StructGraphics: Flexible Visualization Design through Data-Agnostic and Reusable Graphical Structures

Theophanis Tsandilas

StructGraphics: Flexible Visualization Design through Data-Agnostic and Reusable Graphical Structures

Theophanis Tsandilas

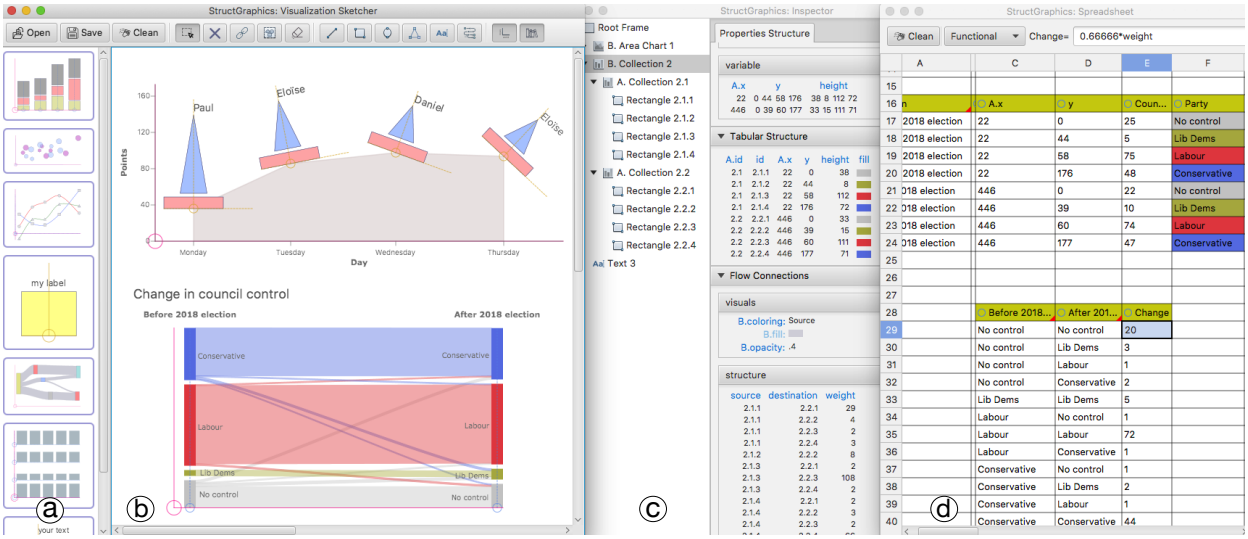


Fig. 1. The StructGrapher's user interface consists of: (a) a library of reusable visualization structures, (b) a visualization sketcher for drawing basic shapes and grouping them into collections, (c) a property inspector for exposing and structuring graphical properties, and (d) a spreadsheet for creating mappings between properties and data. The approach enables designers to construct data-agnostic visualizations and create flexible bindings with data. The bottom visualization design was originally published in the Financial Times [22]. See our gallery at: `https://www.lri.fr/~fanis/StructGraphics`

Abstract—Information visualization research has developed powerful systems that enable users to author custom data visualizations without textual programming. These systems can support graphics-driven practices by bridging lazy data-binding mechanisms with vector-graphics editing tools. Yet, despite their expressive power, visualization authoring systems often assume that users want to generate visual representations that they already have in mind rather than explore designs. They also impose a data-to-graphics workflow, where binding data dimensions to graphical properties is a necessary step for generating visualization layouts. In this paper, we introduce *StructGraphics*, an approach for creating data-agnostic and fully reusable visualization designs. StructGraphics enables designers to construct visualization designs by drawing graphics on a canvas and then structuring their visual properties without relying on a concrete dataset or data schema. In StructGraphics, tabular data structures are derived directly from the structure of the graphics. Later, designers can link these structures with real datasets through a spreadsheet user interface. StructGraphics supports the design and reuse of complex data visualizations by combining graphical property sharing, by-example design specification, and persistent layout constraints. We demonstrate the power of the approach through a gallery of visualization examples and reflect on its strengths and limitations in interaction with graphic designers and data visualization experts.

Index Terms—Visualization design, graphical structures, visualization grammars, layout constraints, infographics, flexible data binding.

1 Introduction

Visualization systems make use of diverse data representations to help people make sense of data. However, previous research has argued that traditional visualization systems have focused on data exploration and analysis tasks, paying less attention to presentation purposes [19]. Past studies [9,51] also suggest that traditional tools do not target designers and thus often fail to meet real design practices; some tools give little design freedom, while others require users to program. Recent systems address this limitation by integrating graphic design tools into the visualization creation process. Some systems focus on how to support expressive data-driven infographics [18,53,56,59]. Others [33,40,49] help visualization authors to produce rich layouts. Each approach has its own strengths and limitations and presents different solutions on how to produce expressive visualizations without programming.

Liu et al. [21] argue that *"designers use familiar tools to draw, select and manipulate vector graphics, and apply data encoding only when it is necessary."* The above systems generally adopt a *lazy* data-binding approach to support such practices, where the design of graphics precedes the data-binding step. Even so, these systems are largely data-driven and still rely on a specific dataset to derive a visualization structure. For example, Data Illustrator [21] requires its users to assign data dimensions at the very beginning of the authoring process. Satyanarayan et al. [41] collectively reflect on the assumptions and capabilities of Lyra [40], Data Illustrator [21], and Charticulator [33]. They conclude that these systems target *authoring* rather than *design* tasks since they assume that users start their task with an *"appropriately formatted dataset"* and have a *"specific chart design in mind."*

In this paper, we target design tasks. Our goal is to help users explore visualization layouts without being constrained by data encodings. We

• *Theophanis Tsandilas is with Université Paris-Saclay, Inria & CNRS, France. E-mail: theophanis.tsandilas@inria.fr*

adopt a data-agnostic approach that produces reusable visualization structures without programming. The approach combines techniques of graphical properties sharing [15], by-example specification [27, 31], and persistent alignment and distribution [13] into a new framework of nested property structures. We call it *StructGraphics* (in distinction with "Infographics") to emphasize its focus on graphical structures rather than data. StructGraphics enables designers to draw their visualization primitives and structure the constraints of their properties without relying on any specific dataset or data schema. Rather than being predefined, data structures are generated on the fly, as designers interactively construct their visualizations. Yet, as Liu et al. [21] report, it is sometimes *"beneficial to bring in real data"* even during the shape-drawing phase. StructGraphics does not require real data but lets designers fully or partially link data to their designs at any moment. Such data bindings are flexible and not persistent, and allow designers to test a design with alternative encodings. Our data-binding mechanism is facilitated by a spreadsheet user interface that complements the manual entry of data [9] with the constraints of a visualization structure.

Overall, StructGraphics follows the inverse workflow than traditional visualization-design systems. Rather than transforming data dependencies into visualization constraints, it allows users to interactively define the property and layout constraints of their visualization designs and then translate these graphical constraints into alternative data structures. Since visualization designs are data-agnostic, they can be easily reused and combined with different datasets.

2 Related Work

StructGraphics builds upon a vast volume of research on Information Visualization, Human-Computer Interaction, and Computer Graphics.

2.1 Visualization Grammars and Toolkits

Wilkinson's [55] and Wickham's [54] seminal work on visualization grammars has led to powerful visualization creation systems such as ggplot2 [2]. Other systems, such as Polaris [45] and its commercial successor Tableau [6], provide expressive interaction tools that help users interactively visualize their data. A third stream of research has investigated programming toolkits [11, 12] and declarative grammars [42, 44] that support interactive data visualization. Finally, hybrid approaches such as VisComposer [24] combine rapid prototyping through interactive controls with visual or textual programming.

A number of authors [21, 33, 50, 56] have pointed out that such systems impose a *bottom-up* workflow, where design is driven by data rather than by graphics, layout, and aesthetics.

2.2 Studies of Design Practices

A number of past studies have investigated how professionals or novices design visual representations or work with graphics. Bigelow et al. [9] studied how designers approach data and how data affect their visualization designs. They observed that designers prefer a *"flexible design environment that does not enforce a specific order of operations"* and create visualizations in a *"top-down, graphical process."* These observations are corroborated by Walny et al. [51] who report that designers who tried to think in terms of *"concepts extrinsic to the dataset"* ended up with deeper observations about the data. Méndez et al. [25] experimentally compared a bottom-up approach (Tableau [6]) to a top-down approach (iVoLVER [26]) with 10 non-expert participants. Their results show that although a top-down approach is more laborious and slower, it can help users get a better understanding of the creation process, feel more in control, and produce more varied designs.

Others have looked on how designers work with graphical structures. In particular, Maudet et al. [23] point out that the creation of layout structures is an essential part of the graphics design process but also observe that many designers struggle when having to go beyond grid-based layouts. The authors argue that graphical properties must be described instead through versatile rules or constraints that they call "*graphical substrates.*" How to describe the property constraints of a visualization design is a key challenge that we address in this paper.

2.3 Design-Oriented Visualization Authoring Tools

Several systems aim to bring visualization tools closer to the graphics design process. Roth et al. [35] are probably the first to describe tools for binding data to the properties of graphical objects. Years later, Bret Victor [49] demonstrates an interactive data-driven system for creating custom data visualizations through a loop-based workflow that resembles programming. Research systems that combine graphics-driven visualization authoring with programming workflows (either textual or visual) include iVoLVER [26], Hanpuku [10], and d3-gridding [50].

Meanwhile, the information visualization research has introduced a range of systems that support expressive visualization design without programming: Lyra [40], iVisDesigner [32], Data-Driven Guides [18], InfoNice [53], DataInk [56], Data Illustrator [21], Charticulator [33], and DataQuilt [59] are representative examples. Satyanarayan et al. [41] describe these systems as *visual builders*. Some visual builders [18, 53, 56, 59] focus on how to help designers produce creative infographics. For example, Data-Driven Guides [18] combines a vector graphics editor with a flexible data-binding mechanism to produce data-driven infographics. InfoNice [53] offers similar functionality but targets users who are not design experts.

A second group of visual builders [21, 33, 40] focus on how to author visualizations with complex (e.g., layered or nested) visualization layouts. In particular, Data Illustrator [21] provides tools for grouping graphical primitives, defining nested collections of similar objects, applying graphical layouts, and establishing lazy data bindings that act as constraints. Charticulator [33], instead, uses a constraint-based layout specification framework that can infer complex visualization layouts from partial user-driven specifications.

Despite their power, these systems still require users to bind data to graphics at the very first steps of the design process. Satyanarayan et al. [41] acknowledge that these systems address the *"the narrower activity of visualization authoring, where the author already has a desired visualization in mind, and has a dataset in the appropriate format."* StructGraphics' approach is fully graphics-driven. Its goal is to encourage users to explore designs by directly interacting with a visualization's graphics, before dealing with concrete data.

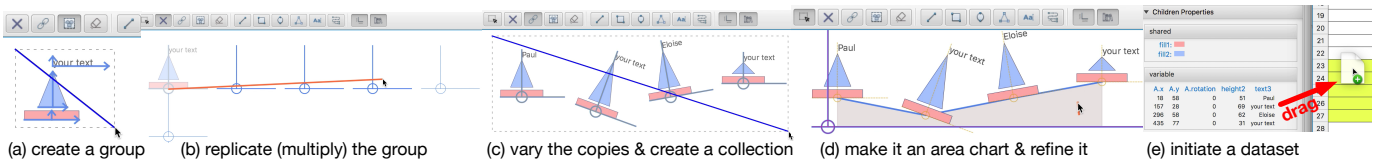
2.4 Sketch-Based Data Visualizations

Other systems focus on how to support visualization analysis and presentation tasks through expressive free-from sketches. For example, Sketchstory [20] allows analysts to communicate results through informal sketch-based representations. SketchSliders [47] support mobile data exploration through interactive sketch-based controllers. Datatoon [17] allows users to produce "data comics" by sketching informal data presentations based on directed node-link diagrams. Finally, DataInk [56] supports expressive shape primitives and free-form layout patterns but does not handle nested layouts. How to support sketch-based visualizations is beyond the scope of our work.

2.5 Visualization by Example

Gold [27] was probably the first system to support visualization creation by demonstration. Gold used various heuristics to interpret example graphics but supported a limited number of charts and required a well-formatted dataset. Visualization research has also looked at how to infer visual transformations from user-driven interactive demonstrations [37, 38]. The approach supports visualization transformation tasks, rather than visualization authoring tasks, and is largely data-driven. Wang et al. [52] have recently introduced a visualization-by-example technique that infers a visualization grammar from a partial visual sketch and a dataset. Unfortunately, the technique requires the pre-specification of inference rules for each visualization type (e.g., scatter plot, bar chart, and line chart) and further assumes that the input dataset is well formatted and labeled in accordance to the visualization example.

In StructGraphics, an example defines the bindings of graphical objects at one level each time, and from those, the system also infers the structure of data. Other approaches use Convolutional Neural Networks to extract the data [16] or the grammar [30] from a visualization image, but their success depends on the size and quality of a training dataset.



(a) create a group (b) replicate (multiply) the group (c) vary the copies & create a collection (d) make it an area chart & refine it (e) initiate a dataset

Fig. 2. Representative scenario: (a) group primitive shapes; (b) create multiple copies of the group; (c) vary their properties to produce an example for a new collection; (d) customize the collection; and (e) drag its variable properties from the inspector to the spreadsheet to create a data table.

2.6 Graphical Layouts and Property Constraints

HCI and Computer Graphics research has studied techniques that infer layouts constraints or aid their construction. Ryall et al. [36] describe a number of techniques that constrain the layout of graphs, including alignment, equal spacing (i.e., distribution), ordering, and grouping. StickyLines [13] take this approach further by making alignment and distribution constraints tweakable and persistent. Other approaches try to automatically infer layout constraints [57] or specify layout designs by example [31]. Also, Hoarau and Conversy [15] present a drawing tool that enables users to interactively create dependencies between graphical properties and inspect multiple instances of a property at the same time. StructGraphics borrows ideas from all this line of work and applies them to nested visualization designs.

Finally, some intelligent systems adaptively optimize a layout design [28,46] or provide suggestions [29] to help users produce better layouts. StructGraphics does not target layout optimization. Its grouping tools rely on simple rules for inferring the intended structure of visual properties with coherent layout constraints.

3 Design Principles

StructGraphics targets design-driven scenarios, where a data visualization is defined progressively as the user designs and interacts with its graphics. Bigelow et al. [9] report that designers *"try to understand the overall appearance of a visualization before plotting real data on axes"* and *"tend to think of influencing existing graphics that they have already created."* Thus, our goal is to help designers explore the graphical structure of a visualization before committing to specific bindings with data. Moreover, we want to encourage users to reuse their previous designs. Overall, StructGraphics is based on four design principles.

DP_1: Detach the graphical structure from the data. Despite their flexible designs, visual builders like Lyra [40], Data Illustrator [21], and Charticulator [33] require users to bind data early in the authoring process. As a result, visualization structures are largely determined by the data. In StructGraphics, the graphical structure of a visualization is constructed in an interactive way and independently from the dataset. The approach relies on a *property sharing* mechanism that allows users to define how the graphical properties of a visualization (e.g, the x and y positions of a scatterplot's points) are shared or vary across dimensions.

DP_2: Enable constrained but flexible layout designs. Grids, stacks, and packs are representative layout constraints in visualization authoring systems [41]. StructGraphics' layout constraints are based, instead, on principles of persistent alignment and distribution [13]. This approach allows for more malleable layouts, e.g., a grid with rows that are freely translated across the x direction.

DP_3: Encourage reusability at all levels of a graphical structure. Other systems support reusable visualization templates that can be exported as Vega specifications [40] or Microsoft Power BI custom visual files [33]. Such templates can be reused with different datasets but constrain the design process. For example, Charticulator [33] templates are hard-coded to the original dataset schema and *"any new data must be structured in an identical format"* [41]. In StructGraphics, visualizations are expressed as hierarchies of data-agnostic graphical structures, and every node in the hierarchy can be easily added to a library and later reused with simple drag-and-drop actions.

DP_4: Support late binding with flexible data schemas. Transforming a dataset into a convenient format requires the user to perform a series of data-formatting operations, such as filtering, transformation, restructuring, and aggregation. Not only are these steps laborious, but they also require careful preplanning. Unfortunately, when exploring design alternatives, designers may not have a clear idea about which data format is the most appropriate for their task. StructGraphics delays all data formatting decisions to the very end of the process, when the visualization structure has already been constructed. In StructGraphics, the data schema is defined by the structure of the graphics, rather than the opposite. We facilitate the conversion of graphical structures to data with a spreadsheet user interface. The spreadsheet serves as a playground for exploring alternative data encodings.

4 StructGraphics

We explain our approach with variations of the two visualization examples in Fig. 1. The top visualization is a custom infographic that presents the performance of a team of children across different dimensions. Fig. 2 summarizes the main steps that a user can follow to create it. The bottom visualization reproduces a Sankey diagram published in the Financial Times [22]. It depicts the change in council control after the UK local elections in 2018.

4.1 Overview

The StructGraphics user interface is composed of three fully synchronized components (see Fig. 1):

Visualization Sketcher. The design of a visualization starts by drawing basic shapes with a vector graphics editor, the visualization sketcher. The sketcher provides tools for repeating shapes, varying their geometries, and grouping them together into *groups* of elementary shapes and *collections* (see Fig. 2). Through *repetition*, *variation*, and *grouping*, designers can create rich hierarchies of visualization collections.

The sketcher relies on techniques of persistent alignment and distribution [13] for laying out visual objects (DP_2). This approach encourages exploration through trial and error. We further use *by-example design specification* [31] to automate the extraction of layout constraints and further infer how graphical properties are shared or vary. For instance, the example in Fig. 2a indicates that the x-coordinate is shared among all three shapes to be grouped. In this case, StructGraphics will create a binding to make their x-alignement persistent. Based on this mechanism, StructGrapher constructs structures of graphical properties that define a visualization design without data (DP_1). At any moment, users can select a group or collection in their visualization structure and add it to a library (see Fig. 1a) for future reuse (DP_3).

Property Inspector. The inspector exposes the graphical properties of a visualization and organizes them under a tree hierarchy. Inspired by the *property sheet* of Hoarau and Conversy [15], the inspector displays the full range of property values of a group of visualization objects and reveals their sharing constraints. We further extend this approach to nested graphical structures. Specifically, we distinguish between *shared* and *variable* properties at all levels of a visualization hierarchy. For example, the fill color of both the hull and the sail of the boats in the collection of Fig. 2d is shared, whereas the x and y coordinates and the rotation of the boats are variable. Variable graphical properties are automatically organized into wide and narrow tabular forms that serve as data templates (DP_4).

The inspector allows users to interactively restructure properties to correct their designs, or explore different variations of constraints. As opposed to other visualization authoring tools [21,32,33,40], the StructGraphics inspector is not directly associated with a dataset and is not directly used for data-binding purposes (DP_1).

Spreadsheet. Users can create mappings between graphical properties (either individual or grouped) and data at any moment, or at the very end of the process, when they need to populate real data values and

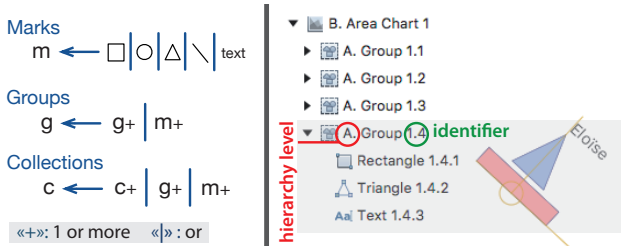


Fig. 3. StructGraphic visualization elements: marks, groups, and collections. Left: Their grammar. Right: An example of a visualization tree. We highlight a group that contains a rectangle, a triangle, and a text field.

generate axes, labels, and legends (DP_4). All data manipulations take place in a spreadsheet. Spreadsheets are extremely popular and have a key role in the workflow of many design experts [9]. According to Dix et al. [14], spreadsheets *"are often the best tools to use."*

In the StructGraphics user interface, designers can create custom data templates (DP_4) by dragging individual properties or tabular property structures from the inspector to a spreadsheet (see Fig 2e). Users can type a name in the spreadsheet to define a data variable and then apply a numerical transformation, or map it to a set of categorical values. They can also manually edit individual values or import them from external spreadsheet applications, such as Microsoft Excel.

Property constraints created in the sketcher and the inspector are translated into value dependencies in the spreadsheet. The spreadsheet user interface also allows designers to choose which variables to visualize as axes, labels, or legends. Rather than imposing a unique data schema, designers can explore alternative scale transformations and reuse a visualization design with different pieces of data (DP_3).

4.2 Visualization Structure

A StructGraphics visualization is a tree structure, whose leaf nodes are graphical primitives, called *marks*: lines, rectangles, ellipses, and text fields. As in Data Illustrator [21], marks can be grouped together to form *collections* and *groups*. Collections and groups can be nested under higher-level collections and groups. Fig. 3 (left) describes the grammar of StructGraphics visualizations. Fig. 3 (right) shows the visualization tree of our infographic in Fig. 1, where we highlight the group of the last "boat." As all nodes in the tree, the group has a unique identifier (1.4) but is further associated with a letter ("A") that specifies its hierarchy level. As we see later, this notation helps distinguish among the graphical properties of nodes at different hierarchy levels.

Groups and collections have a similar structure but distinct roles. Groups enable designers to create complex graphics and have a fixed graphical structure. Top-level groups serve as the *glyphs* [33,41] of a visualization design. They expose the properties of individual shapes, which can vary independently from each other. For example, the fill color and height of the sail in Fig. 3 vary independently of the fill color and height of the hull. In contrast, collections group together a variable (rather than fixed) number of children nodes, where all children have the same graphical structure. The top collection of our example ("B. Area Chart 1") contains four groups, where they all depict a sailboat.

We represent groups and collections (or for nested collections) as Cartesian coordinate systems (see Fig. 1), whose axes dynamically adapt to the size and position of their children nodes. We refer to these representations as *skeletons*. Skeletons serve as interaction handles, as reference lines for the coordinates of children nodes, and as scaffolds for creating the axes of the final visualization. Users can press on the button to hide them and show them back.

4.3 Properties Framework

At the core of the StructGraphics approach is a framework for structuring the graphical properties of a visualization tree.

4.3.1 Basic Graphical Properties

A mark is described by the following graphical properties: its type (e.g., line, rectangle, ellipse, triangle, or text field), its x and y position with respect to the origin of its parent group or collection, its width and height, its rotation angle, its fill color, and its stroke thickness and color. Text fields have additional text-specific properties. Marks have also properties that define their origin location: *left*, *right*, or *center* for the x axis, and *bottom*, *top*, and *center* for the y axis. We display them as vectors (arrows) (see Fig. 4) that depict the mark's orientation, where upwards and rightwards are positive directions. All properties have a name, a data type, and a value.

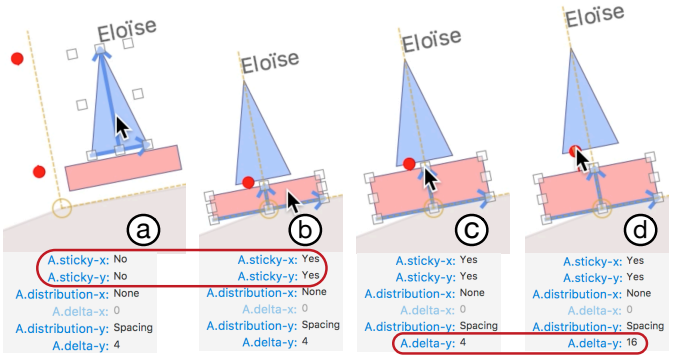


Fig. 4. Layout properties of a group. (a) The three marks in the group are equally spaced along the y axis. (b) The stickiness properties change to "Yes", so the marks are glued to the group's skeleton. (c) When changing the height of the rectangular mark, the other marks move upwards to respect the spacing distribution constraints. (d) The user manipulates the red handle to increase their in-between space.

Groups and collections have their own graphical properties. These properties define the relative position and rotation angle of the group or collection, or alternatively, the layout of its children nodes. Other more specialized collection properties allow designers to customize connected visualizations, such as line or spline charts, area charts, and flow diagrams (see Fig. 1). Group and collection properties have a prefix (e.g., *A.x* and *B.rotation*) that reflects their hierarchy level.

4.3.2 Layout Properties

Layout constraints are defined through the properties of groups and collections. Those include *distribution* and *stickiness* properties that constrain the relative positioning of their children (see Fig. 4). As Ciolfi Felice et al. [13], we support two types of distribution constraints: (i) equal spacing and (ii) equal distances. Such constraints are persistent – they are preserved as users translate or resize a node within a group or collection. Stickiness properties apply further constraints by gluing the children nodes of a group or collection on the x or y axis of its skeleton.

Combined with StructGraphics' property sharing mechanism (see below), these properties let users define grids, stacks, packs, or free layouts at any hierarchy level and later modify them as needed (DP_2).

4.3.3 Property Structures and Property Sharing

StructGraphics distinguishes between the "self properties" of a node (mark, collection, or group) and the properties of its children. Collections and top-level groups gather the properties of all nodes in their subtree and organize them into *property structures*. Property structures take the form of nested property lists, which are then merged into larger tabular structures. For instance, consider the bottom visualization (Sankey diagram) in Fig. 1. The visualization is a B-level collection that contains two A-level collections with four children (rectangular marks) each. Fig. 5a shows the children properties of the first subcollection divided into two sections: (i) a section of *shared* properties, i.e., common properties among all children nodes; and (ii) a section of *variable* properties whose values vary across marks.

Property sharing is at the core of our approach. All visual relationships in a StructGraphics visualization are defined through property sharing. Sharing puts hard constraints on a group of property values. For example, since the *width* property is shared (Fig. 5a), changing the width of a rectangle in the collection will cause the width of the other rectangles to change as well. Likewise, since the x property is shared, all the rectangles within each collection are vertically aligned. Thus,

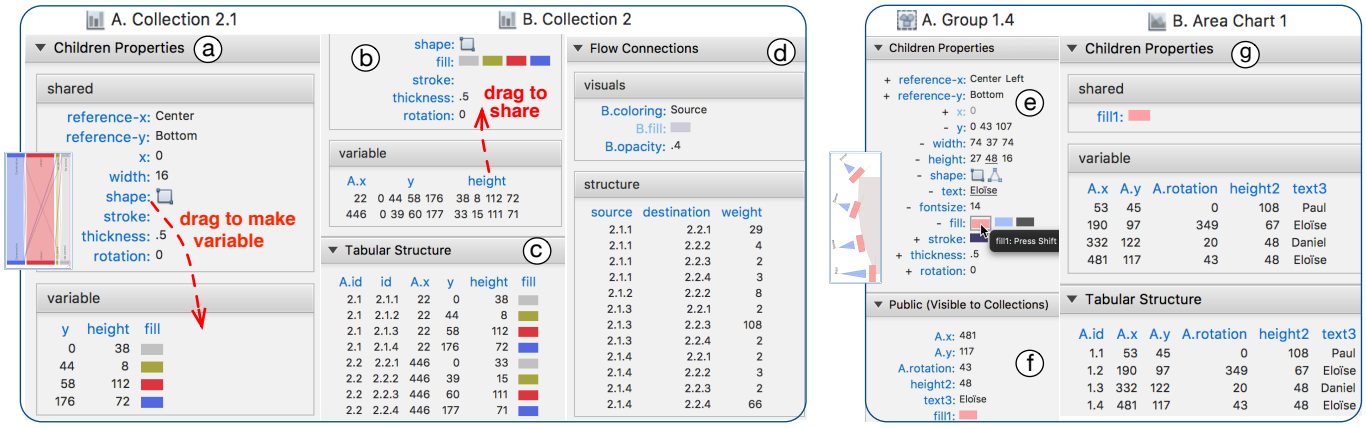


Fig. 5. Property structures of the visualizations in Fig. 1, as represented in the StructGraphics inspector. Left: (a) The children properties of the first A-level collection divided into two groups: shared and variable. (b) Part of the children properties of the B-level collection. (c) A tabular structure that summarizes all variable properties. (d) Structure of connections between nodes. Right: (e) Structure of the children properties of an A-level group. The user can choose which properties to make public. (f) Those properties are highlighted and appear in the "Public (Visible to Collections)" section. (g) The structure of children properties in the B-level collection. Observe that only public group properties are shown.

property sharing can serve as an alignment mechanism that further constrains a visualization layout (DP_2).

Fig. 5b presents part of the children properties of the B-level collection. Observe that the *fill* property is shared at this level, and the same four colors are common in both A-level collections. In contrast, the *y* and *height* properties are variable at both levels, thus they can freely change. Users can modify a property structure by dragging the label of a property from the section of variable properties to the section of shared properties, and vice-versa.

The tabular structure in Fig. 5c shows all properties that are variable at least at one level of the visualization hierarchy, where each variable property is represented by a separate column. Two additional columns in the table represent the identifiers of the A-level collections (*A.id*) and the identifiers of the marks (*id*). Such tabular structures serve as templates for generating data tables in the spreadsheet (see Sec. 4.5), where each column can be mapped to a data variable.

4.3.4 Property Grouping and Scoping

As we discussed earlier, groups can expose the properties of individual marks. For example, the designer of our infographic in Fig. 1 wants to vary the sail height but keep the height of the hull and the flag constant. StructGraphics supports this design through a fine-grained interface for structuring properties within groups. Fig. 5e shows how the inspector displays the property structure of a group. Notice that some properties (e.g., *reference-y* and *thickness*) are bound across all marks and have a common value (and name). Other properties (e.g., *fill*) are split into distinct properties and are named by the mark to which they refer (e.g., *fill1*, *fill2*, and *fill3*). Finally, some properties (e.g., *reference-x*) are only partially split for a subset of the marks in the group.

Unfortunately, the number of individual properties in a group can become large and cause the tabular structures of parent collections to explode in size. We address this problem with *property scoping*. Specifically, the inspector lets users pick which properties to make public, i.e., visible to parent collections, by hovering over a property of interest and pressing a key (see Fig. 5e-f). As Fig. 5g shows, the property structure of a parent collection includes the public only properties of its children. Users can change the visibility of group properties at any moment, and the property structure of parent collections are constantly updated.

4.3.5 Connections and Directional Links

Collections provide additional properties to describe connections among their children. Connections that StructGraphics supports include poly lines, areas, and Bézier curves. The ordering of connections follows the natural ordering of the children in a collection. Connections can be created among the siblings of any type of node, including marks, groups, and sub-collections.

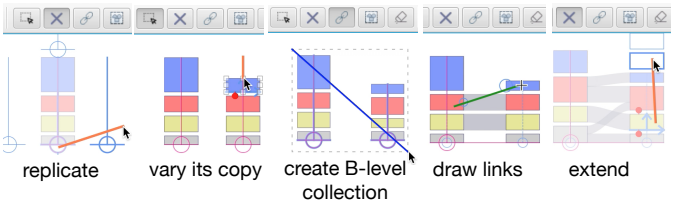


Fig. 6. Creating nested collections: Sankey diagram of Fig. 1.

Furthermore, StructGraphics allows for directional links between any two nodes within a collection. For example, the flow connections in the Sankey diagram of Fig. 1 are weighted directional links between the marks of the two nested collections. As shown in Fig. 5d, links are represented as rows in a tabular structure. Each row includes the identifier of the link's source and destination, and a weight. For this type of visualization, the weight expresses the height of a flow connection in pixels. As with other property structures, the designer can later use the table of links as a template to generate the data that define the flows.

4.4 Constructing Visualization Designs

The StructGraphics user interface provides a set of tools that enable designers to draw shapes, create groups and collections, reuse them, structure their properties, map them to data variables, and decorate visualizations with axes, labels, and legends.

4.4.1 Drawing, Replicating, Grouping, and Linking

As in traditional vector-graphics editors, users can draw shapes on the canvas and then resize them with interactive handles. The effect of all object manipulations is communicated to users with an "elastic band." The band shows the trace of an ongoing manipulation to help users trace their edits and easily cancel them, e.g., by pressing *ESC*. Our implementation resembles the Dwell-and-Spring technique [8] and serves as the base for all tools in the sketcher (see Fig. 6) by coupling direct manipulation with crossing-based selection [7].

A *replication* (multiply) tool accelerates the creation of multiple similar objects. The tool substitutes the "repeat action" of Data Illustrator [21] but does not require the user to associate the action with a data dimension (see Fig. 2b). Users can replicate individual marks, groups, collections, and full visualization trees. As shown in Fig. 6, they can also extend the internal nodes of a collection to achieve a target design. In such cases, the tool ensures that the collection's layout and property sharing constraints are respected. StructGraphics provides additional tools for grouping canvas objects (marks, groups, or collections) into groups or collections. Users can also draw direct links between marks (see Fig. 6), where connected marks must belong to the same visualization tree, i.e., appear under the same top-level collection.

As users create collections, groups, and connections, property structures are automatically updated in a way that no conflicts arise. For example, StructGraphics ensures that the *height* property in the Sankey diagram of Fig. 1 is not shared across rectangles. Likewise, the height of connected rectangles is constantly updated (e.g., the height of a source mark increases) to remain consistent with incoming and outgoing flows.

4.4.2 By-Example Design Specification

Creating the right property structure can be tedious, since the designer must specify which properties are shared or variable at each level of the visualization tree. StructGraphics' grouping tools relax this problem by trying to "guess" the designer's target structure based on the property values of the grouped objects, which serve as examples. To this end, we calculate statistics for alternative layout configuration and pick the most probable configuration. Overall, StructGraphics' by-example specification approach supports a design workflow with four main steps (see Fig. 2 and Fig. 6): (i) draw a shape, (ii) replicate it, (iii) vary the properties of one or more replicas, and (iv) then group.

In order to decide whether a property is shared or variable, we analyze its variance (in pixels) across all grouped objects. If it is below a threshold value, the property is considered as shared. Likewise, we infer reference points, alignment, and distribution constraints by looking at all possible configurations of distance and spacing between grouped nodes. We then try to identify the most common patterns. When more than one possible configurations arise, e.g., equal distances between centers but also equal spacing, we choose a configuration based on a list of priority rules. We prioritize center-based alignments and distance-based distributions on the x-axis and bottom-based alignments and spacing-based distributions on the y-axis.

4.4.3 Design Reuse

Since visualization designs are independent of raw data and data schemas, they can be easily reused (DP_3). StructGraphics supports reuse through a library of visualizations (see Fig. 1a), which can contain individual marks, groups, or collections. The library stores the full visualization tree of a node, including its property structures. Users simply need to right-click on the node and then choose from a menu to add it to the library. Later, they can drag its thumbnail from the library into the canvas to create a copy and work on a different variation.

Visualization nodes, trees, and full libraries are exported and saved as JavaScript Object Notation (JSON) specification files. They can be easily imported into the user's workspace and shared. Our gallery presents several examples of StructGraphics JSON specification files. In addition to the visualization structure, our JSON syntax can describe a full StructGraphics workspace, including the tabular structures in the spreadsheet and their transformations (see next section). Thus, it could be further used to capture the full history of a design process.

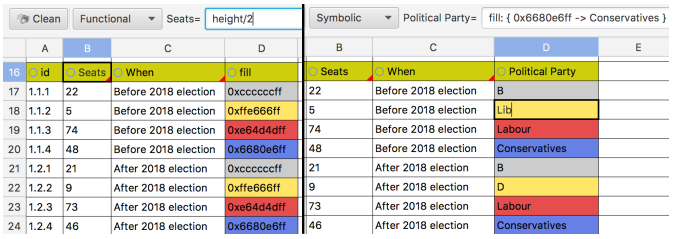
4.5 Binding Visualization Designs with Data

A *"clear trend"* observed by Bigelow et al. [9] was that *"manual encoding is not only tolerated, but even embraced by designers [...]."* However, the authors also report that *"manual encoding consumes significant time and effort [...]."* StructGraphics supports a manual data-encoding workflow but automates it by making use of the sharing relationships in a visualization property structure.

4.5.1 From Graphical Properties to Data Variables

StructGraphics does not assume that a dataset pre-exists (DP_1). Instead, it helps users to create it from scratch from their design (DP_4). Property structures in the inspector (see Fig. 5), including individual property values, property lists, and property tables, are the basis for creating a data template. At any moment, the user can drag a property structure from the inspector into the spreadsheet. In the spreadsheet, properties are transformed into *variables* and appear in columns (see Fig. 7).

Initially, variables are named after properties, but users can designate names that represent real data dimensions, such as *Time* or *Country*. Users can then type text and numbers. Variables are bidirectionally bound to the original properties and are also subject to their structural constraints. As a result, changing a variable value by editing a cell



	A	B	C	D		B	C	D	E
16	id	Seats	When	fill		Seats	When	Political Party	
17	1.1.1	22	Before 2018 election	0xccccccff		22	Before 2018 election	B	
18	1.1.2	5	Before 2018 election	0xffe666ff		5	Before 2018 election	Lib	
19	1.1.3	74	Before 2018 election	0xe64d4dff		74	Before 2018 election	Labour	
20	1.1.4	48	Before 2018 election	0x6680e6ff		48	Before 2018 election	Conservatives	
21	1.2.1	21	After 2018 election	0xccccccff		21	After 2018 election	B	
22	1.2.2	9	After 2018 election	0xffe666ff		9	After 2018 election	D	
23	1.2.3	73	After 2018 election	0xe64d4dff		73	After 2018 election	Labour	
24	1.2.4	46	After 2018 election	0x6680e6ff		46	After 2018 election	Conservatives	

Fig. 7. Transforming variables. Left: Applying a functional transformation to a variable that is bound to the height property. Right: Applying a symbolic transformation to the fill property and replacing its category values. Modifying the yellow value will cause all yellow cells to update.

in the spreadsheet will cause other cells to change, as long as they all correspond to the same property sharing. Such constraints automate the manual entry of data and prevent data-entry errors.

Alternatively, users can import data from external spreadsheet applications, such as Microsoft Excel, through copy and paste. StructGraphics thus takes advantage of the functionality of familiar applications for data formatting. Notice that a table in the spreadsheet can take a wide or a long form, depending on whether it is created from properties in the "Children Properties" section (Fig. 5b) or the "Tabular Structure" section (Fig. 5c) of the inspector. Therefore, a tuple in the spreadsheet does not necessarily represent a single glyph (see analysis by Satyanarayan et al. [41]). It can also represent a lower-level collection. In this case, the properties of its children will appear as separate columns.

A StructGraphics spreadsheet is fully synchronized with the sketcher and the inspector. Changes in a visualization can be initiated at any of these three interfaces and are immediately propagated to the two others. When the user activates the "replicate" tool to add new children to a collection, the data representations of the collection are updated with new rows or columns. However, users cannot currently add new children to a collection from the spreadsheet.

4.5.2 Data Transformations

Traditional visualization systems use *scales* [54] to map data dimensions to visual properties. StructGraphics follows the inverse approach, where users optionally transform properties to real units. To this end, the spreadsheet interface supports two transformation types (see Fig. 7):

Functional. They are expressed as mathematical functions and apply to numerical values. The spreadsheet provides a formula field for editing their syntax. To establish bidirectional mappings between original and transformed values, the system calls a solver to derive the inverse function. If, for example, the user types a function $Seats = height/2-5$, the solver will also derive the inverse function: $height = Seats*2+10$. StructGraphics will then use this formula to map the values entered in the spreadsheet to *height* property values.

Symbolic. They define 1-by-1 mappings between individual property values and alphanumerics that represent discrete categories. StructGraphics initially assigns default names as categories. Then, users can edit their values. All instances of a category are linked together since they are bound to a shared property (see Fig. 7-Right). Changing an instance will automatically cause all other instances to change as well.

Transformations apply to the variables in the spreadsheet, not to the original properties. Therefore, users can create multiple variables from a single property at different areas of the spreadsheet, where each variable can take a different function.

4.5.3 Axes, Labels, and Legends

Variables in a spreadsheet are also the basis for generating axes, labels, and legends. The user can right-click on a variable to activate a menu and choose to associate it with an axis or a legend, where only variables with symbolic transformations can generate legends. Likewise, the user can opt to decorate visualization nodes with a variable's values, e.g., to show the political parties ("Conservative", "Labour," etc.) on the flows of the Sankey diagram in Fig. 1. Collection skeletons serve as scaffolds for constructing axes. If the user activates an axis of a collection, the axis inherits the geometry and alignment constraints of its skeleton.

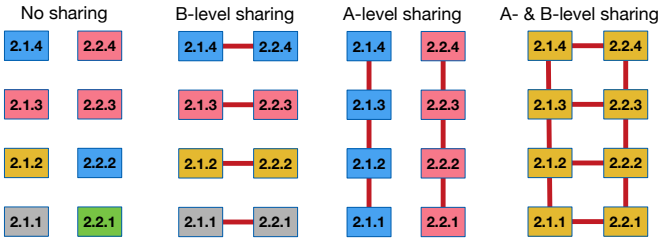


Fig. 8. Alternative binding patterns for the *fill* property of a B-level collection. We take as example the bottom visualization of Fig. 1.

4.6 Implementation Details

The StructGraphics system [4] is a JavaFX [3] application. The spreadsheet UI is an extension of the *SpreadsheetView* class of ControlsFX [1]. The spreadsheet's formula solver uses the Symja algebra library [5].

A major implementation challenge was how to create a reactive and fully synchronized UI that continuously respects all constraints of property structures. We use the property design pattern of JavaFX but extend its bi-directional binding mechanism to a group (multi-directional) binding mechanism that ensures that each property in a sharing relationship is only updated once. StructGraphics applies group property bindings at different levels of a visualization hierarchy that determine how properties are shared within each group or collection. Fig. 8 shows alternative bindings of the *fill* property for the bottom visualization of Fig. 1. The approach generalizes to larger numbers of hierarchy levels, where for each level, we add a new binding dimension.

StructGraphics ensures that all sharing bindings of the same level are synchronized. For example, if the A-level sharing breaks at the first sub-collection of our example ("A. Collection 2.1"), then the sharing will also break at the second sub-collection ("A. Collection 2.2"). As a result, users can change a property structure at any level of a visualization tree, knowing that the system will automatically apply the change to all other nodes of the same level. Finally, StructGraphics ensures that different bindings (as well as layout constraints) do not conflict with each other. For example, suppose the marks in Fig. 8 are equally spaced along the *y* axis through a spacing distribution constraint, while the *height* property is variable. Sharing their *y* property at the B-level would introduce positioning conflicts, and any change in the height of a rectangle would result in unpredictable behavior. StructGraphics automatically breaks the distribution constraint in this case to avoid the conflict.

5 Evaluation

Ren et al. [34] discuss several approaches for evaluating visualization authoring systems, including informative, reproduction, comparative, free-form user studies, and image or video galleries. More recently, Satyanarayan et al. [41] introduce *critical reflections* as an informal method for comparing systems and assessing how each system meets its authors' assumptions. We use a mix of these methods.

5.1 Gallery

We demonstrate the expressive power of StructGraphics with a gallery of visualizations. It includes high-resolution images, videos that illustrate their creation process, as well as StructGraphics JSON files that specify the structure of each visualization and their full workspace. It is available at: `https://www.lri.fr/~fanis/StructGraphics`.

5.2 User Feedback

We held individual sessions with seven participants to collect user feedback. The first two participants (P1, P2) participated in a laboratory environment in proximity with the investigator. However, due to the COVID-19 outbreak, we interrupted the study and adapted its protocol to remote sessions. We used the TeamViewer software [7], which allowed participants to view and take control of the investigator's remote workspace. Participants used a home setup that varied across sessions. Our materials and data are available at: `https://osf.io/dgbae/`.

Participants: The participants were volunteers (four women) with diverse backgrounds. P1, P5, and P7 (denoted below as ●) had 10-25 years of professional experience in graphic design and had practiced visualization design and infographics. P1 had also experience in information visualization research (Ph.D. level). P3 and P6 were faculty members with more than 12 years of experience in information visualization research. P2 was a Ph.D. student with expertise in visualization grammars, while P4 was postdoctoral researcher in interaction design.

Method: Sessions lasted 70 to 85 minutes. The participants filled out a pre-study background questionnaire. The investigator introduced the StructGraphics user interface and provided a step-by-step tutorial (45 to 50 minutes) that explained its main features. During the tutorial, the participants carried out small tasks to test their understanding. After the tutorial, they were asked to complete a chart reproduction task, which involved building a nested visualization structure, applying data transformations, editing specific data values, and displaying axes and legends. After the end of the session, each participant was asked to complete a post-study questionnaire. The questionnaire evaluated the usability of the system with respect to learning, creation difficulty, and overall experience on a 5-point scale. The participants were also asked to report on their own approach to visualization creation (data-driven vs. graphics-driven) and reflect on uses and limitations of the system.

Results: The mean rating for **learning difficulty** was 2.43 (1 = very easy, 5 = very difficult): the tool *"is pretty easy to use and learn"* (P7), *"is straightforward to use once you understand the underlying mechanism"* (P4). P3 found that it *"is extremely flexible and rich, but requires some getting used to."* P6 liked that *"there were only a few tools/concepts in total"* but added that *"one difficulty compared to freeform drawing with Illustrator is that you have to think of constraints right from the start."* A challenge that participants faced was understanding how to change the sharing of properties at the correct hierarchy level. P7 remarked that at higher collection levels, *"shared/variable settings are a bit complex to understand"* and wondered whether their presentation in the inspector could become simpler.

The mean rating for **creation difficulty** was 1.71 (1 = very easy, 5 = very difficult). P5 and P7 (expert designers) were enthusiastic. P5 explained that *"it's very pleasing to be able to easily create graphics with this tool, as it takes me much longer with my usual software."* P7 found that it is *"both intuitive and logic"* and *"allows to start visually and then to insert the real values – really easier than struggling with the settings panels and tabs of a pre-existing graph in Excel."*

Concerning the participants' **overall experience**, the mean rating was 2.0 (1 = very enjoyable, 5 = very frustrating). P6 liked *"that you can drag graphical properties to the spreadsheet, and edit them."* According to this participant, this feature could be useful for other graphic design tasks *"beyond linking visualizations to data."* P3 reported that *"this is a fun tool to experiment with. The logic behind the tool is intuitive and it can allow the creation of interesting visualizations with simple drag and drop operations."* In contrast, P4 found that the grouping *"is not 100% intuitive."* She would prefer to manipulate *"individual objects first, and then allow building groups later on."*

Reflecting on their own approach, few only participants could identify it as purely data-driven (P2) or purely graphics-driven (P5). P1, P3, P6, and P7 explained that their approach depends on the type or the phase of the task. According to P3, data-exploration tasks are always data-driven, and the tool is not well adapted. However, *"when building a vis for communication I often start with the message I want to communicate in mind and think of the visuals first [...] the tool makes this process of adapting subtle properties very easy to do, helps customize all aspects of the visual and gives absolute control on what the vis will look at the end"* (P3). P6 expressed the opinion that whether the task is data-driven (*"graphic meant to communicate a message or insights about a particular dataset"*) or not (*"designing a visualization meant to work with many datasets"*), *"visualization is always graphics-driven."* The participant continued that *"you can definitely design a visualization without a dataset"* but *"you eventually always have to test it and refine it by plugging data"* (P6). In the same line, P7 argued that *"data first"* or *"graphics first"* are both true, since some graphic representations match specific datasets, while others do not. But the designer emphasized that *"it is important to have an idea of what things could look like, even if data is not yet complete (or arrived)"* (P7).

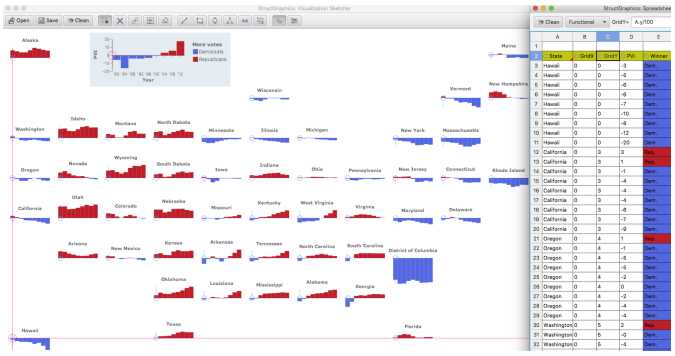


Fig. 9. Recreating the Wall Street Journal's *A Field Guide to Red and Blue America* visualization [58] with StructGraphics.

5.3 Critical Reflections

Satyanarayan et al. [41] present their *collective* critical reflections. Here, although we report on the author's own reflections, we use the analysis of Satyanarayan et al. [41] as reference. We reflect on StructGraphics' capabilities, assumptions, and limitations by referring to results of our user study. We also compare StructGraphics to existing visualization authoring systems and discuss future extensions.

5.3.1 Comparative Scenario

We start with a scenario that reproduces *A Field Guide to Red and Blue America*, a nested visualization published in The Wall Street Journal [58]. Satyanarayan et al. [41] use the same scenario to compare Lyra [40], Data Illustrator [21], and Charticulator [33]. The original visualization contains a bar chart for each of the 51 states of USA that depicts the balance between votes for Democrats and Republicans from 1980 to 2012. Fig. 9 shows the full visualization on StructGraphics' sketcher, while part of the dataset is shown on the spreadsheet. A video demonstration of the creation process and the JSON specification of the visualization are included in our gallery.

As a first step, we bring a bar chart design from our library or build it from scratch. To do so, we draw a rectangle and replicate it along the x direction. Then, we vary the height of the rectangles and use the tool to create a collection. In the inspector, we make their *fill* property variable, apply a red and blue color, and refine their alignement constraints within the collection: *A.sticky-x = Yes* and *A.sticky-y = Yes*. As a second step, we create 50 copies of the bar chart and then create a B-level collection. We do so by replicating the bar chart: first along the y axis ($\times 5$) and then along the x axis ($\times 10$). In the inspector, we ensure that *height* and *fill* are variable at all levels. We then explore alternative layouts by freely moving the bar charts on the cavas.

We now decide to bind our visualization with the original dataset, which contains the following columns: *State*, *GridX* and *GridY* coordinates, *Year*, *Inclination* ("Dem." or "Rep."), and *PVI*, a relative vote measure that is negative for Democrats and positive for Republicans. We open it with Microsoft Excel and sort it first by *Year* and then by *State*. In the inspector, we navigate to the tabular structure of the top collection, drag the columns *A.id*, *A.x*, *A.y*, *height*, and *fill*, and drop them into the spreadsheet. We rename *A.id* to "State," turn its transformation to symbolic, and bring the *State* column from Excel via copy and paste. Similarly, we make the *fill* variable symbolic and copy the *Inclination* column. For the other variables (*A.x*, *A.y*, *height*), we keep a functional transformation and choose a convenient function (e.g., $GridY = A.y/100$, $GridX = A.x/120$, $PVI = 0.7 * height$) to map pixels to data units. When we copy their values from Excel, all bar charts take the correct height values and positions. To display the names of the states, we right-click on the "State" column and choose "Show on glyphs" from the menu. Later, we decide to drag the "Hawaii" chart further left to highlight the relative geographic location of this state.

Finally, we add a larger bar chart to explain the coloring scheme and dimensions of each bar chart (see Fig. 9). We copy the "West Virginia" bar chart, make its bars wider and vary their heights. We drag its x, y, and *fill* properties to the spreadsheet, where we add labels and transformations (as above) and populate axis labels and a legend.

5.3.2 Critical Reflections on System Components

We first reflect on the components of StructGraphics visualizations.

Marks and Glyphs. Like Data Illustrator [21], StructGraphics requires its users to activate a tool from the toolbar to draw a mark. This approach facilitates the use of key shortcuts, which are extensively used by design experts [48]. Currently, StructGraphics only supports predefined shapes (like Lyra and Charticulator), lacking Data Illustrator's capabilities of drawing arbitrary paths. As in Data Illustrator, StructGraphics groups are drawn directly on the canvas. In contrast to groups or glyphs in Data Illustrator and Charticulator, groups in StructGraphics can be infinitely nested. A strength of our design is that groups and collections share the same representation, behavior, and creation process.

Unlike Data Illustrator and Lyra, StructGraphics does not treat curves, areas, and connections as marks – it represents them as separate properties within collections. This design is closer to Charticulator, which also expresses connections as separate objects, but StructGraphics further lets users create connections among sub-collections (not only among glyphs).

Data Editing and Data Binding. The spreadsheet is an integral component of the StructGraphics user interface. It enables users to shape their dataset based on the visual structure of the visualization design, apply data transformations to create mappings between pixels and data units, edit individual or multiple values (e.g., by copy-paste), and communicate with external spreadsheet applications. This is a fundamental difference between StructGraphics and current visualization authoring tools [41], which do not deal with data entry and data formatting.

The data-binding philosophy of StructGraphics is also distinct. In previous systems [21, 33, 40], data-binding precedes and fully specifies the creation of all instances of a certain type of glyphs. In contrast, StructGraphics requires that all glyph instances are explicitly created by the user. For example, in our scenario, we created all 51 bar charts (one for each state) directly on the canvas (see Fig. 10). Unfortunately, the creation process becomes cumbersome as the number of glyphs increases. Thus, StructGraphics may not be suitable if the goal is to visualize large datasets with hundreds or thousands of data points.

Furthermore, even after property structures have been bound to data, StructGraphics still allows users to freely manipulate the graphical properties of their visualizations on the sketcher or the inspector. Bindings between data and graphical properties are fully bi-directional, allowing users to explore visual layouts without being constrained by specific data values. Users can further bind the same property structure with different pieces of data and transformations to explore alternative solutions. On the downside, this freedom raises *ethical considerations,* since users may be tempted to alter or fake their data in order to achieve a compelling visual result. Our study questionnaire asked our participants' opinion on this issue. Overall, participants did not view it as a problem: *"fake graphs based on fake data is a really different problem"* (P7), *"if people want to create fake data, they can already do it without those tools"* (P6). P3 expressed the same opinion but also added: *"I believe it is easy to make mistakes [...] if you have thousands of data points, this is hard to double-check."* Adding common scales of measurement and data summarization mechanisms could potentially improve data entry and prevent errors in the spreadsheet interface.

Scales, Axes, and Legends. In StructGraphics, scales, axes, and legends are generated at will by interacting with the data variables at the spreadsheet interface. However, the sketcher does not currently allow users to edit them and further customize them. Referring to this lack of support, P4 complained that *"it is a little bit frustrating that you cannot manipulate everything on the interface."* This is a clear limitation in comparison with the capabilities of existing tools [41].

Layout. Lyra and Charticulator use anchors to reference and relatively position visual objets. In StructGraphics, anchors are replaced by x and y reference positions at the level of marks, and skeletons at the level of groups and collections. Skeletons are constantly visible such that designers can directly manipulate the position of groups and collections (P6: *"I like that collections are reified. The fact that groups are invisible is definitely an issue with Illustrator"*). Like Charticulator, StructGraphics can express layout constraints at the level

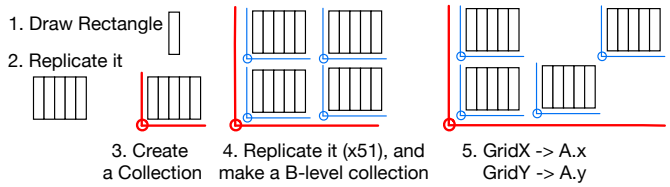


Fig. 10. Main steps for creating the layout of the visualization in Fig. 9.

of graphics. Both systems support alignment and spacing constraints but through different mechanisms. Charticulator's constraints are based on the alignment and margin properties of marks and a mechanism of persistent snapping. StructGraphics' constraints are based on the stickiness and distribution properties of groups and collections and their sharing structures. Note that StructGraphics' sharing mechanism allows for linking together other properties of a glyph. For example, we can create a group of three rectangles to represent an error bar, where the fill, width, and height of the top and bottom rectangles are shared and thus commonly change. This mechanism relieves users from having to bind the properties of each individual object in a group to a data variable and facilitates reuse.

Charticulator makes use of a constraint solver that could potentially support more complex layout constraints. Nevertheless, further complexity comes with a cost, since users may not anticipate the solver's logic. A major requirement for StructGraphics' design was that object manipulations must be incremental across the full interface, where all layout constraints are constantly evaluated and propagated as the user directly manipulates objects. Such behavior requires simple rules with predictable outcomes.

StructGraphics uses the same layout mechanisms for groups and collections, while users can define the full layout without a dataset (see Fig. 10: Steps 1-4). In contrast, Lyra, Data Illustrator, and Charticulator require the visualization author to bind a layout structure to the variables of a dataset. As we discuss below, this key difference derives from our different assumptions about the user task. Finally, StructGraphics lacks Charticulator's support for polar and curve coordinates. In order to support alternative coordinate systems and nested 2D representations (e.g., as tree maps in Lyra), we need to extend StructGraphics with additional by-example-specification tools and layout mechanisms.

5.3.3 Critical Reflections on our Assumptions

We also reflect on how StructGraphics' design meets our assumptions.

The Task: Designing, *not Authoring*. A fundamental assumption of the developers of Lyra, Data Illustrator, and Charticulator *"is that people want to author a chart, not to design one"* [41]. For StructGraphics, we reverse this assumption. We assume that users want to *design* a chart, not to *author* one. Our participants' feedback confirms this assumption. Several participants expressed the opinion that the tool is not well adapted to purely data-driven tasks, in particular data exploration (P3) or visualization generation from existing datasets (P1, P2, P4). In contrast, they all found it valuable for design-oriented tasks (P1, P2, P6, P7), for education purposes (P4), for generating *"simulation data"* (P4), or for data communication purposes (P3, P5).

In summary, visualization authoring is a data-driven activity, whereas design tasks are often driven by graphics. StructGraphics lets designers focus on their graphics, where any reference to data is only implicit. The distinction between "explicit" and "implicit" data is fundamental for design tasks. Data can be incomplete or ill defined in the designer's mind and progressively emerge as a visual structure takes shape.

The Designer: Literacy & Skill Transfer. Since we target design practitioners, we assumed that StructGraphics users have experience with vector-graphics editing applications and spreadsheets, are familiar with visualization tools and grammars but may not know how to code. As we discussed earlier, the participants felt that the tool was overall not difficult to learn but were sometimes perplexed about how to locate and manage the property structures in the inspector. A possible direction for improving the editing experience of property structures is to reify their sharing relationships through constraint holders that users can directly interact with on the sketcher. P6 also remarked that the way transformations are expressed in the formula editor (e.g., $deaths = height * 10$) is different from how *"visualization practitioners are used to think"* (e.g., $height = deaths/10$). Since the StructGraphics workflow may require users to think in either direction, a better approach might be to allow them to switch between those two representations.

The Data: Formatted during Design. Unlike Satyanarayan et al. [41], we do not assume that data are appropriately formatted prior to the task. In StructGraphics, the data schema is defined as the user constructs a visualization structure, based on the property sharing of its components. Users may also decide to not deal with specific data values at all, or even split their data into different areas of the spreadsheet. However, there are situations (like in our scenario) where users will need to bring real data. StructGraphics provides data-transformation capabilities but mostly relies on external spreadsheet applications for loading and transforming existing datasets. For the dataset of our example, we used the formula editor of Microsoft Excel to derive the *Inclination* column: *IF(PVI* > 0*, "Rep.", "Dem.")*. We then copied the column and pasted it to the StructGraphics spreadsheet to associate it with the *fill* property of the bars. However, the functional relationship between *PVI* (mapped to *height*) and *Inclination* (mapped to *fill*) is not expressed at the level of the property structures. Thus, if the user turns the height of a bar from positive to negative, its fill color will not automatically change to blue. More generally, StructGraphics cannot express functional dependencies between properties, which would require more expressive visual languages, such as iVoLVER [26] or *Linkify* [23]. We defer the study of this problem to future work.

Export, Reuse, & Interoperability. Reuse is a fondamental activity of all design practices. A strength of StructGraphics is the fact that graphical structures of any nesting level are fully reusable and independent of datasets or data schemas. Designers can further build their own libraries of visualization structures and share them as lightweight JSON files. P1 commented that StructGraphics would be more valuable as an intermediate design tool, which requires better interoperability with other visualization software. A future direction is to develop wrappers to/from popular formats, such as Vega [43] (as in Lyra [40]), Power BI custom visuals (as in Charticulator [33]), or ggplot2 [2] code snippets.

Finally, during their task, some participants expressed the need to apply structural changes to their collections, such as directly adding shapes to existing groups in the canvas. In order to construct a visualization with StructGraphics, the designer has to start from its basic shapes and progressively move to higher-level groups and collections. Unfortunately, the reverse workflow is not currently supported. For example, suppose a designer reuses the structure of the Sankey diagram in Fig. 1 but wants to replace the rectangular shapes by more complex glyphs, such as the sailboats of the top visualization in the same figure. Extending StructGraphics to support such top-down visualization constructions would allow designers to experiment with richer design-remixing workflows. Other interesting extensions in this direction include adding partitioning tools (as in Data Illustrator [21]) and supporting *embedded merge and split operations* [39].

6 Summary and Conclusion

We presented StructGraphics, a new graphics-driven approach for data visualization design. StructGraphics enable designers to produce data-agnostic visualizations that they can then reuse with different datasets. We introduced a framework for constructing property structures that determine how graphical properties are grouped and shared among the nodes of a visualization hierarchy. We showed that property structures act both as graphical constraints and as templates for data.

Creating a visualization design tool that is appropriate for all users and design needs may be impossible or even not desirable [9]. The StructGraphics approach provides a flexible design workflow but is still limited in the range of design solutions it supports. We second Liu et al. [21] who throughout their design iterations observe that constructing a *"coherent set of concepts and tools that behave consistently"* can be *"a great challenge."* How to extend the StructGraphics approach to deal with more expressive visualizations and alternative design workflows is an exiting challenge for future research.

REFERENCES

[1] ControlsFX. https://github.com/controlsfx/controlsfx.

[2] ggplot2. https://github.com/tidyverse/ggplot2.

[3] JavaFX. https://openjfx.io/.

[4] StructGraphics code repository. https://gitlab.inria.fr/structgraphics/code.

[5] Symja - java computer algebra library. https://bitbucket.org/axelclk/symja_android_library/wiki/Home.

[6] Tableau. https://www.tableau.com.

[7] J. Accot and S. Zhai. More than dotting the i's — foundations for crossing-based interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '02, pp. 73–80. ACM, New York, NY, USA, 2002. doi: 10.1145/503376.503390

[8] C. Appert, O. Chapuis, and E. Pietriga. Dwell-and-spring: Undo for direct manipulation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pp. 1957–1966. ACM, New York, NY, USA, 2012. doi: 10.1145/2207676.2208339

[9] A. Bigelow, S. Drucker, D. Fisher, and M. Meyer. Reflections on how designers design with data. In *Proceedings of the 2014 International Working Conference on Advanced Visual Interfaces*, AVI '14, pp. 17–24. ACM, New York, NY, USA, 2014. doi: 10.1145/2598153.2598175

[10] A. Bigelow, S. Drucker, D. Fisher, and M. Meyer. Iterating between tools to create and edit visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):481–490, Jan. 2017. doi: 10.1109/TVCG.2016.2598609

[11] M. Bostock and J. Heer. Protovis: A graphical toolkit for visualization. *IEEE transactions on visualization and computer graphics*, 15:1121–8, 11 2009. doi: 10.1109/TVCG.2009.174

[12] M. Bostock, V. Ogievetsky, and J. Heer. D3: Data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, Dec. 2011. doi: 10.1109/TVCG.2011.185

[13] M. Ciolfi Felice, N. Maudet, W. E. Mackay, and M. Beaudouin-Lafon. Beyond snapping: Persistent, tweakable alignment and distribution with StickyLines. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, UIST '16, pp. 133–144. ACM, New York, NY, USA, 2016. doi: 10.1145/2984511.2984577

[14] A. Dix, R. Cowgill, C. Bashford, S. McVeigh, and R. Ridgewell. Spreadsheets as user interfaces. In *Proceedings of the International Working Conference on Advanced Visual Interfaces*, AVI '16, pp. 192–195. ACM, New York, NY, USA, 2016. doi: 10.1145/2909132.2909271

[15] R. Hoarau and S. Conversy. Augmenting the scope of interactions with implicit and explicit graphical structures. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pp. 1937–1946. ACM, New York, NY, USA, 2012. doi: 10.1145/2207676.2208337

[16] D. Jung, W. Kim, H. Song, J.-i. Hwang, B. Lee, B. Kim, and J. Seo. ChartSense: Interactive data extraction from chart images. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, p. 6706–6717. Association for Computing Machinery, New York, NY, USA, 2017. doi: 10.1145/3025453.3025957

[17] N. W. Kim, N. Henry Riche, B. Bach, G. A. Xu, M. Brehmer, K. Hinckley, M. Pahud, H. Xia, M. McGuffin, and H. Pfister. DataToon: Drawing dynamic network comics with pen + touch interaction. In *CHI 2019*, pp. 1–12. ACM, May 2019.

[18] N. W. Kim, E. Schweickart, Z. Liu, M. Dontcheva, W. Li, J. Popovic, and H. Pfister. Data-Driven Guides: Supporting expressive design for information graphics. *IEEE Transactions on Visualization and Computer Graphics*, PP(99):1–1, Jan 2017 2017.

[19] R. Kosara. Presentation-oriented visualization techniques. *IEEE Computer Graphics and Applications*, 36(1):80–85, Jan 2016. doi: 10.1109/MCG.2016.2

[20] B. Lee, R. Habib Kazi, and G. Smith. SketchStory: Telling more engaging stories with data through freeform sketching. *IEEE transactions on visualization and computer graphics*, 19:2416–25, 12 2013. doi: 10.1109/TVCG.2013.191

[21] Z. Liu, J. Thompson, A. Wilson, M. Dontcheva, J. Delorey, S. Grigg, B. Kerr, and J. Stasko. Data Illustrator: Augmenting vector design tools with lazy data binding for expressive visualization authoring. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pp. 123:1–123:13. ACM, New York, NY, USA, 2018. doi: 10.1145/3173574.3173697

[22] H. Mance, J. Pickard, and L. Hughes. Labour fails to make big gains in england local elections. https://www.ft.com/content/5f2a3dcc-4f5b-11e8-a7a9-37318e776bab, 2018.

[23] N. Maudet, G. Jalal, P. Tchernavskij, M. Beaudouin-Lafon, and W. E. Mackay. Beyond grids: Interactive graphical substrates to structure digital layout. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, pp. 5053–5064. ACM, New York, NY, USA, 2017. doi: 10.1145/3025453.3025718

[24] H. Mei, W. Chen, Y. Ma, H. Guan, and W. Hu. VisComposer: A visual programmable composition environment for information visualization. *Visual Informatics*, 2(1):71 – 81, 2018. Proceedings of PacificVAST 2018. doi: 10.1016/j.visinf.2018.04.008

[25] G. G. Méndez, U. Hinrichs, and M. A. Nacenta. Bottom-up vs. top-down: Trade-offs in efficiency, understanding, freedom and creativity with infovis tools. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, p. 841–852. Association for Computing Machinery, New York, NY, USA, 2017. doi: 10.1145/3025453.3025942

[26] G. G. Méndez, M. A. Nacenta, and S. Vandenheste. IVoLVER: Interactive visual language for visualization extraction and reconstruction. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, p. 4073–4085. Association for Computing Machinery, New York, NY, USA, 2016. doi: 10.1145/2858036.2858435

[27] B. A. Myers, J. Goldstein, and M. A. Goldberg. Creating charts by demonstration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '94, p. 106–111. Association for Computing Machinery, New York, NY, USA, 1994. doi: 10.1145/191666.191715

[28] A. Oulasvirta, N. Ramesh Dayama, M. Shiripour, M. John, and A. Karrenbauer. Combinatorial optimization of graphical user interface designs. *Proceedings of the IEEE*, 108(3):434–464, 2020.

[29] P. O'Donovan, A. Agarwala, and A. Hertzmann. DesignScape: Design with interactive layout suggestions. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, p. 1221–1224. Association for Computing Machinery, New York, NY, USA, 2015. doi: 10.1145/2702123.2702149

[30] J. Poco and J. Heer. Reverse-engineering visualizations: Recovering visual encodings from chart images. *Computer Graphics Forum*, 36(3):353–363, 2017. doi: 10.1111/cgf.13193

[31] B. Reinert, T. Ritschel, and H.-P. Seidel. Interactive by-example design of artistic packing layouts. *ACM Trans. Graph.*, 32(6):218:1–218:7, Nov. 2013. doi: 10.1145/2508363.2508409

[32] D. Ren, T. Höllerer, and X. Yuan. iVisDesigner: Expressive interactive design of information visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2092–2101, 2014. doi: 10.1109/TVCG.2014.2346291

[33] D. Ren, B. Lee, and M. Brehmer. Charticulator: Interactive construction of bespoke chart layouts. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):789–799, Jan. 2019. doi: 10.1109/TVCG.2018.2865158

[34] D. Ren, B. Lee, M. Brehmer, and N. Henry Riche. Reflecting on the evaluation of visualization authoring systems. In *BELIV Workshop at IEEE VIS: Evaluation and Beyond - Methodological Approaches for Visualization*. IEEE, October 2018.

[35] S. F. Roth, J. Kolojejchick, J. Mattis, and J. Goldstein. Interactive graphic design using automatic presentation knowledge. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '94, pp. 112–117. ACM, New York, NY, USA, 1994. doi: 10.1145/191666.191719

[36] K. Ryall, J. Marks, and S. Shieber. An interactive constraint-based system for drawing graphs. In *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology*, UIST '97, pp. 97–104. ACM, New York, NY, USA, 1997. doi: 10.1145/263407.263521

[37] B. Saket and A. Endert. Demonstrational interaction for data visualization. *IEEE Computer Graphics and Applications*, 39(3):67–72, May 2019. doi: 10.1109/MCG.2019.2903711

[38] B. Saket, H. Y. Kim, E. T. Brown, and A. Endert. Visualization by demonstration: An interaction paradigm for visual data exploration. *IEEE Transactions on Visualization and Computer Graphics*, 23:331–340, 2017.

[39] A. Sarvghad, B. Saket, A. Endert, and N. Weibel. Embedded merge split: Visual adjustment of data grouping. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):800–809, Jan 2019. doi: 10.1109/TVCG.2018.2865075

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

[41] A. Satyanarayan, B. Lee, D. Ren, J. Heer, J. Stasko, J. L. Thompson,

M. Brehmer, and Z. Liu. Critical reflections on visualization authoring systems. *IEEE Transactions on Visualization and Computer Graphics*, 26:461–471, 2019.

[42] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer. Vega-Lite: A grammar of interactive graphics. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):341–350, 2017.

[43] A. Satyanarayan, R. Russell, J. Hoffswell, and J. Heer. Reactive Vega: A streaming dataflow architecture for declarative interactive visualization. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):659–668, 2016.

[44] A. Satyanarayan, K. Wongsuphasawat, and J. Heer. Declarative interaction design for data visualization. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST '14, p. 669–678. Association for Computing Machinery, New York, NY, USA, 2014. doi: 10.1145/2642918.2647360

[45] C. Stolte and P. Hanrahan. Polaris: A system for query, analysis and visualization of multi-dimensional relational databases. In *Proceedings of the IEEE Symposium on Information Visualization 2000*, InfoVis '00, pp. 5–. IEEE Computer Society, Washington, DC, USA, 2000.

[46] K. Todi, D. Weir, and A. Oulasvirta. Sketchplore: Sketch and explore with a layout optimiser. In *Proceedings of the 2016 ACM Conference on Designing Interactive Systems*, DIS '16, p. 543–555. Association for Computing Machinery, New York, NY, USA, 2016. doi: 10.1145/2901790.2901817

[47] T. Tsandilas, A. Bezerianos, and T. Jacob. SketchSliders: Sketching Widgets for Visual Exploration on Wall Displays. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pp. 3255–3264. ACM, Seoul, South Korea, Apr. 2015. doi: 10.1145/2702123.2702129

[48] T. Tsandilas, M. Grammatikou, and S. Huot. BricoSketch: Mixing paper and computer drawing tools in professional illustration. In *Proceedings of the 2015 International Conference on Interactive Tabletops & Surfaces, ITS 2015, Funchal, Portugal, November 15-18, 2015*, pp. 127–136. ACM, 2015. doi: 10.1145/2817721.2817729

[49] B. Victor. Drawing dynamic visualizations. `http://worrydream.com/DrawingDynamicVisualizationsTalkAddendum`, 2013.

[50] R. Vuillemot and J. Boy. Structuring Visualization Mock-ups at a Graphical Level by Dividing the Display Space. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):424 – 434, Oct. 2017. doi: 10.1109/TVCG.2017.2743998

[51] J. Walny, S. Huron, and S. Carpendale. An exploratory study of data sketching for visual representation. *Computer Graphics Forum*, 34(3):231–240, 2015. doi: 10.1111/cgf.12635

[52] C. Wang, Y. Feng, R. Bodik, A. Cheung, and I. Dillig. Visualization by example. *Proc. ACM Program. Lang.*, 4(POPL), Dec. 2019. doi: 10.1145/3371117

[53] Y. Wang, H. Zhang, H. Huang, X. Chen, Q. Yin, Z. Hou, D. Zhang, Q. Luo, and H. Qu. InfoNice: Easy creation of information graphics. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18. Association for Computing Machinery, New York, NY, USA, 2018. doi: 10.1145/3173574.3173909

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

[55] L. Wilkinson. *The Grammar of Graphics (Statistics and Computing)*. Springer-Verlag, Berlin, Heidelberg, 2005.

[56] H. Xia, N. Henry Riche, F. Chevalier, B. De Araujo, and D. Wigdor. DataInk: Direct and creative data-oriented drawing. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pp. 223:1–223:13. ACM, New York, NY, USA, 2018. doi: 10.1145/3173574.3173797

[57] P. Xu, H. Fu, T. Igarashi, and C.-L. Tai. Global beautification of layouts with interactive ambiguity resolution. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST '14, pp. 243–252. ACM, New York, NY, USA, 2014. doi: 10.1145/2642918.2647398

[58] R. Yeip, S. A. Thompson, and W. Welch. A field guide to red and blue america. `http://graphics.wsj.com/elections/2016/field-guide-red-blue-america`, 2016.

[59] J. E. Zhang, N. Sultanum, A. Bezerianos, and F. Chevalier. DataQuilt: Extracting visual elements from images to craft pictorial visualizations. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20. ACM, 2020. doi: 10.1145/3313831.3376172