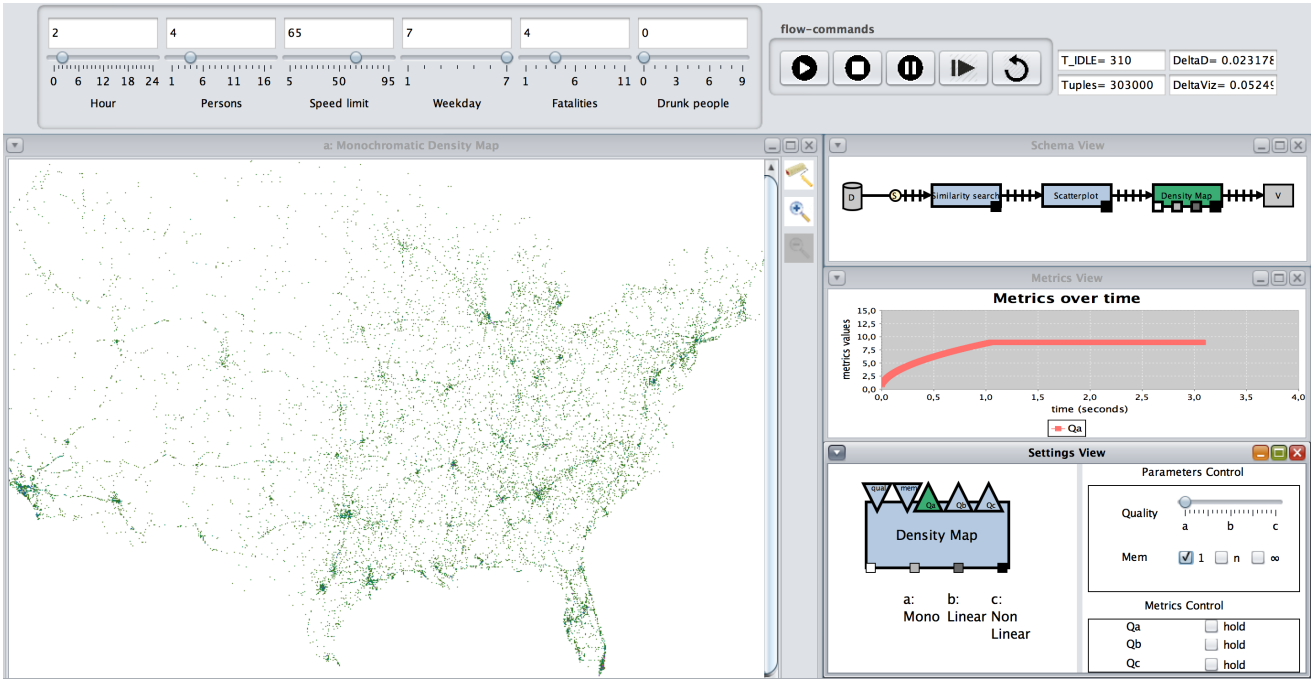

Fig. 6. Screenshot of our incremental visualization system. At the top, it shows the sliders with which to perform the similarity search, as well as buttons for controlling the process (start, stop, pause, step-by-step, restart). The four rearrangeable and resizable views show the visualization pipeline (top-right), the details of the selected `Density Map` operator (bottom-right), the actual output visualization of the first intermediate, monochromatic density map (left), and its quality metric Q_a plotted over time (middle-right). The *Quality* parameter is associated with a slider to provide a better understanding of its effect on the pipeline ($a = 1$, $c = \infty$).

the data required for mapping c and after 1.5 seconds mapping c outperforms mapping a (see Figure 7).

As 1.5 seconds is still too high for insuring interactivity, the user explores lower values for the $Quality$ parameter. It appears that mapping b, in order to produce an outcome that is comparable with the monochromatic mapping a, requires a time that does not allow for interactive exploration, so the user selects the monochromatic mapping a. Having fixed the operator subdivision, the user starts to increase the chunk size and finds that a size of 25,000 tuples produces close to 5 FPS. The error for this chunk size is 0.011, meaning that the results will contain 98.9% of correct data, which is acceptable for the purposes of the user. Changing the sliders confirms that these parameters are adequate for the task at hand: similarity searches can be fluently refined, as the first intermediate result is shown almost immediately.

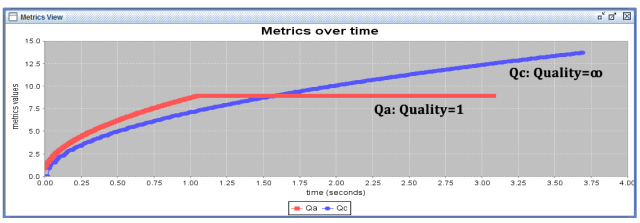

Fig. 7. Values of Q_a and Q_c metrics for two different values of the $Quality$ parameter.

5.5 Discussion

The above optimization of the incremental visualization parameters to fulfill all constraints happens in the space of different subdivisions of the `Density Map` operator and different chunk sizes of the data. The space of possible combinations of the two is depicted in Figure 8. By (re-)parametrizing the pipeline, the user navigates this space to find a solution that fits the criteria of quantity, quality, and interactivity. Looking at this space of possible visualization solutions, it is clear that data chunks with more than 29,000 tuples ($i \leq 13$ data chunks) will not produce a target frame rate of ≥ 5 FPS. Hence, useful solutions are above the line $i = 13$. Yet, too many small chunks produce higher errors and an overhead in the total computation time. In our system, chunks of 50 tuples produce a frame rate of about 3,000 FPS, but increase the processing time to 5.6 seconds. Moreover, slower density map algorithms produce better outcomes but process less data in the critical time frame of 0.2 seconds and might be visually less effective than simpler but faster algorithms. This is reflected by the user's choice of $i = 15$ data chunks and $j = 3$ processing steps.

It is noteworthy that changing parameters, like *Mem* or $Quality$, allows getting a better understanding of the consequences and interplay of subdivision and chunking. They make explicit some usually hidden implementation choices, which would normally be very hard to tailor to one's own use case without

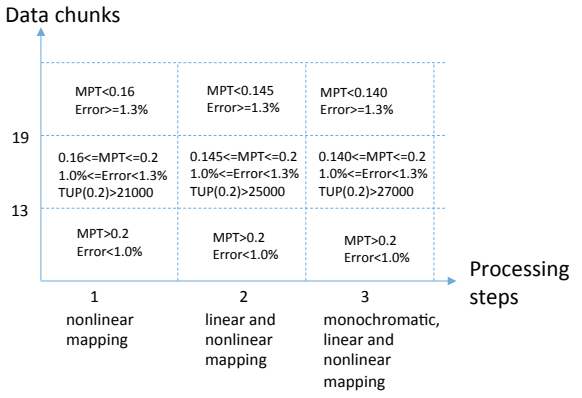

Fig. 8. Different combinations of chunking the FARS data and subdividing the `Density map` operator. Measured metrics are the Minimum Processing Time MPT (i.e., time needed to process a single chunk), the Error (i.e., expected percentage of incorrect data in the visualization), and the average number of plotted data tuples within 0.2 seconds (1 frame) TUP(0.2).

source code access. In cases where these parameters are not available, buffers and sequencers can help to use incremental visualization in a minimally invasive way that does not require re-implementing operators. For example, if the `Scatterplot` operator was not able to accumulate its results ($Mem = \infty$), we could also have used a buffer in between `Scatterplot` and `Density Map` to achieve the same.

6 Conclusion

As the example from the last section shows, with our generalized model it becomes possible to model, realize, and configure a desired balance between different visualization strategies. This allows us to compromise between existing solutions and even to deviate from them—tailoring the incremental visualization process and its behavior to the data's characteristics at design time (input-driven) and to the user's intended goal at runtime (output-driven). The model provides for a high-level representation of the incremental visualization process and presents a suitable means to interact with the visualization pipeline. The intervention points for this interaction are clearly embedded in the model and allow for steering the process in a transparent and immediate manner.

Using our approach internally to model incremental visualization processes in our software prototype has been a very beneficial experience that allowed us to find good default parameter settings by simply trying them out interactively during runtime. On top of that, the graphical notation helped us to clearly communicate with colleagues about incremental visualization processes, as it strictly separates the process architecture, consisting of chunked data and subdivided operators, from the process behavior, as it is defined by the parameters, such as *Mem* and *Quality*.

While our generalized model gives us the necessary terminology for specification and communication of incremental visualization processes, it does not give us any guidance on how to use it to produce well-formed and maybe even elegant incremental visualizations. One can see from the discussion in Section 5.5 that the space of possible data chunkings and operator subdivisions can be inherently complex—maybe even too complex for a solely interactive exploration. Output-oriented control heuristics and automated parameter optimization, as explored recently by Frey et al. [19], can help to find a suitable compromise within that space and to adapt it to changing demands. While we acknowledge these open research questions, it has to be noted that our generalized model with its parametrizable incremental operators and data sequencers opened up this space of possible subdivisions to systematic exploration in the first place. So, we see our model as a first stepping stone for future work in this direction by us and others.

Acknowledgments

We thank Dieter Schmalstieg for fruitful discussions on the architectural aspects of incremental visualization, as well as the reviewers for their feedback. Partial funding by the state of Mecklenburg-Vorpommern and EFRE within the project "Basic and Applied Research in Interactive Document Engineering and Maritime Graphics" is gratefully acknowledged.

References

[1] M. Angelini and G. Santucci, "Modeling incremental visualizations," in *Proc. of EuroVA'13*. Eurographics, 2013, pp. 13–17.
[2] E. H. Chi and J. T. Riedl, "An operator interaction framework for visualization systems," in *Proc. of IEEE InfoVis'98*. IEEE, 1998, pp. 63–70.
[3] R. B. Haber and D. A. McNabb, "Visualization idioms: A conceptual model for scientific visualization systems," in *Visualization in Scientific Computing*, G. M. Nielson, B. D. Shriver, and L. J. Rosenblum, Eds. IEEE, 1990, pp. 74–93.
[4] S. K. Card, J. Mackinlay, and B. Shneiderman, *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann, 1999.
[5] Advanced Visual Systems, "The Application Visualization System (AVS)." [Online]. Available: http://www.avs.com
[6] Kitware, "The Visualization ToolKit (VTK)." [Online]. Available: http://www.kitware.com
[7] M. Angelini, G. Santucci, H. Schumann, and H.-J. Schulz, "Towards a visualization process model for online visualization," in *Interactive Poster Session IEEE VIS'14*, 2014.
[8] N. Elmqvist, A. Vande Moere, H.-C. Jetter, D. Cernea, H. Reiterer, and T. J. Jankun-Kelly, "Fluid interaction for information visualization," *Information Visualization*, vol. 10, no. 4, pp. 327–340, 2011.
[9] E. H. Chi, "A taxonomy of visualization techniques using the data state reference model," in *Proc. of IEEE InfoVis'00*. IEEE, 2000, pp. 69–75.
[10] ——, "Expressiveness of the data flow and data state models in visualization systems," in *Proc. of AVI'02*. ACM, 2002, pp. 375–378.
[11] S. dos Santos and K. Brodlie, "Gaining understanding of multivariate and multidimensional data through visualization," *Computers and Graphics*, vol. 28, no. 3, pp. 311–325, 2004.
[12] A. Pang and N. Alper, "Mix&match: A construction kit for visualization," in *Proc. of IEEE Vis'94*. IEEE, 1994, pp. 302–309.

[13] K. Moreland, "A survey of visualization pipelines," *IEEE TVCG*, vol. 19, no. 3, pp. 367–378, 2013.
[14] J. A. Cottam, A. Lumsdaine, and P. Wang, "Abstract rendering: Out-of-core rendering for information visualization," in *Proc. of VDA'14*. SPIE, 2014, p. 90170K.
[15] K. I. Joy, "Massive data visualization: A survey," in *Mathematical Foundations of Scientific Visualization, Computer Graphics, and Massive Data Exploration*, T. Möller, B. Hamann, and R. D. Russel, Eds. Springer, 2009, pp. 285–302.
[16] D. Song and E. Golin, "Fine-grain visualization algorithms in dataflow environments," in *Proc. of IEEE Vis'93*. IEEE, 1993, pp. 126–133.
[17] R. Rosenbaum and H. Schumann, "Progressive refinement: More than a means to overcome limited bandwidth," in *Proc. of VDA'09*. SPIE, 2009, p. 72430I.
[18] M. Glueck, A. Khan, and D. Wigdor, "Dive in! Enabling progressive loading for real-time navigation of data visualizations," in *Proc. of CHI'14*. ACM, 2014, pp. 561–570.
[19] S. Frey, F. Sadlo, K.-L. Ma, and T. Ertl, "Interactive progressive visualization with space-time error control," *IEEE TVCG*, vol. 20, no. 12, pp. 2397–2406, 2014.
[20] C. D. Stolper, A. Perer, and D. Gotz, "Progressive visual analytics: User-driven visual exploration of in-progress analytics," *IEEE TVCG*, vol. 20, no. 12, pp. 1653–1662, 2014.
[21] J. Ahrens, K. Brislawn, K. Martin, B. Geveci, C. C. Law, and M. Papka, "Large-scale data visualization using parallel data streaming," *IEEE Computer Graphics and Applications*, vol. 21, no. 4, pp. 34–41, 2001.
[22] J. P. Ahrens, N. Desai, P. S. McCormick, K. Martin, and J. Woodring, "A modular extensible visualization system architecture for culled prioritized data streaming," in *Proc. of VDA'07*. SPIE, 2007, p. 64950I.
[23] H. T. Vo, J. L. D. Comba, B. Geveci, and C. T. Silva, "Streaming-enabled parallel data flow framework in the Visualization ToolKit," *IEEE Computing in Science and Engineering*, vol. 13, no. 5, pp. 72–83, 2011.
[24] W. Benger, G. Ritter, M. Ritter, and W. Schoor, "Beyond the visualization pipeline: The visualization cascade," in *Proc. of High-End Visualization Workshop*. Lehmanns Media GmbH, 2009, pp. 35–49.
[25] C. P. Botha and F. H. Post, "Hybrid scheduling in the DeVIDE dataflow visualisation environment," in *Proc. of SimVis'08*. SCS Publishing House, 2008, pp. 309–322.
[26] J. D. Mulder, J. J. van Wijk, and R. van Liere, "A survey of computational steering environments," *Future Generation Computer Systems*, vol. 15, no. 1, pp. 119–129, 1999.
[27] P. C. Wong, H. Foote, D. Adams, W. Cowley, L. R. Leung, and J. Thomas, "Visualizing data streams," in *Visual and Spatial Analysis: Advances in Data Mining, Reasoning, and Problem Solving*, B. Kovalerchuk and J. Schwing, Eds. Springer, 2004, pp. 265–291.
[28] J. A. Cottam, "Design and implementation of a stream-based visualization language," Ph.D. dissertation, Indiana University, November 2011.
[29] H. Piringer, C. Tominski, P. Muigg, and W. Berger, "A multi-threading architecture to support interactive visual exploration," *IEEE TVCG*, vol. 15, no. 6, pp. 1113–1120, 2009.
[30] S. Diehl and C. Görg, "Graphs, they are changing – Dynamic graph drawing for a sequence of graphs," in *Proc. of GD'02*. Springer, 2002, pp. 23–31.
[31] Y. Frishman and A. Tal, "Online dynamic graph drawing," *IEEE TVCG*, vol. 14, no. 4, pp. 727–740, 2008.
[32] J. Choo, C. Lee, and H. Park, "PIVE: A per-iteration visualization environment for supporting real-time interactions with computational methods," Georgia Institute of Technology, Tech. Rep. GT-CSE-13-06, 2013.
[33] D. Fisher, I. Popov, S. M. Drucker, and mc schraefel, "Trust me, I'm partially right: Incremental visualization lets analysts explore large datasets faster," in *Proc. of CHI'12*. ACM, 2012, pp. 1673–1682.
[34] A. Nandi and H. V. Jagadish, "Guided interaction: Rethinking the query-result paradigm," in *Proc. of VLDB'11*. VLDB Endowment, 2011, pp. 1466–1469.
[35] D. Fisher, S. M. Drucker, and A. C. König, "Exploratory visualization involving incremental, approximate database queries and uncertainty," *IEEE Computer Graphics and Applications*, vol. 32, no. 4, pp. 55–62, 2012.
[36] F. Olken and D. Rotem, "Random sampling from database files: A survey," in *Proc. of SSDBM'90*. Springer, 1990, pp. 92–111.
[37] A. Dix and G. Ellis, "By chance – Enhancing interaction with large data sets through statistical sampling," in *Proc. of AVI'02*. ACM, 2002, pp. 167–176.
[38] M. Lux, "Level of data – A concept for knowledge discovery in information spaces," in *Proc. of IV'98*. IEEE, 1998, pp. 131–136.
[39] J. M. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman, T. Roth, and P. J. Haas, "Interactive data analysis: The Control project," *IEEE Computer*, vol. 32, no. 8, pp. 51–59, 1999.
[40] T. Mühlbacher, H. Piringer, S. Gratzl, M. Sedlmair, and M. Streit, "Opening the black box: Strategies for increased user involvement in existing algorithm implementations," *IEEE TVCG*, vol. 20, no. 12, pp. 1643–1652, 2014.
[41] A. Dasgupta and R. Kosara, "The importance of tracing data through the visualization pipeline," in *Proc. of BELIV'12*. ACM, 2012.
[42] National Highway Traffic Safety Administration (NHTSA), "Fatality Analysis Reporting System (FARS)," 2013. [Online]. Available: http://www.nhtsa.gov/FARS

Hans-Jörg Schulz received a master degree in computer science in 2004 and his PhD in 2010 from the University of Rostock. He is currently working as a senior researcher at the Fraunhofer IGD Rostock on industrial and academic projects in the domain of Big Data Analytics. His research interests include graph visualization, visualization design spaces, and Visual Analytics for biomedical and systems biological applications. More about his research can be found at http://www.informatik.uni-rostock.de/~hs162/.

Marco Angelini is a PhD student at the Department of Computer Science of Sapienza Universitá di Roma, where he received his MSc. in Computer Science in 2011. His main research activities concern data analysis, Information Visualization and Visual Analytics, focusing on predictive analysis. He applies his research in Information Retrieval and Cyber-Security domains. More about his research can be found at http://www.dis.uniroma1.it/~dottoratoii/students/marco-angelini.

Giuseppe Santucci is associate professor at the Department of Computer Science of Sapienza Universitá di Roma, where he teaches courses on Python, Software Engineering, and Infovis (in English). His main research activities concern human computer interaction and Visual Analytics, focusing on evaluation and quality aspects. On such topics he has published more than 100 papers on international Journals and conferences. More about his research can be found at http://www.dis.uniroma1.it/~santucci/.

Heidrun Schumann is heading the Computer Graphics Research Group at the University of Rostock since 1992. Her research covers Information Visualization, Visual Analytics, and Rendering. Her current projects, supported by funding agencies and industry, include scalable frameworks for information visualization and adaptive visual interfaces. More about her research can be found at http://www.informatik.uni-rostock.de/~schumann/.