STAG: Smart Tools and Applications in Graphics (2019)
M. Agus, M. Corsini and R. Pintus (Editors)

ReVize: A Library for Visualization Toolchaining with Vega-Lite

Marius Hogräfer[1] and Hans-Jörg Schulz[1]

[1]Department of Computer Science, Aarhus University, Denmark

Abstract

The field of tools for data visualization has been growing in recent years, with each tool contributing new ways to create and work with visualizations, and each offering a specialized set of features, interaction metaphors and user interfaces. This means on one hand that users have a wide choice in visualization tools. On the other hand, though, this choice might also lock-in the user: Once made, it becomes difficult and sometimes even impossible to switch to another tool – e.g., to further refine a visualization made in one tool inside another. In turn, users are forced to work around any shortcomings of the chosen tool, as switching to another tool is even more cumbersome. In this paper, we introduce ReVize, an open-source library for visualization toolchaining. ReVize makes use of Vega-Lite as a common exchange format to be able to add toolchain support to web-based tools. In contrast to existing approaches, this solution to visualization toolchaining allows for authoring a visualization with multiple tools in a back-and-forth fashion, without a preset order in which tools are to be used. We demonstrate ReVize by adding toolchain support to three existing tools – KNIME, ColorBrewer, and VisFlow – for using them in concert to author visualizations.

CCS Concepts

• ***Software and its engineering*** → *Software tools and libraries;* • ***Human-centered computing*** → *Visualization systems;*

1. Introduction

Working on a data visualization project usually involves a number of specific steps that are supported by specific tools: OpenRefine might be used to scrape and preprocess the data. R is then invoked to run some data aggregation. The resultant dataset is then loaded into Lyra [SH14] to create the mapping between data and visuals. Multiple excursions are made from Lyra to ColorBrewer [HB03] for deciding on a suitable color scale. And in the end, the visualization is then included in a Jupyter Notebook [KRkP*16] for record keeping and later publication.

We term such a sequence of tools used for data visualization a *visualization toolchain*. Each tool in this toolchain fulfills a certain role along the visualization pipeline [CMS99]; some tools focus on data pre-processing while others allow for the interactive definition and adjustment of the mapping. This results in a wide variety of visualization tools [GBTS13, MMWC18, SLR*20, Kir19].

Getting back to our example above, let us assume that a collaborator finds a problem with the final visualization in the Jupyter Notebook. To fix it will require adjustments to the preprocessing of the data carried out in OpenRefine. While doable, this will in turn necessitate a re-run of the entire toolchain to re-generate the visualization with the now improved data.

The reason for having to re-run the subsequent steps of the toolchain is that no information about the transformation between steps is stored in the file formats. In the data space for example, CSV files are passed between tools. The mapping transforms those into the geometry space, for example encoding its output as SVG files. And those are finally rendered in image or view space, for example resulting in PNGs. Not only does each transformation between formats lose all information about the previous ones, but also within one stage of the pipeline no process information is retained. There is no part in a CSV to store a prior, e.g. unaggregated set of data; there is no part in an SVG to store a prior, e.g. unbundled set of edges; and there is no part in a PNG to store a prior, e.g. unsmoothed version of a heatmap. All this process information gets lost when moving from one tool in the toolchain to the next.

This is the point where we advocate to shift from "lossy" file formats like the ones mentioned above, to declarative visualization grammars. These grammars are able to describe, store, and exchange visualizations in all their aspects from filtering and aggregating the data via color mapping and labeling all the way to defining interactions. As a result of this shift, any visualization tool can access and modify any part of the same full visualization description at any time without incurring the need to re-run the whole toolchain.

While this shift to visualization grammars is well under way with grammar-based visualizations becoming more and more popular, tool support is lagging behind. This is particularly so for tools being able to not only import and render such grammar-based visualizations, but also to export a visualization to a declarative grammar [SH14, WQM*17]. This might be due to the growing complex-

DOI: 10.2312/stag.20191375

ity of existing grammars such as Vega and Vega-Lite [SMWH17], which probably discourages authors from implementing the required functionalities. After all, importing and exporting a visualization grammar is a little more involved than parsing and writing CSV files. Yet, such import/export functionality for a visualization grammar would be needed to use multiple, different visualization tools in such an ad hoc manner and without incurring dependencies on other tools (i.e., the need to re-run).

At this crucial point, our toolchaining library – ReVize – comes into play, which we present in this paper. ReVize is an open-source JavaScript library for extending web-based visualization tools with toolchaining support by allowing for the import and export of Vega-Lite specifications. To handle the aforementioned complexity of Vega-Lite, ReVize hierarchically deconstructs composite views of a visualization specification, which allows us to access encodings and configurations on a per-view basis. Additionally, the model maintains a bi-directional mapping between data and views that allows exporting partial views of a composition as independent declarative specifications. We present architectural constraints and lessons learned from using ReVize to extend three visualization tools with toolchaining support (ColorBrewer, KNIME, and VisFlow) and discuss the use of a common exchange format between visualization tools based on a use case. ReVize is published on the node package manager repository and on Github (see `https://www.npmjs.com/package/revize`).

2. Related Work

In this section, we present the literature on how visualization is shared across toolchains. We divide this problem into two parts: First, we present approaches to working on a visualization within a single tool and then present ways in which a visualization is shared between multiple tools.

2.1. Authoring a Visualization within the same Tool

To begin, we briefly discuss the different approaches in the literature, in which a visualization is authored within one tool.

The first approach is the use of proprietary file formats to store a visualization's structure for future editing. This approach is often implemented in "monolithic" tools, in which all aspects of a visualization can be manipulated, such as Lyra [SH14] and Charticulator [RLB19]. While limiting the authoring process to a single tool, proprietary formats nevertheless enable refining an existing visualization or reusing its structure as a template for other datasets.

Another common technique in this category are multiple coordinated views (MCV) [Rob07, JE12]. Therein, a visualization is viewed and authored inside one tool in context of other views on the same dataset. Whenever the user interacts with the data in one view, all other visualizations are also updated. MCV is commonly achieved through the Observer pattern, in which the tool notifies views about changes of shared variables, which are manipulated by the user. For example, Improvise [Wea04], Snap-Together [NS07] or VisFlow [YS17] implement MCV as linked selections. Lark [TIC09] allows explicit manipulation of different steps along the visualization pipeline in a joined view.

An approach related to MCV are computational notebooks such as Observable [Obs19] or Jupyter Lab [Jup19], where users can edit a visualization's source code directly and see immediate changes in the resulting view. These notebooks can be stored for later refinement and shared with others. In Idyll [CH18], authors can make parameters of experiments discussed in an article modifiable. Then, readers can change these variables and observe their influence on the outcome of the experiment, as all dependent part of the text and diagrams in the article automatically update.

Our toolchaining approach has similarities with single-tool approaches and expands on them. First, similar to MCV and computational notebooks, a central model of the visualization is viewed and edited using different actors. While in MCV, these actors are views, in toolchaining they are full visualization tools. Second, similar to proprietary file formats, a model of the visualization pipeline can be made persistent for future editing, albeit in toolchaining more than one tool can load the model.

2.2. Authoring a Visualization in multiple Tools

Next, we present how a visualization can be authored by multiple tools. Splitting the visualization authoring process across tools is rather intuitive, since the process of developing a visualization is often conceptualized as a multi-step process, for example with the visualization pipeline model [CMS99] or the data state reference model [CR98]. Hence, the transformations between steps of these models are split among multiple tools, with each tool being specialized to the particular transformation for which it is used. Here, we distinguish between two general strategies of modifying a visualization in multiple tools: consecutively and in parallel.

2.2.1. Consecutive Usage of Visualization Tools

In this first category, tools are executed consecutively and isolated from each other.

Coupling visualization tools through exchange formats is a common solution for tools that are developed independently from each other. Tools that fall under this category share information by passing datasets through comma-separated values (CSV), or files in JavaScript object notation (JSON) [BOH11, SH14, MNV16, SRHH16, MEC*17, RLB19]. They often also allow for generating vector graphics or bitmaps [SRHH16, SMWH17] to modify visualization details with graphic editing tools such as Inkscape [Poj19], Illustrator [Ado19] or GIMP [Tea19] (see Figure 1). In contrast to the parallel approach, usually no mediating instance is needed as information is shared through either the file system by storing data in a file, the memory by copy-and-pasting data or by accessing data from a remote server. One example is the visualization authoring tool Charticulator [RLB19], which can import data from CSV and TSV files, and produces a visualization for that data in publishing formats such as bitmap, vector graphics, HTML snippets or as templates for Microsoft Power BI.

A limitation of coupling tools in this way was identified by Bigelow et al. as the "challenge of iteration" [BDFM17], in that it is not possible to use other visualization authoring tools to iteratively refine these results. Thus, the authors present an approach, in which

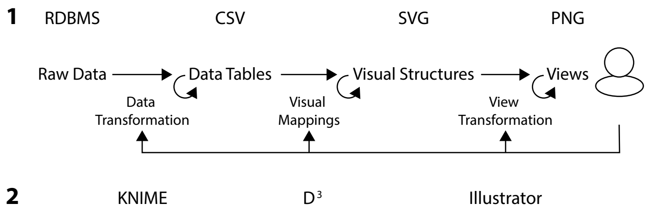


Figure 1: *A visualization scenario with consecutive combination of tools: Using a dedicated exchange format (1) between two stages of the visualization pipeline, a specific visualization tool (2) is applied. Output formats do not describe the operations that are applied to the input data in each tool (Visualization pipeline adapted from [CMS99]).*

a core model captures changes to a visualization from multiple connected tools and derives a common merged version [BDFM17]. While each tool can only represent parts of the full model, a diff-mechanism computes the partial models for each tool. That way, tools with very different feature sets can be used iteratively, for example Adobe Illustrator and D^3.

Another way of exchanging visualizations between tools are visualizations defined in visualization grammars. These use a common set of rules that resemble natural language rather than imperative code as presented in the *Grammar of Graphics* by Wilkinson [Wil05]. Wilkinson's work has been adapted in different visualization grammars, such as ggplot2 [Wic10], Brunel [IBM19], Vega [SRHH16], and Vega-Lite [SMWH17]. Through the declarative approach, the structure of a visualization can be shared across tools by distributing the description used to generate it, rather than its rendered graphic representation. Yet, to our knowledge no visualization framework currently allows importing such declarative visualization descriptions for further modification.

In our toolchaining approach, we build on Bigelow's solution through the merits of declarative visualization grammars. Bigelow's core model covers aspects of the visualization pipeline based on a set of supported tools. Whenever a new tool is added to the core model, this feature set is extended to incorporate the novelties for that tool. In our toolchaining solution, we use a full model of the visualization pipeline (i.e. a visualization grammar) as our exchange format. In that sense, the model exchanged between tools is already "complete" and does not change per added tool. Nevertheless, Bigelow's solution can potentially incorporate tool-specific features that are not part of the visualization pipeline, for instance the arrangement of toolbars in the workspace of a specific tool.

2.2.2. Parallel Usage of Visualization Tools

In the second approach to visualization toolchains, tools are run in parallel to each other, with shared access to a model of a visualization. This approach usually requires a mediating layer that manages the access to the model and that propagates changes to all tools [SRN*19].

One example for this approach is the Obvious toolkit [FHBW11], which enables using multiple visualization toolkits in one project. Obvious achieves this by providing a meta-model for toolkits, allowing developers to combine instantiations of this meta-model from any supported framework.

Our toolchaining approach conceptually resembles Obvious' meta-model approach. In both cases, an abstract model of the visualization pipeline is used to combine feature sets. For toolchaining however, we propose using a declarative description rather than imperative code.

Rogowitz and Matasci presented an approach to visualization toolchains [RM11], in which metadata about changes is propagated between two tools. For this purpose, each tool implements a dedicated interface for one or more other tools in the process. Whenever changes are made in one tool, metadata describing the changes is propagated through the network of tools.

ManyVis allows integrating multiple visualization tools in a single application view by providing a communication layer for coordinating process management, window management, user input, and any communication between applications [RSD*13]. This allows users to create custom interfaces from a selection of tools.

Our approach to visualization toolchaining also supports such parallel use of tools. To achieve this, the visualization model can be distributed between any number of visualization tools. Nevertheless, this requires an external communication layer that distributes this model between tools (see Section 8).

3. The Challenges of Tool-Interoperability in Visualization

In this section, we present conceptual considerations on authoring visualizations across toolchains through declarative visualization grammars. We do so by motivating three requirements for a library that aims to provide such toolchain support.

The library allows importing and exporting a full model of the visualization pipeline (R1). As outlined in Section 2, some approaches exist for combining multiple tools to work jointly on one visualization. Yet, current visualization toolchains generally involve a particular set of tools that is used in a specific order.

One problem with this are "asymmetric" interfaces, in that each tool imports and exports information about different parts of the visualization pipeline. In turn, it is usually impossible to combine multiple tools for the same purpose without additional support, as the data exported by one tool does not contain the information that the other needs. A visualization exported from Lyra in SVG cannot be edited further in Charticulator, since the format describes the geometry but contains no information about the data that is represented. Since every tool comes with its own benefits and drawbacks, users are forced to make tradeoffs when deciding on a visualization toolchain. This hinders users' flexibility when authoring a visualization.

To solve this, a full model of the visualization pipeline must be available to every tool in the toolchain. That way, each tool can modify the specific parts of the visualization and express them in that model (see Figure 2). Furthermore, using a full model of the visualization pipeline addresses information-theoretic problems discussed by Chen et al. [CJ10]. This is because information that is usually lost when using "partial" exchange formats can always be recovered when the full pipeline is available.

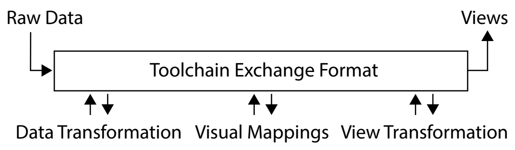


Figure 2: *Illustration of our toolchain approach using a common exchange format: Given an input dataset, the visualization pipeline is modeled in the exchange format. Each tool in a toolchain can thus express its modifications to the pipeline by authoring the exchanged file. Each tool can render a preview based on the exchanged file.*

The library allows for iterative use of visualization tools (R2). The second challenge for visualization toolchains is the "challenge of iteration" [BDFM17], which we have discussed in Section 2. Refining a visualization with another tool is hindered, if for instance the link between a visual mark and its underlying data item cannot be retrieved from a bitmap image. To solve this, the exchange format used by the toolchain library should model all aspects of the visualization pipeline, thus making any feature of the visualization available to any tool at any point in time.

The library supports both parallel and consecutive use of visualization tools (R3). As discussed in Section 2, there are generally two approaches to combining multiple visualization tools – consecutive and parallel – with each having its own merits. Using multiple tools through a mediating communication layer allows utilizing a variety of features at the same time. In case of the consecutive use of tools, no such mediating layer needs to be maintained, allowing for independent development of tools. Therefore, a toolchaining library should facilitate both usage scenarios.

4. ReVize: A Library for Toolchains in Visualization

In this section we present implementation details on ReVize, our library for visualization toolchains utilizing a declarative visualization grammar. First, we present design decisions for the library, then briefly introduce the component architecture, and then describe details about its import and export functionalities.

4.1. Design Decisions

Here, we discuss how we implemented the design requirements from Section 3 in ReVize.

Realizing R1, we chose declarative visualization grammars as an exchange format between tools. Visualization grammars model the visualization pipeline through structural descriptions using a common set of rules (see Section 2). Specifically, we decided to use Vega-Lite as exchange format for ReVize. Other alternatives for visualization grammars were Vega, ggplot2 and Brunel. As Vega-Lite is JSON-based, parsing and distributing the visualization description is significantly more convenient than it is in the case of Brunel and ggplot2. At the same time, visualization tools that are not based on web technologies in the future can still become part of a toolchain, since JSON is not restricted to JavaScript (see for example the Python-based Altair [VGH*18]). While Vega offers more freedom in what can be expressed (R2), Vega-Lite's concise structure and use of defaults reduced the implementation complexity of ReVize significantly. However, this conciseness also introduces limitations into ReVize, which we will discuss later in Section 7.

We address R2 in two ways in our design decisions: First, we designed ReVize around two main components, the SpecParser class providing the `parseSpec()` function and the SpecCompiler class providing the `compileSpec()` function. These are functionally inverse, in that one produces view objects from specifications and the other produces specifications from view objects. As a result, unlike the solution of Bigelo et al. discussed previously, no core model needs to be maintained, as all modifications to the shared visualization are expressed through specifications in the declarative grammar.

The fact that visualization grammars are used as an exchange format also ensures that R3 is fulfilled: Since a complete model of the shared visualization is exchanged between tools, rather than meta information, ReVize can be utilized both in consecutive and parallel settings. While parallel use requires an additional scheduling mediator between tools, ReVize itself does not prevent this use case.

4.2. Implementation

Here we briefly present implementation details about ReVize. A common problem hindering tool interoperability is that tools are developed for different platforms, i.e. programming languages and application programming interfaces (APIs) [GAEk*16]. State of the art visualization tools are currently based on web-technologies. To promote tool interoperability with current visualization tools, we implemented ReVize to provide a uniform API for visualization tools using TypeScript and additionally compiling the sources to JavaScript. Therefore, ReVize can be included as a library in an existing project as package from npm or linked to directly from within an HTML file.

4.3. Parsing a Vega-Lite View Composition

Vega-Lite specifications contain views, which define a mapping between data and visual marks. These views can be either atomic, meaning that they directly encode the relation to the data or they can be composite, in which case they define an arrangement of one or more sub-views. These arrangements can be vertical or horizontal concatenations of views (aligning multiple views next to each other), overlaying (placing views on top of each other), faceting (representing value subsets of the data in separate plots) and repetition (representing multiple dimensions of the data in separate plots).

A difficulty of using Vega-Lite as an exchange format between tools is its growing syntactical complexity. While it is more concise than its superset Vega, the set of abstractions that are part of Vega-Lite is constantly growing. Thus, it makes sense to centralize the development efforts into a toolchaining library.

Additionally, directly editing partial views from a Vega-Lite specification is also non-trivial. This is due to dependencies that

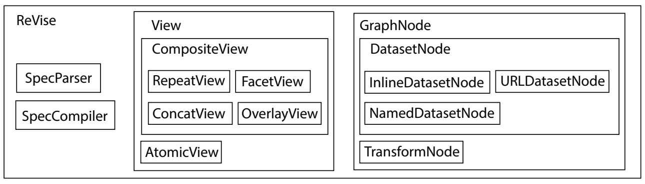


Figure 3: *Architectural diagram of ReVize's main components.*

some view composition operators introduce into the visual mapping of a child view. Views on deeper levels of the hierarchy inherit general specification configurations such as the width and height of the view space of the higher levels, unless they provide a new configuration, in which case higher-level values are overwritten. Furthermore, child views potentially inherit visual encodings and configurations from their parent, which can easily be overlooked. This makes it impossible to edit a child view independently from its parent. It is desirable for visualization toolchaining, to hide such constraints from the user of an API.

ReVize provides two components that serve as interface to Vega-Lite: The SpecParser and the SpecCompiler classes, the former for importing Vega-Lite specifications and the latter for exporting them. The compiler component produces Vega-Lite specifications for any given view object recursively from its child views in a straightforward manner. The parser takes care of disassembling the complex view hierarchy mentioned above, recursively traversing Vega-Lite specifications and creating view objects for each view type it encounters, while storing the parent-child relationships in the respective properties of that object. An overview of ReVize's architecture is depicted in Figure 3. Listing 1 shows pseudocode for the view decomposition algorithm of ReVize, with Listing 2 illustrating the process of inferring visual encodings from a repeating view composition. The full implementation can be found in the Github project of ReVize.

```
function parseSpec(vegaLiteSpec)
  configuration <- getConfig(vegaLiteSpec)
  datasets <- getAllDatasets(vegaLiteSpec)

  view <- getView(vegaLiteSpec)

  view.datasets <- datasets
  view.configuration <- configuration
  return view
```

Listing 1: *Pseudocode for the parseSpec function of ReVize. The view composition is traversed recursively in the getView function.*

4.3.1. Parsing the Data Transformation Pipeline

Vega-Lite offers different ways to represent data in a specification: As an inline dataset, for which values are stored directly in the specification, via a URL from which values are retrieved, or as a named dataset that references an object containing entries of the other two types. While parsing the view hierarchy, ReVize looks for these three types of data and stores references to DatasetNodes in the respective instance of the view class.

In addition to data sources, Vega-Lite offers some basic data transformation operators, which are stored separately from the data. If ReVize encounters a transformation entry, it creates a TransformationNode for each encountered operator and links them in a transformation pipeline. Instead of referencing a DatasetNode in the respective view instance, the TransformationNode is used. This allows ReVize to express data transformation graphs via Vega-Lite's linear transformation lists.

```
function getRepeatView(vegaLiteSpec)
  repeatedSpec <- vegaLiteSpec.spec
  repeatedFields <- repeatedSpec.repeat
  repeatedEncodings <-
    getRepeatedEncodings(repeatedSpec)

  parentView <- new RepeatView()
  childView <- getView(repeatedSpec)

  for (encoding in repeatedEncoding)
    childView.encoding = repeatedFields.rand()

  parentView.childViews += childView
  return parentView
```

Listing 2: *Pseudocode for the decomposition of a Vega-Lite specification that uses a repeating view composition. Visual mappings that reference a "repeat" entry are overwritten for the child view with a direct reference to a field of the dataset.*

4.4. Using ReVize to enable existing Tools with Toolchain Support

ReVize can be used either as a module from npm or as a bundled JavaScript library from GitHub. To extend an existing visualization tool with toolchain capabilities, generally two components need to be instantiated: SpecParser and SpecCompiler. Toolchains support is then added through their respective interface functions `parseSpec()` and `compileSpec()`. The returned view element of the former gives access to the view composition and the individual configurations of each view in the visualization specification provided as a parameter. The latter returns a visualization specification representing a view object that was passed as a parameter.

In addition to the functions provided by ReVize, it is usually necessary to extend the user interface of a tool with widgets that provide ways of importing and exporting a specification. For consecutive toolchains, a text field into which specifications are pasted from the clipboard is often sufficient. Another option is to implement a file input that loads a JSON file from local storage. Exporting the visualization specification in turn can be achieved through a text element from which the specification can be copied into the clipboard, or even by reusing the import text field. For parallel toolchains, a connection to the mediating instance needs to be implemented, for example as web socket or REST interface. Exporting a specification in turn can be implemented as another operation on the respective interface. We want to emphasize that ReVize does not prescribe using either parallel or consecutive toolchains, but can be adapted to both scenarios.

Examples for adding toolchain support to existing tools are presented in the following section.

5. Case Study: Extending Visualization Tools with Toolchain Support using ReVize

Using ReVize, we added visualization toolchain support to three existing visualization tools: The data analytics tool KNIME, the color scale manager ColorBrewer, and the visual analysis editor VisFlow. In case of the latter two tools, we have forked the current release of each project from GitHub and published a toolchain-enabled fork respectively under open-source licenses again. In case of KNIME, we have created a workflow for toolchain support that can be imported into any instance of that tool.

For VisFlow and ColorBrewer, we used the npm version of ReVize and imported specific components of ReVize into the application, whereas for KNIME, we used requireJS to download the bundled version. In each tool, we began by extending the graphical user interface with two components: a text widget into which we could copy and paste a specification and a DOM element in which to render the visualization. For KNIME, the text widget was the input field of a textual dataflow variable, for VisFlow and Color-Brewer we added HTML textarea elements. In all three cases, the specification is rendered by the Vega-Embed module, which also automatically adds an export interface to Vega-Lite and Vega to the view.

The Vega-Lite specifications where parsed using JavaScript's `JSON.parse` function. If no error was thrown by that function, the resulting object was passed to the `parseSpec` function of a SpecParser object.

In the next step, the visual encodings are presented to the user. For this, each tool provides a way to select a view from the Vega-Lite specification. After selecting a view, the visual encodings of that view can be modified. In ColorBrewer and VisFlow, widgets for each color-related visual variable were provided. In KNIME, encodings were set through workspace variables.

A modified view object was then passed to a SpecCompiler instance, which generated a Vega-Lite specification. This specification was then rendered in a DOM element using Vega-Embed. In the following, more detail is given on the specific implementations.

5.1. KNIME

KNIME [DTK*09] is a dataflow-driven analytics platform, providing a graphical interface in which many standard data analytics operators can be combined into a data processing pipeline. KNIME provides a few standard visualization techniques and allows for some basic customization. Nevertheless, this selection is limited to standard charts types and the resulting visualizations cannot be authored further outside KNIME. We have created a workflow that allows importing arbitrary Vega-Lite specifications and rendering them in a web view, using KNIME's generic JavaScript node. The resulting visualizations including data pre-processed by the KNIME operators can be exported as Vega-Lite specifications.

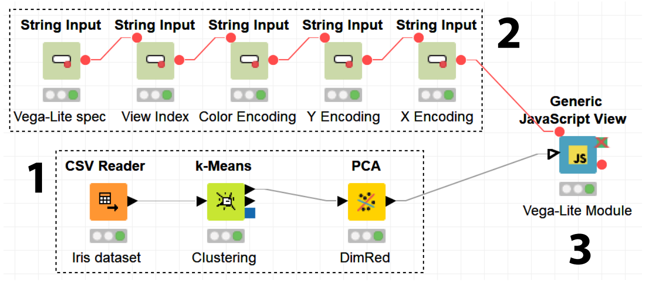
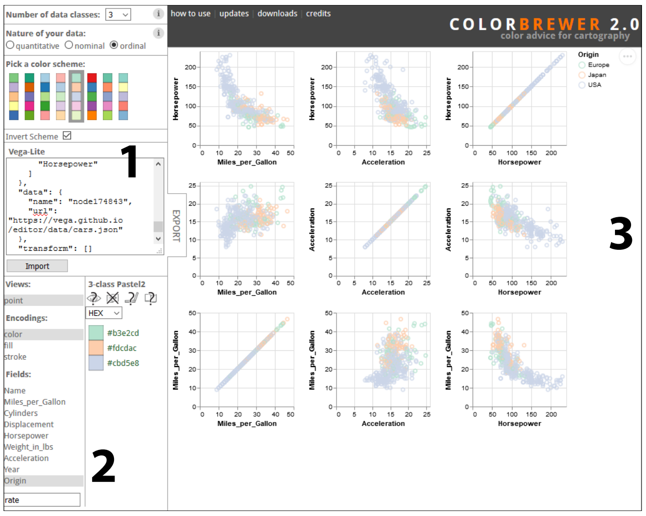


Figure 4: *Screenshot of a KNIME workflow that supports toolchaining: (1) A data analytics pipeline is applied to a dataset read from a CSV file. (2) The encodings of the visualization can be configured through FlowVariables. (3) A generic JavaScript node uses ReVize to apply the configuration to the visualization specification and induces the dataset.*

For this purpose, the Vega-Lite specification and its configuration are passed to the generic JavaScript node as FlowVariables. Using ReVize, the visual encodings are set and the dataflow is passed to the root view of the specification. A screenshot of the grammar-based dataflow is depicted in Figure 4. The toolchain-enabled workflow can be downloaded from `https://vis-au.github.io/toolchaining/knime`.

5.2. ColorBrewer

ColorBrewer [HB03] is a frequently-used reference website for color schemes, offering perceptually optimized schemes for ordinal, nominal, and quantitative data. After selecting a color scheme, ColorBrewer renders a preset cut-out of a geographic map using



```
      "Horsepower"
    ]
  },
  "data": {
    "name": "node174843",
    "url":
"https://vega.github.io
/editor/data/cars.json"
  },
  "transform": []
```

Figure 5: *Screenshot of the toolchain-enabled ColorBrewer: Visualization specifications are pasted into the textarea in the left sidebar (1). Configurations to the color mapping are made using the widgets below (2), the visualization is rendered in the main view on the right (3).*

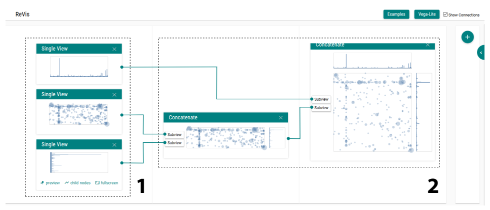


Figure 6: *Screenshot of ReVisualize, a visualization authoring tool built on ReVize: The view hierarchy of an imported declarative specification is shown as atomic (1) and composite (2) views that allow exporting partial views into visualization specifications.*

the colors of that scheme. It is however not possible to directly represent a color scheme on another visualization.

Using ReVize, we have added the option to import Vega-Lite specifications into ColorBrewer to directly apply color encodings to custom visualizations. For this reason, we have extended the graphical interface of ColorBrewer with basic widgets, while retaining most of the layout of the original tool. After pasting a specification in a text area widget, the bottom of the interface allows for picking a view from the specification, a field of the data and visual variable to which the color is to be mapped (“fill”, “stroke”, and “color”). Then, when selecting a color scheme, the visualization is rendered accordingly in the panel on the right side of the screen and the visualization specification in the text area is updated.

A screenshot of the user interface of the modified ColorBrewer is depicted in Figure 5. Our toolchain-enabled fork of the ColorBrewer project is published on Github `https://github.com/vis-au/colorbrewer`.

5.3. VisFlow

VisFlow [YS17] is a dataflow editor that allows editing a dataflow through multiple coordinated views. Selecting visual marks through brushing in one view of the dataflow serves as a filter, propagating only the contained data items to downstream nodes. While VisFlow offers a variety of standard visualization techniques, it is not possible to use customized techniques and editing the created dataflow (or visualizations) outside of VisFlow is not possible.

Using ReVize, we have added a new node type to VisFlow, which

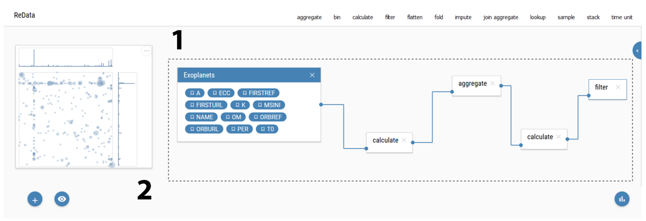


Figure 7: *Screenshot of ReData, a dataflow tool built on ReVize: The dataflow of the specification is displayed as nodes in a graph (2), changes are immediately reflected in the preview (1).*

allows importing visualization specifications into the dataflow editor. In the panel on the right side of the workbench, visualization specifications can be pasted into a text field. Underneath the text field, encodings between a visual variable and a field of the data can be made using a drop-down menu, which is populated with all fields of the incoming data stream.

The new node behaves similar to other visualization nodes in VisFlow, in that it can filter the incoming dataflow if the visualization specification defines a brush selection. The rendered visualization is updated whenever the incoming dataflow changes. In the workbench, the new node offers a preview of the imported visualization, which updates whenever changes are made to the encoding or the input dataflow. After importing, the configuration panel presents the view hierarchy from the specification, allowing us to specify a mapping from data to any visual variable. Similar to other nodes in VisFlow, any visualization allows for defining a data selection on the visual marks through a brush interaction, which is then propagated to downstream nodes.

If no dataflow is connected to the new node type, it propagates the data that is described in the visualization specification. Thus, the new node can serve as both data source and a data viewer.

A screenshot of a dataflow using the toolchain node in VisFlow is depicted in Figure 8. Our toolchain-enabled fork of the VisFlow project is published on Github `https://github.com/vis-au/visflow`.

5.4. Using ReVize as Framework for new Visualization Tools

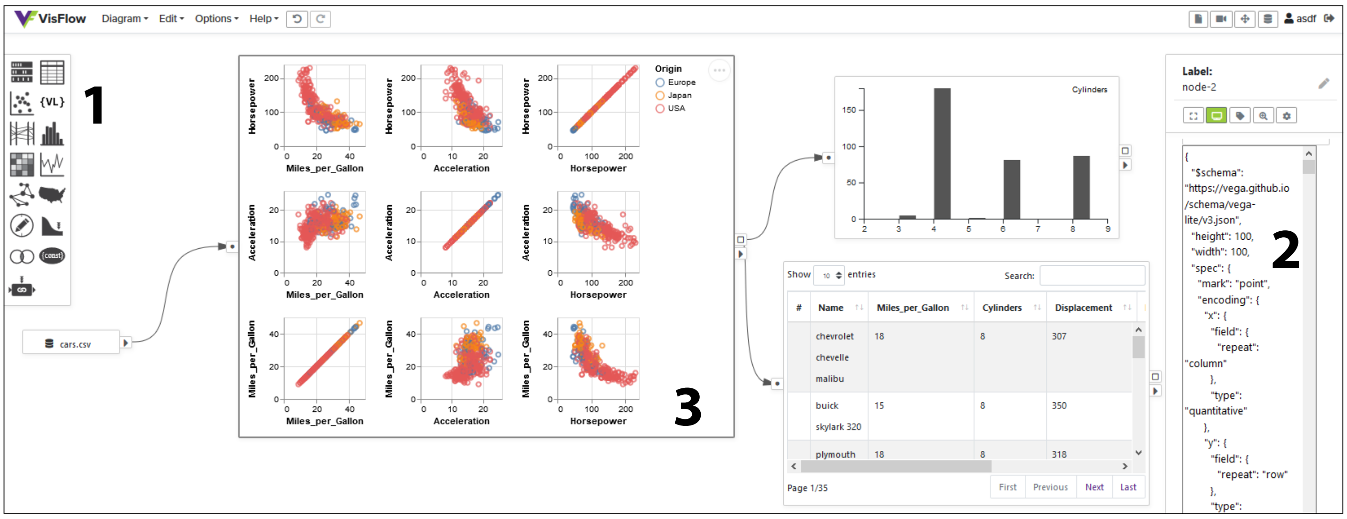
In addition to extending visualization tools with toolchaining, we have built two novel visualization tools that make toolchaining a central design aspect: The view composition tool ReVisualize and the data transformation editor ReData. ReVisualize allows importing a visualization defined in Vega-Lite to edit the individual subviews present in the specification. These sub-views can also be individually exported as Vega-Lite specifications again or reassembled in new view compositions.

ReData renders the data transformation pipelines found in an imported Vega-Lite specification for parametrization. Operators can be added or removed from the transformation. A view of the visualization using this pipeline is updated alongside this modification and can be exported into Vega-Lite, once a satisfying state is reached.

Screenshots of the user interfaces are depicted in Figure 6 and Figure 7. Both tools are published under open-source licenses on Github (see `https://github.com/vis-au/revis` and `https://github.com/vis-au/redata`).

6. Case Study: Authoring a Visualization with a Toolchain

Reconsider the toolchain scenario presented in Section 1, in which it was not possible to use multiple visualization authoring tools to modify a shared visualization and required reiterating work even if one went back and changed something. In the previous section, we described toolchain support for three of the tools mentioned in the scenario. Repeating the same workflow with the extended tools, the



```
{
  "$schema":
"https://vega.github.io
/schema/vega-
lite/v3.json",
  "height": 100,
  "width": 100,
  "spec": {
    "mark": "point",
    "encoding": {
      "x": {
        "field": {
          "repeat":
"column"
        },
        "type":
"quantitative"
      },
      "y": {
        "field": {
          "repeat": "row"
        },
        "type":
```

Show 10 entries Search:

#	Name	Miles_per_Gallon	Cylinders	Displacement
	chevrolet chevelle malibu	18	8	307
	buick skylark 320	15	8	350
	plymouth	18	8	318

Page 1/35 First Previous Next Last

Figure 8: *Screenshot of the toolchain-enabled VisFlow workbench: Using the new node from the toolbar on the left (1), any visualization specification can be pasted into the text field in the configuration panel on the right (2). Visualizations rendered from a specification that define a selection in the specification can filter the dataflow similar to other VisFlow nodes (3).*

benefits of visualization toolchains based on a declarative grammar become evident.

Since all tools can import the same visualization format, there is no longer an inherent order in which tools need to be executed. Therefore, one can easily modify the dataflow of the resulting image in KNIME, maintaining the remaining aspects of the visualization pipeline. Our approach thus solves the iteration problem [BDFM17] for those tools.

At the same time, we can incorporate authoring tools to modify the visualization specification – for example, using the Vega text editor [Lab19]. This allows users to utilize different user interfaces and feature sets when authoring the same visualization rather than forcing them to make a tradeoff by choosing a single tool. In the future, we plan to add toolchaining to other visualization authoring tools to leverage the library's potential (see Section 8).

7. Discussion

In this section, we reflect on the use of a declarative visualization grammar as exchange format for visualization toolchains and present lessons learned from using ReVize in our case study.

Visualization toolchaining based on a declarative visualization grammar distinguishes between two roles: The passively exchanged description of a visualization pipeline and the tools that actively modify it. This resembles design philosophies in other fields in computer science, such as tools and materials in software design [RZ95], instrumental interaction in user experience design [BL00], or standard input and output pipes in Unix. Similar to these philosophies, visualization tools in a toolchain could become multi-purpose instruments, rather than being restricted to a specific workflow. As a result, our approach promotes personalized visualization toolchains consisting of these individual instruments.

There are also drawbacks to using a visualization grammar for visualization toolchains. For instance, the visualization grammar used as exchange format influences which visualization features are easier to adapt and which are more complex. Vega-Lite for example allows directly including data transformation operations in the specification, whereas Brunel can reference complex visualization techniques such as the treemap as a single keyword. Expressing these features in respective other tools is difficult. A more expressive grammar such as Reactive Vega allows for creating customized visualizations and tweak every aspect of the view, but at the same time providing an interface to the grammar is naturally more involved. Thus, the applicability of visualization toolchains based on a common exchange format heavily depends on which visualization grammar is used.

Another drawback of using a standard exchange format across toolchains is that this standard might be interpreted differently by different tools. Tools may for example implement only the most common parts of a standard or implement different versions of the same standard. ReVize functions as an adapter and thus reduces some of these issues by providing an interface to one feature set of one specific version of the Vega-Lite grammar. But that grammar is actively developed and may be updated in the future to new versions that introduce breaking changes. This is a general problem when an exchange format is implemented by different developers.

While visualization grammars model the visualization that is modified by a tool, they do not model meta-information about the creation of the model, such as provenance information. In contrast to the grammar-based approach, proprietary file formats that are accessed by the same tool type [YS17, RLB19] can represent such specific meta information.

As discussed in Section 4, we were able to follow a default procedure of adding toolchain support to existing visualization tools. We benefited from the fact that these days many visualization tools are based on web technologies and their source code is often published. This made the procedure similar in tools that were built in standard JavaScript (ColorBrewer), or that were built in Typescript using Vue (VisFlow). In case of KNIME, we benefited from the generic JavaScript node's freedom of running arbitrary JavaScript code. This indicates that ReVize's interface is generalizable to fit different modern environments.

A challenge we faced during the implementation is that Vega-Lite, while generally a concise grammar, in some cases offers multiple ways of expressing a constraint for a visualization. As an example, consider the "data" property, which in the grammar can either reference remote values or provide these values inline. The "data" property can however also reference a named entry in the top-level "dataset" property. This complexity is often overlooked at first when one uses Vega-Lite, but underlines the usefulness of a library like ReVize that resolves these ambiguities transparently and without the user having to worry about them.

Our approach of using a declarative visualization grammar as exchange model between visualization tools is similar to Bigelow et al.'s bridge model, in that the exchange format captures every part of the visualization pipeline, while each tool in the toolchain only accesses those parts that it contributes to. However, since in our approach a model of the visualization pipeline is the exchanged artifact, it is not necessary to adapt the core model for every tool that is used in a toolchain. Nevertheless, this assumes that the grammar used as an exchange format can fully describe every aspect of the visualization pipeline. At the same time by exchanging a declarative model, every tool along the pipeline can render an image of the resulting visualization, providing immediate feedback to user changes from preprocessing to final color adjustments.

One benefit from using a visualization grammar as toolchain exchange format that we did not initially consider is that it supports diff-based versioning tools such as git. Since the visualization is expressed as descriptive text, changes made to the specification are detected by diff tools and are at the same time human-readable. In collaborative settings, one can therefore interpret what other users have added to the shared visualization.

8. Open Challenges for Visualization Toolchains

In Section 2, we have presented two interaction types to using multiple visualization tools in concert to author a visualization. While our current implementation of toolchains describes the consecutive scenario, future work could revolve around how tools can be used in parallel. ReVize does not prevent this use case; parallel workflows merely require a mediating scheduler that orchestrates access to the shared visualization model. Different frameworks and technologies could be used for such mediation – e.g., Webstrates [KEB*15], which in combination with ReVize would even lend itself to facilitate distributed, collaborative visualization authoring.

A central server component would also enable us to incorporate a shared versioning mechanism that would provide provenance over the creation of a visualization. While visualization versions are not specifically captured in the Vega-Lite grammar (even though there are potential workarounds through the "description" property), they could nevertheless be shared with tools as metadata along with the visualization model. This versioning could make use of the mentioned ability to use line-based diffs between Vega-Lite specifications.

Another issue with the consecutive scenario lies in the additional workload placed on the user. First, it requires to manually copy and paste a visualization description between tools. Additionally, the user is required to keep track of which tool currently holds the latest version of a visualization. Both could be resolved through a centralized distribution approach.

While our use case demonstrates the versatility of visualization toolchains based on declarative grammars, more visualization authoring tools need to be extended with ReVize to showcase the full potential of the approach. Future implementations could include one of the different open-source web-based visualization tools available, for example RAWGraphs [MEC*17], Charticulator [RLB19] or Lyra [SH14].

Parallel toolchains naturally present new research questions for collaborative scenarios. The challenge will lie in allowing users from different professional backgrounds to modify each other's work at any point and with any toolchain-enabled visualization tool they happen to be comfortable with.

Other open challenges revolve around expressive limitations of Vega-Lite. In order to overcome the discussed restrictions of the grammar, future expansions on visualization toolchains could revolve around using Vega [SRHH16] as an exchange format. While providing more customizability in details of a visualization, Vega-Lite tools could still integrate into the toolchain, due to unidirectional compilation from Vega-Lite to Vega specifications.

9. Conclusion

We have introduced the concept of visualization toolchains, the combined use of multiple tools in concert to author one visualization through a declarative visualization grammar as a shared model. Based on a literature review of the state of the art in consecutive and parallel use of different tools, we have presented a set of functional requirements for a library that allows importing and exporting a declarative visualization grammar. As an implementation of our concept, we have presented ReVize, a library for extending web-based visualization tools with toolchain support through Vega-Lite. To demonstrate the applicability of ReVize, we have integrated it into three well-known visualization tools: KNIME, ColorBrewer, and VisFlow. In a case study, we have demonstrated the usefulness of visualization toolchains and discussed limitations of our approach.

Acknowledgements

The authors would like to thank Jo Vermeulen, Henrik Korsgaard, and the anonymous reviewers for their valuable input to this paper. The idea for this paper originated in part from the UniVA research project funded by the German Research Foundation (DFG).

References

[Ado19] ADOBE: Adobe illustrator, 2019. URL: https://www.adobe.com/products/illustrator.html. 2

[BDFM17] Bigelow A., Drucker S., Fisher D., Meyer M.: Iterating between Tools to Create and Edit Visualizations. *IEEE Transactions on Visualization and Computer Graphics 23*, 1 (2017), 481–490. doi:10.1109/TVCG.2016.2598609. 2, 3, 4, 8

[BL00] Beaudouin-Lafon M.: Instrumental Interaction: An Interaction Model for Designing. In *Proc. of CHI'00* (2000), ACM, pp. 446–453. doi:10.1145/332040.332473. 8

[BOH11] Bostock M., Ogievetsky V., Heer J.: D^3 Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics 17*, 12 (2011), 2301–2309. doi:10.1109/TVCG.2011.185. 2

[CH18] Conlen M., Heer J.: Idyll: A Markup Language for Authoring and Publishing Interactive Articles on the Web. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology* (2018), ACM, pp. 977–989. doi:10.1145/3242587.3242600. 2

[CJ10] Chen M., Jänicke H.: An information-theoretic framework for visualization. *IEEE Transactions on Visualization and Computer Graphics 16*, 6 (2010), 1206–1215. doi:10.1109/TVCG.2010.132. 3

[CMS99] Card S. K., Mackinlay J. D., Shneiderman B.: *Readings in Information Visualization: Using Vision To Think*. Academic Press, 1999. 1, 2, 3

[CR98] Chi E. H.-H., Riedl J. T.: Operator interaction framework for visualization systems. In *Proceedings of the IEEE Symposium on Information Visualization* (1998), pp. 63–70. doi:10.1109/INFVIS.1998.729560. 2

[DTK*09] Dill F., Thiel K., Kötter T., Gabriel T. R., Cebron N., Meinl T., Ohl P., Berthold M. R., Wiswedel B.: KNIME - the Konstanz information miner. *ACM SIGKDD Explorations Newsletter 11*, 1 (2009), 26. doi:10.1145/1656274.1656280. 6

[FHBW11] Fekete J. D., Hémery P. L., Baudel T., Wood J.: Obvious: A meta-toolkit to encapsulate information visualization toolkits - One toolkit to bind them all. In *VAST 2011 - IEEE Conference on Visual Analytics Science and Technology 2011, Proceedings* (2011), pp. 91–100. doi:10.1109/VAST.2011.6102446. 3

[GAEk*16] Gürdür D., Asplund F., El-khoury J., Loiret F., Törngren M.: Visual Analytics Towards Tool Interoperability – A Position Paper. In *Proceedings of the 11th Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - Volume 2: IVAPP, (VISIGRAPP 2016)* (2016), SciTePress, pp. 139–145. doi:10.5220/0005751401390145. 4

[GBTS13] Grammel L., Bennett C., Tory M., Storey M.-A. D.: A Survey of Visualization Construction User Interfaces. In *EuroVis'13 Short Paper Proceedings* (2013), pp. 1–5. doi:10.2312/PE.EuroVisShort.EuroVisShort2013.019-023. 1

[HB03] Harrower M., Brewer C. A.: ColorBrewer.org: An online tool for selecting colour schemes for maps. *Cartographic Journal 40*, 1 (2003), 27–37. doi:10.1179/000870403235002042. 1, 6

[IBM19] IBM: Brunel visualization, Sept. 2019. URL: https://dataplatform.cloud.ibm.com/docs/content/wsj/analyze-data/brunel-visualization.html. 3

[JE12] Javed W., Elmqvist N.: Exploring the design space of composite visualization. *IEEE Pacific Visualization Symposium 2012, PacificVis 2012 - Proceedings* (2012), 1–8. doi:10.1109/PacificVis.2012.6183556. 2

[Jup19] Jupyter P.: Jupyter lab, 2019. URL: https://github.com/jupyterlab/jupyterlab. 2

[KEB*15] Klokmose C. N., Eagan J. R., Baader S., Mackay W., Beaudouin-Lafon M.: Webstrates: Shareable dynamic media. In *Proc. of the ACM Symposium on User Interface Software and Technology* (2015), ACM, pp. 280–290. doi:10.1145/2807442.2807446. 9

[Kir19] Kirk A.: Resources, Sept. 2019. URL: https://www.visualisingdata.com/resources/. 1

[KRkP*16] Kluyver T., Ragan-kelley B., Pérez F., Granger B., Bussonnier M., Frederic J., Kelley K., Hamrick J., Grout J., Corlay S., Ivanov P., Avila D., Abdalla S., Willing C.: Jupyter Notebooks – a publishing format for reproducible computational workflows. *Positioning and Power in Academic Publishing: Players, Agents and Agendas* (2016), 87–90. doi:10.3233/978-1-61499-649-1-87. 1

[Lab19] Lab U. I. D.: Vega editor, 2019. URL: https://vega.github.io/editor. 8

[MEC*17] Mauri M., Elli T., Caviglia G., Uboldi G., Azzi M.: RAWGraphs: A Visualisation Platform to Create Open Outputs. In *Proceedings of the 12th Biannual Conference on Italian SIGCHI Chapter* (2017), pp. 28:1–28:5. doi:10.1145/3125571.3125585. 2, 9

[MMWC18] Mei H., Ma Y., Wei Y., Chen W.: The design space of construction tools for information visualization: A survey. *Journal of Visual Languages and Computing 44* (2018), 120–132. doi:10.1016/j.jvlc.2017.10.001. 1

[MNV16] Méndez G. G., Nacenta M. A., Vandenheste S.: iVoLVER: Interactive Visual Language for Visualization Extraction and Reconstruction. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2016), ACM, pp. 4073–4085. doi:10.1145/2858036.2858435. 2

[NS07] North C., Shneiderman B.: Snap-Together Visualization: A User Interface for Coordinating Visualizations via Relational Schemata. *The Craft of Information Visualization* (2007), 341–348. doi:10.1016/b978-155860915-0/50043-3. 2

[Obs19] Observable I.: Observable, 2019. URL: https://observablehq.com/. 2

[Poj19] Poject T. I.: Inkscape, 2019. URL: https://inkscape.org/. 2

[RLB19] Ren D., Lee B., Brehmer M.: Charticulator: Interactive Construction of Bespoke Chart Layouts. *IEEE Transactions on Visualization and Computer Graphics 25*, 1 (2019), 789–799. doi:10.1109/TVCG.2018.2865158. 2, 8, 9

[RM11] Rogowitz B. E., Matasci N.: Metadata Mapper: a web service for mapping data between independent visual analysis components, guided by perceptual rules. *Human Vision and Electronic Imaging XVI 7865* (2011), 78650I. doi:10.1117/12.881734. 3

[Rob07] Roberts J. C.: State of the art: Coordinated multiple views in exploratory visualization. In *Proc. of the Conf. on Coordinated and Multiple Views in Exploratory Visualization (CMV 2007)* (2007), pp. 61–71. doi:10.1109/CMV.2007.20. 2

[RSD*13] Rungta A., Summa B., Demir D., Bremer P. T., Pascucci V.: ManyVis: Multiple Applications in an Integrated Visualization Environment. *IEEE Transactions on Visualization and Computer Graphics 19*, 12 (2013), 2878–2885. doi:10.1109/TVCG.2013.174. 3

[RZ95] Riehle D., Züllighoven H.: A pattern language for tool construction and integration based on the tools and materials metaphor. In *Pattern languages of program design*, Coplian J. O., Schmindt D. C., (Eds.). Addison Wesley Professional, 1995, pp. 9–42. 8

[SH14] Satyanarayan A., Heer J.: Lyra: An interactive visualization design environment. *Computer Graphics Forum 33*, 3 (2014), 351–360. doi:10.1111/cgf.12391. 1, 2, 9

[SLR*20] Satyanarayan A., Lee B., Ren D., Heer J., Stasko J., Thompson J., Brehmer M., Liu Z.: Critical reflections on visualization authoring systems. *IEEE Transactions on Visualization and Computer Graphics 26*, 1 (2020), 461–471. doi:10.1109/TVCG.2019.2934281. 1

[SMWH17] Satyanarayan A., Moritz D., Wongsuphasawat K., Heer J.: Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics 23*, 1 (2017), 341–350. doi:10.1109/TVCG.2016.2599030. 2, 3

[SRHH16] SATYANARAYAN A., RUSSELL R., HOFFSWELL J., HEER J.: Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization. *IEEE Transactions on Visualization and Computer Graphics 22*, 1 (2016), 659–668. doi:10.1109/TVCG.2015.2467091. 2, 3, 9

[SRN*19] SCHULZ H.-J., RÖHLIG M., NONNEMANN L., AEHNELT M., DIENER H., URBAN B., SCHUMANN H.: Lightweight Coordination of Multiple Independent Visual Analytics Tools. In *Proceedings of the 14th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - Volume 3: IVAPP* (2019), SciTePress, pp. 106–117. doi:10.5220/0007571101060117. 3

[Tea19] TEAM T. G.: Gimp, 2019. URL: https://www.gimp.org/. 2

[TIC09] TOBIASZ M., ISENBERG P., CARPENDALE S.: Lark: Coordinating co-located collaboration with information visualization. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 1065–1072. doi:10.1109/TVCG.2009.162. 2

[VGH*18] VANDERPLAS J., GRANGER B., HEER J., MORITZ D., WONGSUPHASAWAT K., SATYANARAYAN A., LEES E., TIMOFEEV I., WELSH B., SIEVERT S.: Altair: Interactive Statistical Visualizations for Python. *Journal of Open Source Software 3*, 32 (2018), 1057. doi:10.21105/joss.01057. 4

[Wea04] WEAVER C.: Building Highly-Coordinated Visualizations in Improvise. *IEEE Symposium on Information Visualization* (2004), 159–166. doi:10.1109/INFVIS.2004.12. 2

[Wic10] WICKHAM H.: A Layered grammar of graphics. *Journal of Computational and Graphical Statistics 19*, 1 (2010), 3–28. doi:10.1198/jcgs.2009.07098. 3

[Wil05] WILKINSON L.: *The Grammar of Graphics*, 2nd ed. Springer, 2005. doi:10.1007/0-387-28695-0. 3

[WQM*17] WONGSUPHASAWAT K., QU Z., MORITZ D., CHANG R., OUK F., ANAND A., MACKINLAY J., HOWE B., HEER J.: Voyager 2: Augmenting Visual Analysis with Partial View Specifications. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems - CHI '17* (2017), pp. 2648–2659. doi:10.1145/3025453.3025768. 1

[YS17] YU B., SILVA C. T.: VisFlow - Web-based Visualization Framework for Tabular Data with a Subset Flow Model. *IEEE Transactions on Visualization and Computer Graphics 23*, 1 (2017), 251–260. doi:10.1109/TVCG.2016.2598497. 2, 7, 8