Computers & Graphics 92 (2020) 13–27

Contents lists available at ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Technical Section

Visception: An interactive visual framework for nested visualization design

Yngve Sekse Kristiansen*, Stefan Bruckner

Department of Informatics, University of Bergen, Norway

ARTICLE INFO

Article history:
Received 16 January 2020
Revised 11 August 2020
Accepted 11 August 2020
Available online 17 August 2020

Keywords:
Information visualization
Nested visualizations
Nesting

ABSTRACT

Nesting is the embedding of charts into the marks of another chart. Related to principles such as Tufte's rule of utilizing micro/macro readings, nested visualizations have been employed to increase information density, providing compact representations of multi-dimensional and multi-typed data entities. Visual authoring tools are becoming increasingly prevalent, as they make visualization technology accessible to non-expert users such as data journalists, but existing frameworks provide no or only very limited functionality related to the creation of nested visualizations. In this paper, we present an interactive visual approach for the flexible generation of nested multilayer visualizations. Based on a hierarchical representation of nesting relationships coupled with a highly customizable mechanism for specifying data mappings, we contribute a flexible framework that enables defining and editing data-driven multi-level visualizations. As a demonstration of the viability of our framework, we contribute a visual builder for exploring, customizing and switching between different designs, along with example visualizations to demonstrate the range of expression. The resulting system allows for the generation of complex nested charts with a high degree of flexibility and fluidity using a drag and drop interface.

1. Introduction

Nesting or embedding, i.e., the integration of additional visualizations into the marks of a chart, enables the presentation of information-dense graphical data depictions. By augmenting an outer visualization with additional details presented as information layers as part of its marks, rich depictions of complex data can be constructed from a few basic building blocks.

Nested visualizations are frequently applied in order to convey multi-faceted data and facilitate storytelling. In particular in fields such as data journalism, users would greatly benefit from being able to create such visualizations without programming.

In recent years, a new generation of visual authoring systems such as Data Illustrator [1] and Charticulator [2] have been developed to enable the creation of custom charts via intuitive visual interfaces accessible to non-experts. In particular, they aim to support the flexibility and customization options of design tools such as Adobe Illustrator, while still providing a data-driven visualization environment. While these systems feature advanced mechanisms for designing bespoke charts, they provide no or only very limited support for nesting.

In this paper, we present Visception, a visualization framework built from the ground up based on nesting as a first-class operation. For this purpose, we introduce the VC-tree as our central data structure. We detail how this approach offers flexible data mappings for transforming tabular input data into data objects, enabling the expression of a wide range of different groupings when creating a nested visualization. Individual charts are made composable with other charts through our framework's implicit handling of nesting and deformation. By providing a set of simple operations to manipulate a VC-tree, we are able to realize a large number of different embedding and layering relationships. Furthermore, we show that using a more inclusive definition of what constitutes a data-mappable channel provides additional design flexibility in the context of nesting. The full functionality of our framework is exposed in the form of a visual builder, and we demonstrate that our approach allows for the easy generation of a variety of complex nested visualizations.

2. Related work

2.1. Formal graphics specifications

Foundational works like Bertin's Semiology of Graphics [3] and Wilkinson's Grammar of Graphics [4] provide constructs for

* Corresponding author:
E-mail address: ykr088@uib.no (Y. Sekse Kristiansen).

https://doi.org/10.1016/j.cag.2020.08.007

concisely specifying and reasoning about graphics. Bertin describes marks as basic graphical units, and visual variables as modifications (position, shape, value, color, orientation, texture, and so on) that can be applied to marks. Munzner [5] consolidated and extended similar approaches for discussing visualizations, and also introduced the term channel as a way to control the appearance of marks. Visception features a more inclusive kind of channel, denoted as a VC-channel. Layout parameters as well as global chart properties such as for example the title or background of a chart, are exposed as VC-channels.

Early information visualization techniques utilized low level libraries. While such low level libraries enable the expression of graphics, they are not necessarily suitable for visual thinking. Thus, multiple visualization toolkits that raise the level of abstraction have been developed. Examples of such libraries include Prefuse [6], Protovis [7] and D3 [8].

At an even higher level of abstraction, visualization grammars such as Vega [9] enable clear expression of a wide range of visualizations declaratively. In Vega, each chart is a unit which takes in data and associated transformations, mark type, and encodings. Each encoding is a specification for how a channel is mapped. Built on Vega, Vega-Lite [10] is both a grammar of interaction and graphics. A prominent feature of the Vega-Lite grammar is its view composition algebra with four operations: *Layer* for placing one chart on top of another, *Concatenation* for placing charts side by side, *Facet* for creating one view per distinct value of a field, and *Repeat* for creating several views with the same input data. Visception uses nesting and data groupings independent of chart inputs to provide a more flexible and recursive nesting behavior. In terms of data, Visception's nesting operation corresponds to Vega's facet operation. Visually, Vega provides rows and columns as host spaces for child charts, while Visception provides a set of customizable charts as host spaces. These charts are more flexible and controllable than rectangular host spaces generated via the facet operation, and may be mapped to data. Visception's VC-channels function similarly to Vega's encodings, although each chart in Visception will have a larger set of VC-channels to modify the layout.

Tree visualization grammars are closely related to nesting. Li et al. [11] introduced a declarative grammar of tree visualizations, enabling users to rapidly specify both explicit and implicit depictions. Their visual builder allows the user to combine different tree layout algorithms, and to adjust finer aspects such as margin and padding between nodes. Visception has a similar approach in that it combines layouts. However, Visception is focused on enabling the creation of nested visualizations from tabular data, while GoTree is focused on creating tree visualizations from predefined hierarchies. Schulz et al. [12] propose a set of functional building blocks denoted as layout operators that enable building explicit node-link layouts as well as implicit space-filling layouts. They specify a highly flexible layout pipeline for rendering such trees, and expose operators that allow the user to modify the layout in a variety of ways. Similarly, Visception uses a layout pipeline in its underlying implementation, and exposes parameters that modify the layout as VC-channels. Visception only does top-down explicit layouts, and does not work bottom-up as Schulz' generative approach.

Other more specialized grammars focus on particular categories of visualizations. ATOM [13] is grammar for unit visualizations. With this grammar the user can subdivide a space at multiple levels and fill in units — one for each datum. Visception exposes a similar design space by providing a *unit* chart type. Wickham and Hofmann's Product Plots [14] is a framework for transforming and combining area-based visualizations. They define three 1D primitives: bars, spines and tiles. With these three primitives, they show that it is possible to express a wide range of both simple and complex visual representations of data. While both Product Plots and Visception use nesting, Visception leverages the chart type itself to provide host spaces for child charts, while Product Plots subdivide rectangles with a small set of rectangle-based 1D primitives.

Schulz and Hadlak [15] presented a way of representing visualizations by blending together existing visualizations defined as presets. In the process of describing how to interpolate between visualizations, they expose connections between different chart types, such as the polar area chart and the bar chart. Rather than presets, Visception offers a set of charts the user may choose and combine. Thus, Visception covers a discretized subset of the design space covered by preset-based visualization.

We are inspired by these approaches of combining and deforming 2D geometries, and use such concepts to handle nesting and deformation behavior for all chart types within our framework. Vuillemot and Boy [16] use nested and composite visualizations to facilitate the exploration of designs, regardless of data. They define a visual grammar with partitioning patterns and data transform operations. With our framework, we enable the specification of charts without requiring explicit specification of nesting behavior, making it easy to introduce new building blocks to the language.

2.2. Data exploration and visual authoring

Data exploration tools focus primarily on what the user can learn about the data, rather than design and aesthetics. IVEE [17], Visage [18], and Tioga2 [19] were some of the first systems to enable visual building of queries, and visualizing the results. Polaris [20] by Stolte et al. (later commercialized as Tableau) enables rapid exploration of large multidimensional datasets, leveraging a table algebra to display a wide range of charts.

Visual authoring tools are more focused on design than data exploration, yet they serve a similar purpose and can potentially be as powerful as data exploration systems. Charticulator [2] enables the user to define a chart by articulating compound marks or glyphs, as well as links between these. Lyra [21] enables the interactive design of a wide range of visualizations using drag and drop operations. Lyra also provides visual data pipelines that allow for the expression of advanced layouts and data transformations. iVisDesigner [22] aims to cover a wide range of the visualization design space by leveraging modular visualization design. Data Illustrator [1] augments vector design tools with new concepts and operators, enabling users to bind parts of a vector-based illustration to data. Data Driven Guides [23] has a similar approach to Data Illustrator, allowing users to create data-driven shapes (also known as guides) that can be decorated by custom vector graphics. iVolver [24] provides users with the means to extract and reconstruct visualizations from both data sets and existing visualizations (including images and webpages).

While our work shares many of the general goals with the approaches mentioned above, Visception focuses on nested, aesthetic visualizations. We provide an editor and a framework to enable the design of highly customized information rich visualizations by leveraging nesting. Our definition of charts with VC-channels are made to be compatible and consistent for both nested and non-nested charts.

2.3. Nested visualization and related techniques

Hierarchical and small-multiple layouts are closely related to nesting. Schulz et al. [25] surveyed the design space of hierarchy visualization, providing an overview of a large number of different techniques (both 2D and 3D) used to visualize hierarchies, as well as exposing unexplored parts of the design space. LeBlanc et al. [26] describe the technique of dimensional stacking, allowing the user to map high-dimensional data to a relatively small 2D space. Similar expressiveness can be achieved in Visception by nesting

rows and *columns*. Wang et al. [27] introduced the circle packing layout, nesting circles within circles with arbitrary depths. This layout may also be expressed by nesting circles with a force-directed layout [28] within one another, which is achievable in Visception by nesting *plot* charts. Treemap layouts are often used to visualize hierarchies. Baudel et al. [29] present a generic algorithm that expresses most of the different existing treemap layouts using only a few basic operations. Visception follows a similar line of thought: to expose parametrized generic charts that may express other specific charts.

Using nested visualizations it is possible to express complex relationships by only using a few simple building blocks. Parker et al. [30], as early as 1998, designed NestedVision3D, allowing for the exploration of nested graphs to explore the structure of computer programs. ZAME [31] (Zoomable Adjacency Matrix Explorer) nests glyphs inside each cell of an adjacency matrix. Combined with zooming, panning, and aggregation represented as glyphs, ZAME allows for the exploration of large datasets. Javed and Elmqvist [32] detail four visual composition operators: juxtaposition, superimposition, overloading and nesting. Visception provides a flexible layering operation that, combined with movable and resizable bounds, achieves a similar level of expressiveness as using these four operators. Juxtaposition and superimposition are expressible by simply editing the bounds of a chart. HEDA (Heterogenous Data Attributes) [33] is a generic interactive visualization component that aims to enable users to explore heterogenous data as a reorderable matrix. Visception maintains reorderability, but as a side notion with an *order* VC-channel exposed for reorderable charts and focuses on providing a visual language for building nested visualizations. Slingsby et al. [34] explored the use of different layouts with editable hierarchies to incrementally answer research questions. Their approach could be described as explorative nesting of data. They define the language HiVE (Hierarchical Visualization Expression Language) which includes operations for editing, deleting, inserting and swapping different levels of a hierarchy. Visception can express similar hierarchies as HiVE, with more focus on design flexibility and support for a wider range of chart types. NodeTrix [35] enables the visualization of large networks using juxtaposition and overloading by linking adjacency matrices together. It combines the node-link diagram and the adjacency matrix into one visualization, enabling the designer to show more data and data relations using less visual space. Similarly, Domino [36] uses overloading and juxtaposition to compare and manipulate subsets across multiple datasets.

Overall, multiple specific cases of using nesting have been explored by related works. In Visception, we go beyond existing solutions by enabling the specification of nestable charts without needing to specify nesting behavior. We also enable the user to specify a wide range of different data groupings without having to modify the dataset. Finally, we include layout parameters and global properties as VC-channels, making many specific chart types expressible as configurations of a more general chart type. Visception's use of charts as building blocks follows the same line of thought as Pattison et al. [37] who proposed a "generalized layout", similar to treemaps but with more available intra-container layouts.

3. The Visception framework

In this section we detail our framework for nested visualization design. As the use of certain terms varies in the literature, we present the terminology used here in Table 1. First, in Section 3.1, we discuss how individual charts are represented and manipulated in Visception. We then introduce the VC-tree, our central data structure that enables the specification of and interaction with a nested visualization in Section 3.2.

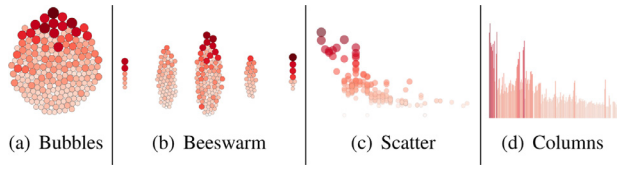


Fig. 1. Four visualizations with the same chart mapping, and different channel configurations. The first three charts (a-c) are different configurations of a *plot*, while (d) is the *columns* chart equivalent of (c).

3.1. Charts and VC-channels

Charts form the basic building blocks of Visception. A chart transforms tabular input data into output data objects, which are then used to generate graphical elements referred to as output marks.

VC-channels represent the parameters that control the layout and style of a chart. Layout VC-channels affect the shape or position of a chart's output marks directly and are hence typically specific to a particular chart, while other VC-channels affect the styling of the stroke, fill, drop shadows, etc. and are generally shared among multiple charts. As a convenience, bundles of common channels are represented as three general chart prototypes, which are then used to compose more specific charts: A *base* chart with all the common VC-channels for high-level transformations applied uniformly to the entire chart, a *stroked* chart with all VC-channels relating to the stroke, and a *filled* chart with all VC-channels relating to the fill of a chart.

Layout VC-channels allow the user to control different aspects related to the layout of a chart. For instance, a bubble chart, a beeswarm plot, and a scatter plot, instead of being available as separate chart types, can be expressed as configurations of one flexible chart as illustrated in Fig. 1a–c. In 1 a the *collision radius*, which is a scaling factor of the repulsive force between the nodes of a force layout, is set to 1.0, while in 1 c it is set to 0. 1 c also has both the x and y VC-channels mapped to data. In 1 b we see that the *force x* is higher than *force y*, causing the circles to accumulate along the vertical axis. All VC-channels available in Visception can be seen in Appendix A, Figs. A.22 and A.23.

Some attributes of a chart control global visual elements instead of the appearance of individual marks. They are usually referred to as properties in related works [2] and typically cannot be mapped to data since such an operation has limited utility. However, when nesting is introduced, the data of the parent chart can serve as input data for these properties, making them meaningfully mappable to data. For these reasons, such properties are also exposed as VC-channels, enabling an increased level of design flexibility without introducing additional complexity. For instance, we can adjust the *stroke width* of a nested chart based on the parent datum or use a categorical dimension to enable/disable effects such as drop shadows for a subset of the data.

Similar to Vega-Lite [10], we populate all VC-channels with editable default values. For example, if a *columns* chart is created, the *bar height* VC-channel is by default mapped to the (editable) *value* of 1, resulting in *N* bars with the same height. These defaults allow for rendering the chart at its intermediary stages, without requiring a complete specification of data mappings.

Furthermore, in order to be able to switch between chart types while preserving existing data mappings as much as possible, we maintain a set of semantic VC-channel equivalences between chart types as shown in Table 2. For example, if we change a *plot* to a *columns* chart (see Fig. 1 c and d), there are two equivalence groups: one containing the *plot y* and *bar height* VC-channels, and another containing *plot x* and *bar order* VC-channels. Hence, these existing data mappings can be transferred to the new chart.

Table 1
Terminology used throughout the paper.

Term	Explanation
Chart	A chart has a type and associated tabular input data that is represented by output marks.
Chart type	A type of chart within the Visception framework, consists of its own layout that is controlled by some of its VC-channels. The layout controls the shape and position of the chart's output marks.
Output marks	Graphical marks of a single chart.
VC-channel	Controls a particular aspect of the appearance or layout of a chart.
VC-channel mapping	The assignment of data (for example a dimension) to a VC-channel (for example *fill color*).
Layout space	Normalized space in which a layout is initially computed.
Parent space	Space(s) in which a chart's output marks are embedded according to the computed layout. If the node is a root node, the parent space is simply the root viewport. Otherwise, a chart-dependent region within the output marks of the parent chart.
Data object	Represents a selection from a tabular dataset. May be one of the following: 1) A single row, 2) A list of rows, 3) A list of values.
Chart input data	A set of data objects inherited from the parent VC-node. If the node is the root VC-node, the input data is the list of rows of the entire dataset.
Chart mapping	Transforms every input data object into a new set of data objects.
Chart output data	A set of data objects, used for rendering the chart, and possibly as input data to child charts.

Table 2
Examples of equivalence groups within the implementation of Visception. We followed Munzner's [5] ranking of channel types by effectiveness to determine the most significant channels.

Group	Explanation	VC-channels
Position Major	Most significant VC-channel controlling the position of the marks.	plot x, line x, bar order, circular bar order (for the charts: polar area, sectors, tubes)
Size Major	Most significant VC-channel controlling the size of the marks.	tile size, unit size, stream size, plot size, bar height
Position Minor	Secondary VC-channel controlling the position of the marks.	plot y, line y
Size Minor	Secondary VC-channel controlling the size of the marks.	bar width (columns), bar height (rows), tube height (tubes chart)

3.2. Visception tree

Visception aims to be a visual and conceptual framework for nested charts by enabling operations for building and editing a hierarchy of charts, as well as implicitly handling common nesting behaviors for multiple charts. Other works such as HiVE [34] and ATOM [13] enable setting up hierarchies of charts and data. Drawing inspiration from these previous approaches, we propose the Visception Tree (VC-tree) data structure. The VC-tree provides fine-grained control over data mappings at different hierarchical levels, and implicit handling of deformation and nesting behavior. In this section we will go into detail on how the VC-tree encapsulates a tree of charts, data mappings, and spaces.

Structure and properties: The VC-tree consists of VC-nodes which have two explicit properties: A chart type and a data mapping. The data mapping represents a chosen grouping of the chart's input data, while the chart type defines the layout and thus the transformation of the output data into output marks that make up the visual representation of the data. When a VC-node is the child of another VC-node, this corresponds to nesting one chart within another. The contents of the nested chart is then displayed within the output marks of the parent chart [32]. For example, nesting a *plot* within a *columns* chart with N bars will result in N plots, one within each bar. The left-to-right order of nodes corresponds to the Z-index, with the leftmost nodes rendered on top as shown in Fig. 2. VC-nodes can be added, moved, and deleted as also illustrated in Fig. 2.

Data mappings: Each VC-node has both input and output data, consisting of a number of data objects. The output data of a node is implicitly defined by its data mapping and input data. Depending on the type of data mapping, a data object may represent a row in the dataset, a list of rows, or a list of values.

Each node's input data corresponds to the output data of its parent (at the root of the tree, the input data is the list of all rows in the tabular dataset). The data mapping of a node turns its input data into output data as shown in Fig. 3. For each chart corresponding to a VC-node, the output data is used for generating geometric shapes. A VC-node is considered nestable, i.e., it can have children, if it has one areal output mark per output data object. In order to enable nesting without requiring a very specific dataset format, we define four types of data mappings, *dimension*, *all*, *monolith*, and *identity*, which are summarized in Table 3.

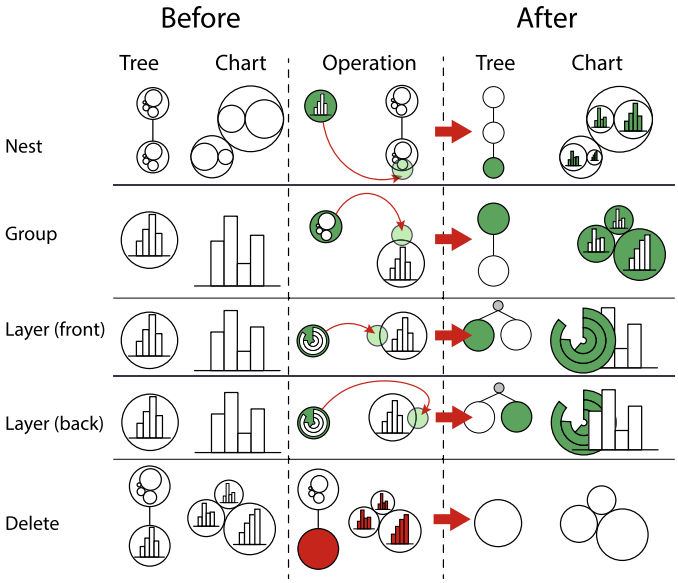


Fig. 2. This figure depicts operations that can be performed on a VC-tree, along with example charts. Note that the *move* operation is not shown here, since it corresponds to a *delete* operation followed by *nest*, *group* or *layer* operation that re-inserts the node into the hierarchy. Red represents a deleted chart, while green represents an added node

For *dimension* and *all*, the input data objects must always represent a list of rows to be applicable. The dimension mapping corresponds to grouping by a given dimension D. Thus, we partition each data object by distinct values of D. For example, if the input data is a single data object representing a list of all rows, and the mapping is a dimension D, the output data will be a set of data objects, each object representing a distinct value of D with a list of matching rows. If the data mapping is set to *all*, each input

Table 3
A summary of all possible data mappings in Visception. Not all mappings are nestable, some are only nestable if the child has a certain mapping. The cardinality of each mapping describes the size of each set of data items per parent data object. If a non-identity, non-monolith mapping results in only sets with a cardinality of 1, this is equivalent to an *identity* mapping.

Mapping	Description	Cardinality
Dimension (-,+)	Groups every input data object by a given data dimension *D*, with, creates one data object per existing (if not sparse-mapped) value of the domain of *D*. Always nestable.	between 0 and number of values in domain of *D*
All (-,+)	Ungroups input data object, creating one data object per row in the input data object. If sparse-mapped, this will create one data object per row in the root dataset. Nestable only if the child is *identity*-mapped	between 0 and number of rows in dataset
Monolith (-)	Creates *D* data objects per input data object, given *D* dimensions. Not nestable.	*D*
Identity (-)	Creates one identical data object per input data object. Always nestable.	1
(-) Sparse	Mapping excludes empty rows and data objects when nesting.	
(+) Non-sparse	Mapping includes empty rows and data objects when nesting.	

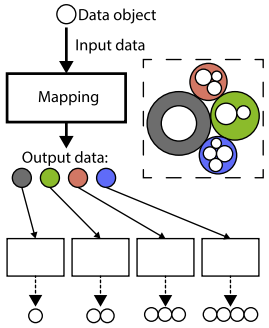


Fig. 3. A data mapping transforms every input data object into a set of new data objects. By applying this relationship recursively, each node can compute its own output data.

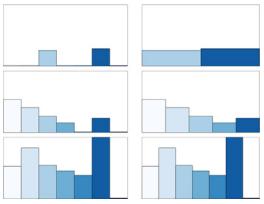
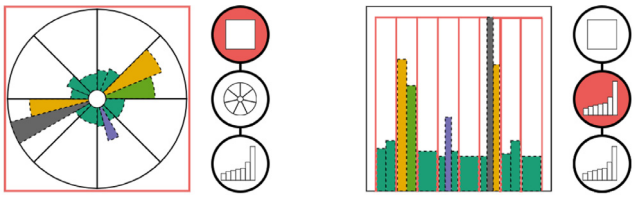


Fig. 4. On the left, we see a non-sparse mapping, which includes empty data. On the right, we have a sparse mapping showing only non-empty data objects. Observe how the non-sparse mapping horizontally arranges the innermost bars uniformly.



(a) Since the immediate parent space of the *columns* chart is an arc, the bars are deformed into arcs that fit within the parent arcs.

(b) As the parent's space is Cartesian, the bars are fit into the Cartesian coordinate system of the parent marks.

Fig. 5. Two examples of nesting with different types of parent spaces. If the parent space is deformed, each bar of a *columns* chart is also deformed as seen on the left.

data object (always representing a list of rows) is "unpacked" into a list of data objects, each containing one row. For instance, if an input data object is a list of three rows, an *all* mapping would output three data objects, each containing one row. The *identity* mapping generates a list of data objects identical to those of its parent. Creating such "dummy" data objects allows for overloading charts with extra information by nesting more charts within existing marks as shown in Fig. 10. The *monolith* mapping creates one data object for every specified numeric dimension. For the list of rows within every input data object, it generates one data object per dimension, containing the list of values of that dimension. Intuitively, the *monolith* mapping can be seen as "one mark per data dimension". An example of this mapping can be seen in Fig. 9.

In the case of nesting, the *all* and *dimension* mappings can be specified as *sparse* or *non-sparse*. A sparse mapping is the default, and will only create data objects that exist when nesting. For example, suppose a dataset is grouped by gender: male and female. Furthermore, the hair colors of both males and females include red, brown, grey, black and white, but only the female set has brown and grey hair. Then, if we apply a sparse mapping, the female set will include a mark for red and brown, but the male set will not. If the mapping is non-sparse, empty data objects are generated in all sets of data. This construct is useful when, for instance, creating bar charts on grids, and makes the nesting uniform as shown in Fig. 4. The examples shown in Figs. 13 and 14 show the implications of a sparse vs. a non-sparse mapping.

Space transformations: Since the layout of an individual chart only outputs geometric shapes into a normalized space denoted as the *layout space*, the framework must handle the rest of the nesting behavior. Existing works have already addressed the problem of deforming/transforming charts. For example, Schulz and Hadlak [15], Wickham and Hofmann [14], and Charticulator [2] transform charts from Cartesian to non-Cartesian spaces. ATOM [13], Vega [10], Vuillemot and Boy [16] compose different layouts in Cartesian spaces via nesting. Using similar methods, the layout component of the Visception framework transforms the shapes from a normalized space to fit within the parent space.With nested charts, we need to consider two different spaces in order to render the chart. Given a parent-child pair of VC-nodes, each mark of the parent node holds an inner space. We refer to this space as the *parent space*. Each chart is first defined in *layout space*, a normalized space in which the shapes of each chart are calculated. Our framework implicitly handles nesting and deformation behavior by fitting layout shapes into a parent space. Fig. 5 shows how a *columns* chart can be fit into both a Cartesian and a circular parent space.

The transformation from layout space to parent space depends on the type of the parent space, the layout shapes and the specific defined behavior for the shapes of the chart. For example, nesting a *columns* chart within a *polar area* chart corresponds to transforming rectangles to fit within arcs. If the parent space is an arc, charts nested within each arc will be either deformed or fit within the arcs. For *columns* charts it makes sense to deform the rectangles as shown in Fig. 5b, whereas for scatter plots or force-directed layouts it makes sense to deform the position, but

not the shape. These deformations can be reduced to a matter of either fitting (scaling and translating the whole shape to fit within the parent shape), or deforming 2D shapes by transforming every coordinate into the coordinate system of the parent. For example, to transform a rectangle to an arc, we simply transform the Cartesian (x, y) coordinates of each corner into the polar coordinates of the parent arc.

4. Implementation and visual builder

4.1. Implementation

Visception was implemented as a web application using VueJS for the front-end UI components, and D3 [8] for rendering the SVG. A prototype of the framework is available at https://vis.uib.no/visception/.

D3's data selections allow for creating SVG elements on a per-datum basis. This also enables creating a set of child elements for each parent element. Our implementation heavily relies on this mechanism for specifying a hierarchy of SVG groups and paths corresponding to the hierarchy of data.

The VC-tree and its VC-nodes act as a skeleton for the rest of the logic. Each VC-node has a channel manager, layout manager, guides manager, chart type, and data input. With this information, each node can compute its own layout and style. The VC-tree was realized as a simple tree data structure, with functions for moving, adding, and removing VC-nodes. Each VC-tree is tied to an SVG element, and each VC-node to a D3 selection representing the chart's output marks.

Data queries and local selections: Whenever the data mapping or chart type changes, a data query is made, and the selection of the node is updated accordingly. The data mapping and chart type of the VC-node is used to query the dataset, and thus infer the cardinality of the selection.

Layout: Whenever a VC-channel affecting the layout changes, the layout step, which itself is implemented in the form of a pipeline, is executed. The layout computes the position and shape of each mark of a node's selection. Since most charts have commonalities, we implemented a general layout pipeline where we can easily replace/insert steps for customization, but also reuse many steps across multiple chart types.

Guides: After the style or layout has changed, the guides of the chart are rendered, independent of the layout pipeline. Floating guides, such as color legends (see Fig. 9) are rendered to a group at the root of the SVG. Fixed guides such as axes (see Fig. 21) are rendered in a selection local to each node. While guides are not the main focus of this paper, they use a similar mechanism to the layout pipeline for rendering, and are deformable as seen in Fig. 21. The geometric components of the axis are transformed along with the output marks of the corresponding chart. Both axes and floating guides can be styled using VC-channels.

4.2. Visual builder

Design and components: Our visual builder uses drag and drop operations to expose a majority of the framework's functionality. An overview of the user interface is shown in Fig. 6 and the main functions provided by each of the components are summarized in Table 4.

The *data view* enables the user to drag data mappings, dimensions and aggregates. Dragging an item from the data view expresses an intent to map that item to a chart or a VC-channel and potential drop targets are immediately highlighted. The data view provides all possible data mappings and individual data dimensions. A dimension can be clicked and expanded into a set of draggable aggregates (see Fig. 6). We currently provide the following aggregation functions: *sum*, *quartile*, *quantile*, *median*, *min*, *max*, *avg*, *distinct*, and *count*. Furthermore, by dragging the respective icons, the user can indicate whether the dragged mapping should be sparse ( ) or non-sparse ( ). Dragging the tile corresponds to dragging a *monolith* mapping of the dimension. A drag and drop operation of a data mapping or dimension aggregate can express a wide range of operations as shown in Table 5.

All drag operations originating from the data view have three possible drop destinations: the *outline view*, *channels view*, or *canvas view*. Thus, the data view is placed in the center to all these views. The canvas view shows the rendered charts, and accepts both chart and channel mappings. When the Visception builder is initially opened, only the canvas and data views are visible. When a data mapping is dropped onto the canvas, the outline view and channels view appear. The outline view provides a high-level overview by showing the hierarchy of charts, and enables the

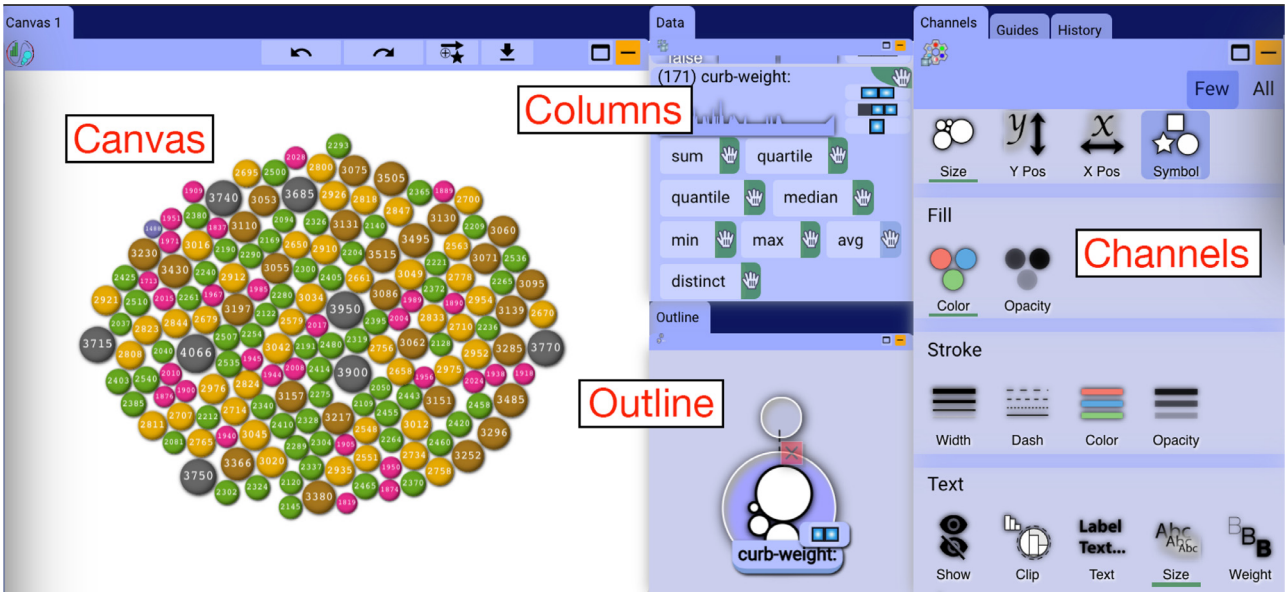
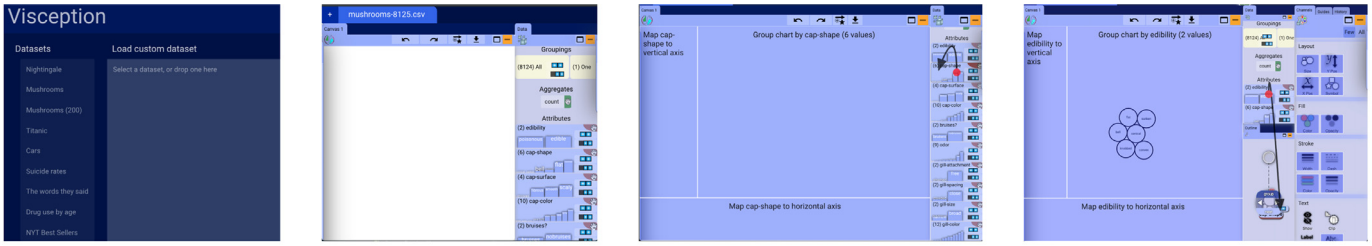


Fig. 6. A screenshot of all views exposed within Visception. Note how the Guides view is within a tab in this example. The guides view lets the user select a guide, and shows a list of VC-channels (similar to the Channels View displayed on right) that the user can edit the guide's style through.

Table 4
A summary of information each user interface component shows, and which functions it addresses. Together, these views enable the creation of nested charts, editing individual charts and accompanying VC-channels and guides.

Component	Information displayed	Functions
Canvas	Result visualization.	Receive drops, mapping data to chart or VC-channels.
Data	Data mappings, dimensions and aggregates.	Initialize drags.
Outline	Hierarchy of charts (a VC-tree) & data and chart type of each node, selected node.	Receive drops (map data to chart), rearrange hierarchy (group, nest, layer), changing chart type of node, selecting a node.
Channels	VC-channels of selected chart.	Receive drops (map data to VC-channel), edit individual VC-channels.
Guides	Guides (legends and axes) of selected chart.	Edit guide by editing channels.



(a) The user can select, or load a template csv dataset.

(b) After loading a dataset, the user can see the data dimensions, and an empty canvas.

(c) When initializing a drag, only the canvas is highlighted as a drop zone.

(d) After creating a chart, the outline and channels view are made visible. When dragging a data dimension, nodes in the outline view and VC-channels in the channels view, are highlighted as drop zones.

Fig. 7. Visception, getting started step by step.

Table 5
When dragging and dropping a data dimension, there is a limited set of available operations and outcomes. D denotes a dragged data dimension, or aggregate of an dimension. C is the selected chart, and C' is a new chart grouped by D.

Target	Area	Operation	Result
Channel	Center	Map (VC-channel)	Map D to VC-channel
Outline Node	Center	Map (chart)	Map chart to D
	Left	Layer (front)	C' layered on top of C
	Right	Layer (back)	C' layered beneath C
	Top	Group	C nested within C'
	Bottom	Nest	C' nested within C
Canvas	Center	Map	Map chart to D
	Bottom	Map (VC-channel)	Map D to C x-axis
	Left	Map (VC-channel)	Map D to C y-axis

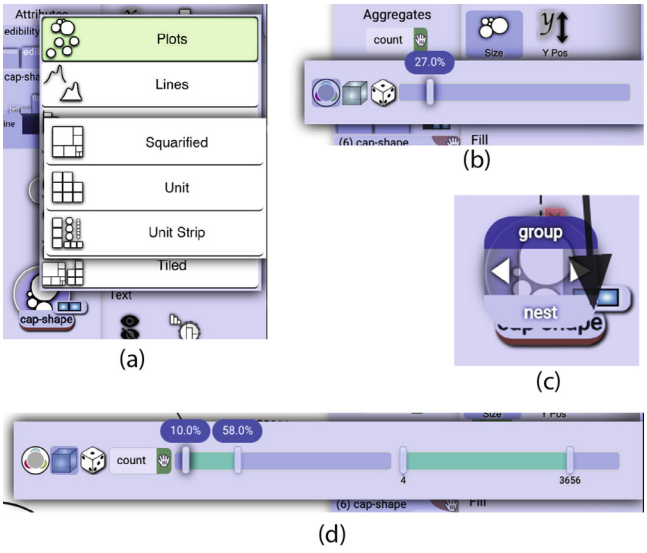


Fig. 8. Snapshots of Visception's user interface elements during an interaction session.

expression of operations such as nesting, grouping, layering (see Table 5), as well as changing chart types. When a node in the outline view is clicked, it is selected. When selected, the channels view displays all available VC-channels for a chart, and enables the user to edit and map data to individual VC-channels. For editing guides, we provide the *guides view* that allows the user to select a guide and edit its VC-channels, in the same way the VC-channels of a chart are edited. These views allow for expressing and editing a hierarchy of charts, as well as individual charts.

Example workflow: Here we demonstrate a general workflow considering the main operations of mapping data to charts, mapping data to VC-channels, editing the hierarchy of charts, and editing VC-channels.

After selecting a dataset (see Fig. 7a), the user initiates a drag operation on a data mapping. This operation highlights possible drop areas and a preview of the result will be shown (see Fig. 7c and d). If the chart is empty, the only drop area will be the visualization canvas, as shown in Fig. 7b. After the drop operation, the dropped data mapping becomes the data mapping of the chart, and the user will see a chart grouped by the given mapping (see Fig. 7d).

With a non-empty chart, two more views will appear: the outline view and the channels view, as seen in Fig. 7d. The outline view shows a tree corresponding to the current hierarchy of charts. The user can select a node by clicking it. If it is clicked again, a chart menu is shown (see Fig. 8a), enabling the user to change the chart type. When a node is selected, the channels view will display the VC-channels for that chart. For example, in Fig. 7d the channels view corresponds to a *plot*.

With four active views there are some operations to consider. If the user wants to edit an individual chart, the chart is selected by clicking it in the outline view. When that chart is selected, the channels view will display a set of editable VC-channels in the form of small labeled icons, grouped into categories. The category helps the user decide what to edit on a general level, while the icons and accompanying labels provide more specific hints. When the user has found and clicked a VC-channel, the widget for editing it pops up as seen in Fig. 8b. The widget can be a slider, a color picker or another kind of control. Undo/redo functionality allows the user to try out different controls and learn from resulting changes to the chart.

By interacting with the widget, and immediately seeing the results, the user can learn what the VC-channel does. Sequentially editing VC-channels lets the user control one aspect of the chart at a time. The user must also be able to map data to a VC-channel. When initializing a drag operation of a dimension or dimension aggregation, potential target channels will be highlighted. An example of this is shown in Fig. 7d. If the user drops a dimension on a VC-channel, a corresponding mapping is created. When a VC-channel is clicked, the user may turn a mapping on or off, and the widget will change accordingly.

For example, if a VC-channel is mapped to a dimension, the user can edit the output ranges shown in Fig. 8c (for example [0%, 100%] on the x-axis, and the domain (for example [0, 20] even though the dataset only contains [8, 20]).

When a dimension is dragged, the user can drop it on one of the areas of the node. These areas appear when the drag is initiated (see Fig. 8d). Table 5 illustrates the outcomes of a drag operation.

5. Results

Here we demonstrate a gallery of example charts created with the Visception builder. Each example is accompanied by a screenshot of the outline view displaying the corresponding hierarchy of charts. Each chart is generated by creating such a hierarchy and applying styling/mappings to one chart at a time. We selected a broad range of different datasets in order to demonstrate a wide variety of data mappings and chart hierarchies. Our generated examples cover a variety of designs and aim to demonstrate the generative expressiveness of our framework.

Nightingale's rose is an early and well-known data visualization, used by Florence Nightingale to illustrate avoidable deaths of soldiers during the Crimean war. A row in the dataset holds a month number, army size and death counts. Here we demonstrate the usefulness of nesting and the *monolith* mapping. The chart is created by leveraging the *monolith* mapping to nest the dimensions *disease, wounds, other* as a *vertical stack* inside a *polar area* chart mapped to *all*, as shown in Fig. 9.

UCI's Mushroom dataset [38] has been widely used for machine learning, and as an example dataset for visualizing categorical data. It has 22 dimensions and over 8000 rows. Each row in the dataset is one mushroom sample. Here we will see how the *identity* mapping can be used to decorate output marks at different levels of nesting. First, we consider 200 samples of mushroom (see Fig. 10) representing the hierarchy *gill size* → *stalk-surface*. Within each container, we have a *unit* chart representing all rows in the dataset (one square per row). Within each unit, we nest an *identity*-mapped (one datum per parent datum) *plot* where the *symbol* is mapped to *cap-shape*. The *identity* mapping allows us to overload the unit squares with more information, in this case the *cap-shape*.

For the second visualization, we show all 8124 rows (samples) of mushrooms, by displaying the hierarchy *cap-surface* → *cap-shape* using the *squarified* chart, with its *size* VC-channel mapped to *count* (the count of rows per aggregation). Inside each square is a *unit* chart, showing one unit per mushroom. Layered over the unit chart, an *identity*-mapped *plot* (lower left node) has its *symbol* VC-channel mapped to the cap surface, and helps show the cap shapes. With both layering and nesting available, we can overload charts with great flexibility as shown in Fig. 11.

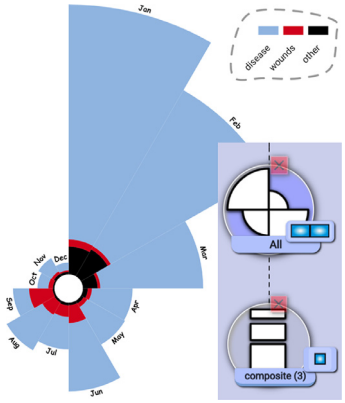


Fig. 9. Nightingale's Rose.

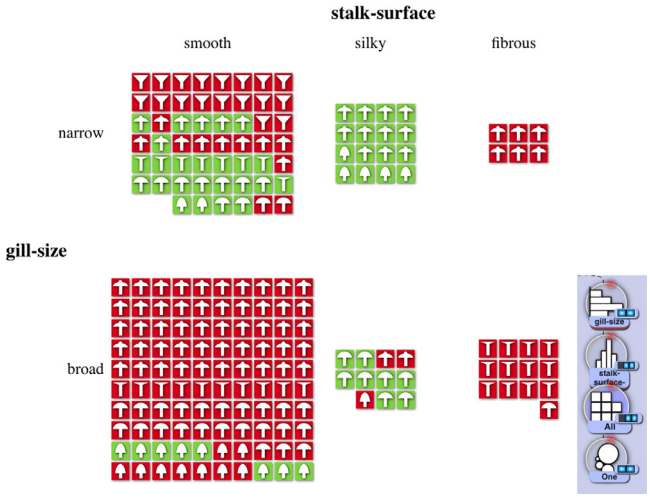


Fig. 10. 200 samples of mushroom.

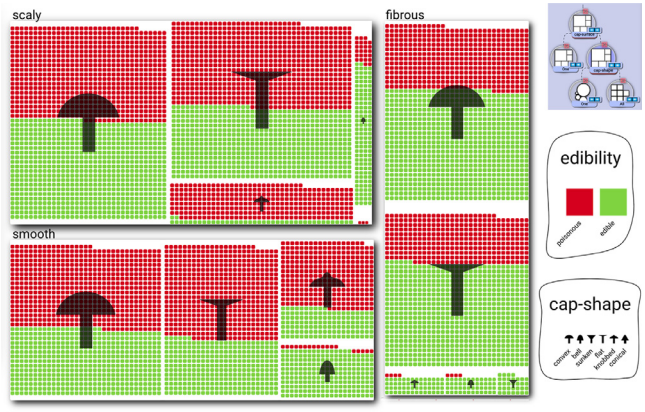


Fig. 11. 8125 samples of mushroom.

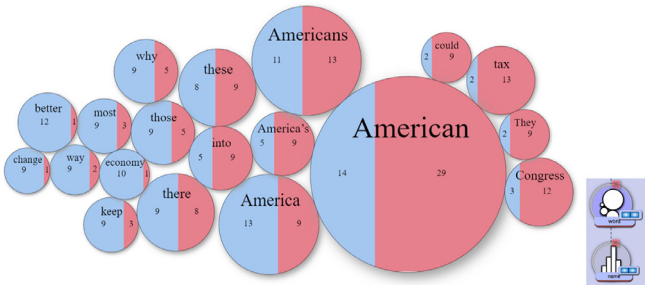


Fig. 12. Recreation of "At the National Conventions, the Words They Used".

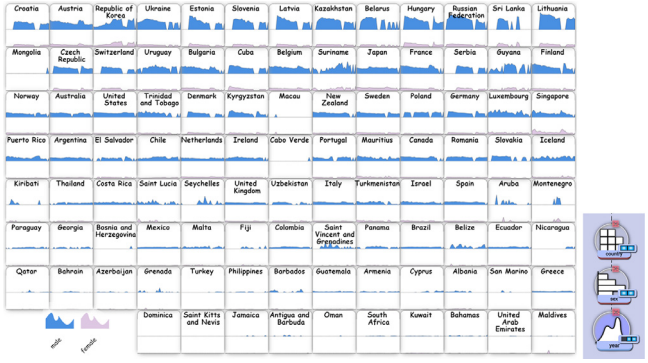


Fig. 13. Suicides per country, by gender, over time.

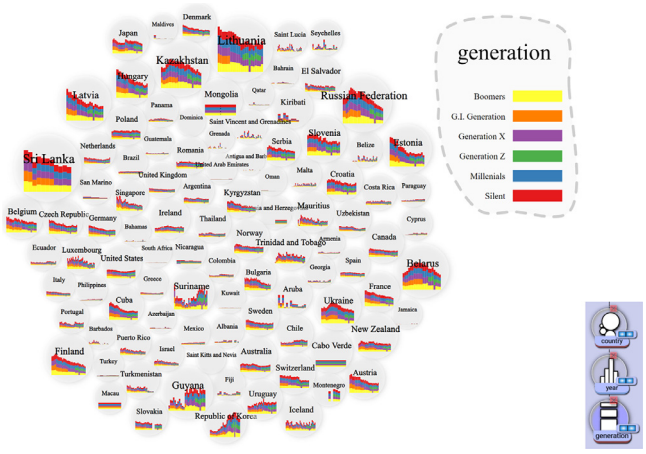


Fig. 14. Suicides per country over time, by generation, over time.

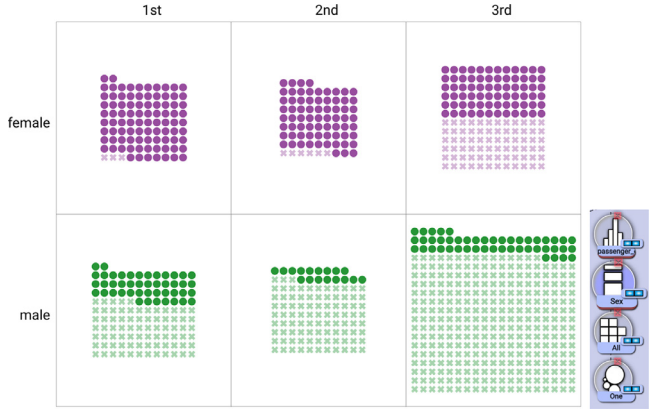


Fig. 15. A recreation of Fig. 10 of the ATOM paper [13].

"At the National Conventions, the Words They Used" was published by the New York Times in 2012, illustrating how much different words are used by different political parties. We create a similar visualization based on data from the 2016 presidential election. Each row in the dataset represents the following: *(word, name, mentions, Trump, Obama)*. Here we will see how the *clip* VC-channel, combined with nesting can "slice" the circles. First, we nest a *columns* chart within the circles, and enable the *clip* VC-channel, and use the *bounds* VC-channel to stretch the columns to properly cover their parent shapes. The final result is shown in Fig. 12.

Kaggle's suicide rate dataset has many dimensions, such as *(country, year, sex, suicides/100k pop, generation,...)*. In the following two examples, we explore this dataset and demonstrate how nesting can be utilized to generate information-rich small multiples. For the first example (Fig. 13) we investigate suicide rates per country, by gender over time. First, we create a *unit* chart mapped to *country*. We sort the chart by *avg(suicides)*. Each unit is subdivided by sex using a *rows* chart. Within each *rows* chart, an *area* chart is grouped by *year* (non-sparse), with *avg(suicides)* mapped to *y*, and *year* mapped to *x*. The data mapping of the *area* chart uses the parent datum to show *gender*. The non-sparse grouping of the *area* chart creates empty data points for years with missing data points, thus exposing this in the visualization.

Next, we look at suicide rates for the different generations per country, over time (see Fig. 14). At the root we have a force-directed layout with one node per country. Next, we nest *year* within the root chart, and set the chart type to *columns*. Finally, we subdivide the bars by nesting a *vertical stack* chart (mapped to *generation*) within it. In contrast to Figs. 13 and 14 uses a sparse mapping. We show this example to demonstrate the significance of whether or not empty data items are included in a nested chart.

The Titanic dataset [39] shows how many passengers perished, and how many survived. Each row represents a passenger. In the following examples, we demonstrate how the combination of nesting and the *unit* chart enables the visualization of both the global patterns and individual details. Fig. 15 shows a recreation of Fig. 10 of the ATOM paper [13], depicting survivors of the Titanic [39]. Here we see a faceting by sex and class, with a centered *unit* layout. By using nesting, we nest a *plot* within the units, mapping the *symbol* and *opacity* to the *survival* dimension. We use the color mapping to display gender.

Now we wish to investigate the distribution of survivors, by gender and across different age groups. We do this in the form of a "unit stream" as shown in Fig. 16, by age. This chart is created with one column for every age bin, then nesting a *unit* chart within the *columns* chart. The *unit* chart is centered vertically, and ordered by *sex-survival*. Finally, we nest a *plot* within the *unit* chart, and map *fill color* to *gender*, *fill opacity* and *symbol* to *survival status*.

Next, we split up the unit stream by gender. The root of the visualization is a *columns* chart, creating one column for every age bin. We subdivide by *sex* by creating a *vertical stack* within each of the age ranges. Then, we nest a *unit* chart within the *vertical stack*. The unit chart is sorted by *survival*. To customize the symbols, we nest a *plot* within it, and map the *symbol* and *opacity* VC-channels to *survival*. The result is shown in Fig. 17. The only difference between this figure and the previous is that there is one *rows*

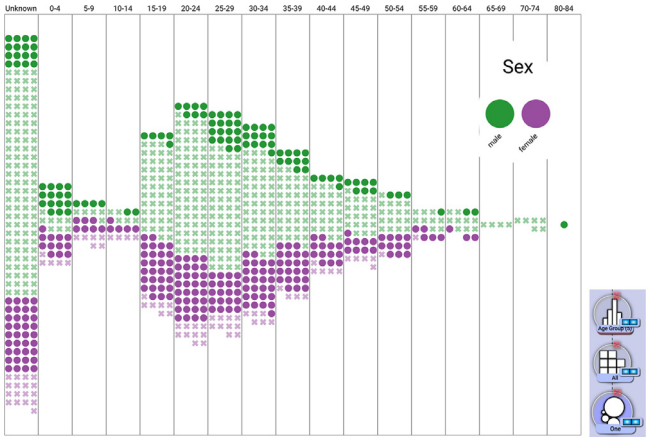


Fig. 16. A "unit stream", by age.

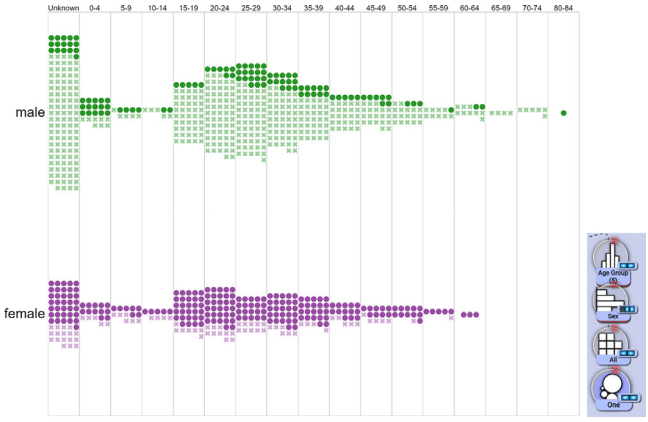


Fig. 17. Two "unit streams", by gender, then age.

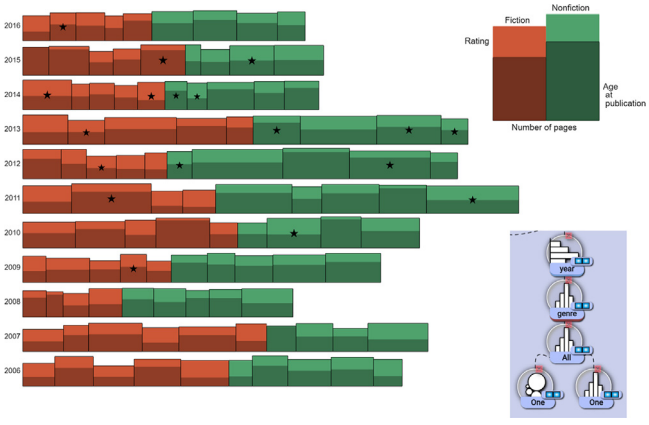


Fig. 18. An approximation of the Best Bookshelf [40] visualization.

chart inserted into the hierarchy, above *all*, expressing the "group by gender" operation.

The Best Bookshelf [40] visualization displays a wide range of dimensions for the book best sellers dataset. Every row in the dataset represents a single book publication and related dimensions such as *(year, genre, title, author, author age,...)*. We recreate this visualization as shown in Fig. 18 by utilizing nesting, layering and the *identity* mapping. Each square represents a book, with a width representing the number of pages, and the height representing the average rating of the book. Within each square, the proportion of darkened area indicates the age of the author at the time of publication. The data is faceted by creating a *rows* chart mapped to *year*. Within each year, we subdivide by genre with a *columns* chart. Finally, we create a *columns* chart mapped to all (one mark per row) where each column represents one book. The width of the bars is mapped to the *numPages* dimension. To generate the age indication, as well as the best seller star, we nest single marks within each bar using an *identity* mapping. For the age indicator, we map the *age* dimension to the *height* VC-channel. The stars are created with a *plot* with the *symbol* set to star, and *isBestSeller* mapped to *size*, setting it to 0 for False, and a non-zero value for True. We could overload the squares to show more dimensions using the nesting mechanism.

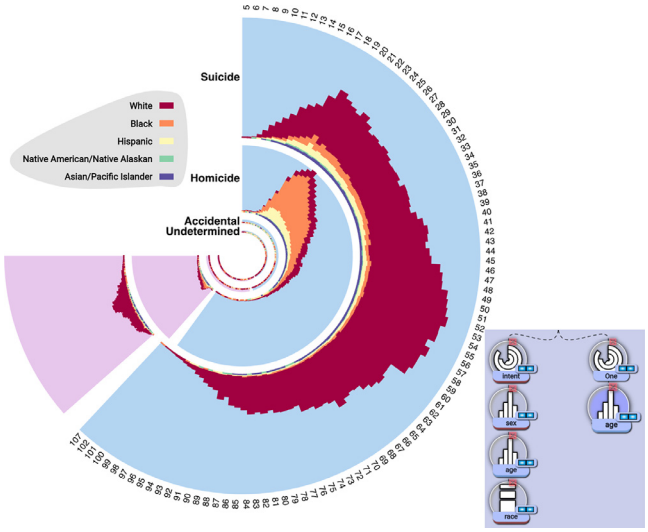


Fig. 19. Gun crime broken down by *intent, gender, race*.

FiveThirtyEight's Gun Crime dataset [41] contains over 100,000 gun crime incidents from 2012 to 2014. Each incident is represented as a row: *(year, month, intent, age,...)*. Here we demonstrate deformation behavior, as well as layering and tweaking of a VC-channel to fit a series of labels along a single arc. At the root node we create one circular row for each *intent*. Each row is divided by *sex*, by creating a *columns* chart mapped to *sex*. We then map the aggregate *count* to the *size*. Within each gender subdivision, we subdivide again by *age*, using a *columns* chart. Finally, we nest a *vertical stack* mapped to race. To generate labels along the largest outermost arc, we create a new *identity*-mapped *tubes* chart, nest all ages within it as bars, and fit the *tubes* chart to match up with the largest arc. We do this fitting by tweaking the *start angle* and *width* VC-channels. The resulting visualization is shown in Fig. 19. The *identity*-mapped chart is used to provide a polar space in which the *columns* by *age* are laid out.

The Cars dataset is commonly used as a basis for example visualizations of high dimensional data. Each row represents a car and a large number of accompanying dimensions. Here, we demonstrate different chart type nestings, representing the same data hierarchy: *engine-type* → *all*. All of these charts can be toggled between by swapping the root chart type. The chart types used as root are the following: *columns, unit, squarified, sectors, tubes*. It would also be possible to change the chart type of the lower node. This allows for interactively exploring and prototyping new designs. Some example charts are shown in Fig. 20.

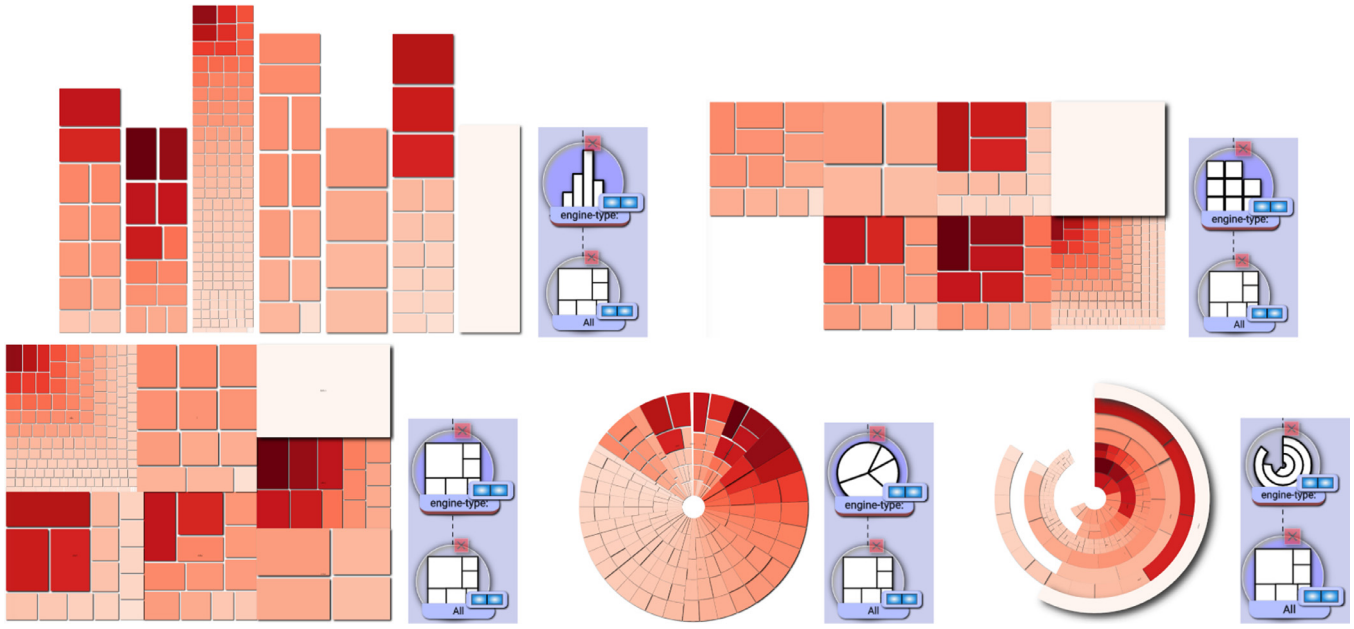


Fig. 20. Variations of the data hierarchy *engine-type* → *all*. Swapping between these variations only requires the user to change the chart type. Mapped VC-channels are transferred, thus the style is transferred.

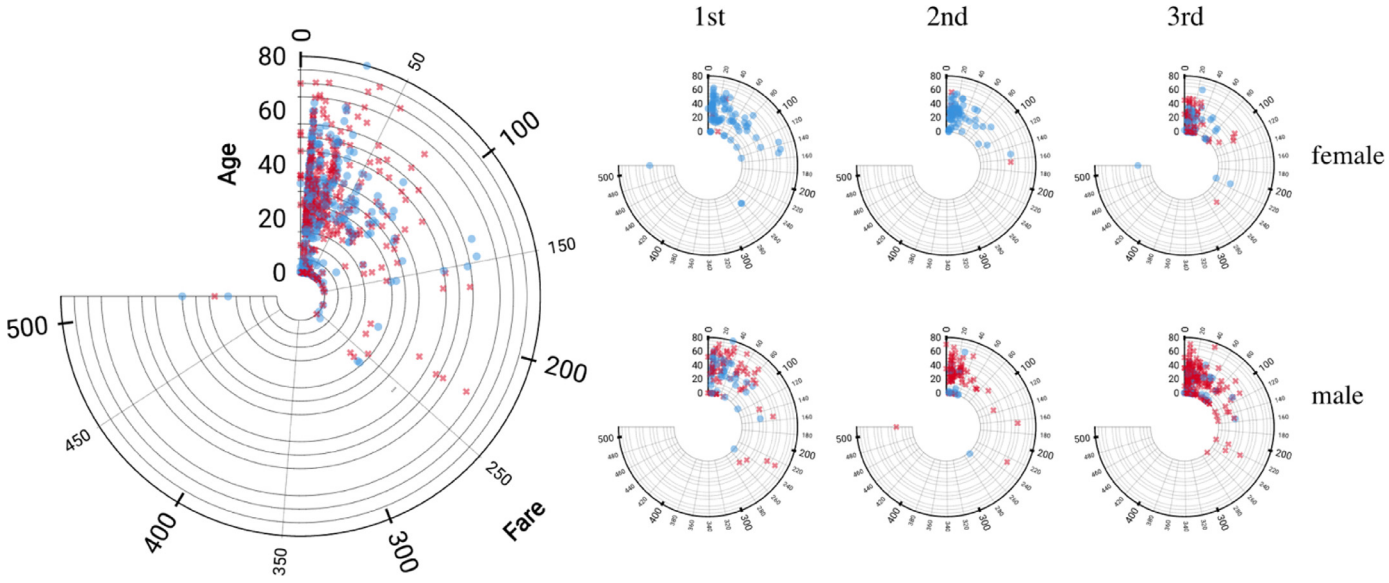


Fig. 21. Titanic survivors, faceted by class and gender, showing a polar plot with *Age* mapped to *y*, and *Fare* mapped to *x* for every category.

Axes are available when a chart has its axis VC-channel mapped to data value. Here, we aim to demonstrate that axes are available for non-nested, nested, Cartesian and non-Cartesian layouts. Fig. 21 shows a plot with *fare* mapped to *x*, and *age* mapped to *y*, both for all entries, as well as for every *class-gender* permutation. The axes are deformable, are thus nestable and flexible similarly to the charts themselves.

These examples show a range of different expressions that can be achieved via nesting. We have demonstrated implications and uses of different kinds of data mappings and charts used in combination. The *identity* mapping allows for overloading charts with more information and the use of charts as containers for other charts. By combining custom mappings and nesting, a great variety of visualizations can be expressed.

6. Discussion and limitations

Comparison to other visual builders: Satyanarayan et al. [42] recently proposed a set of criteria to evaluate visual builders, and compare the three most feature rich, recent works: Data Illustrator [1], Charticulator [2] and Lyra [21]. Visception focuses on achieving expressiveness by nesting charts. We aim to show that features such as glyph composition, coordinate systems and data scoping can all be expressed by leveraging nesting functionality.

Table 6
(W=what, H=how) Summary of the Visception visual builder system components. To compare to Lyra, Data Illustrator and Charticulator we recommend the user to view this table next to the table presented in the work by Satyanarayan et al. [42]. Visception achieves many of these features through the use of nesting, while in other systems these features are more explicitly specified. Furthermore, in Visception the data scoping of a chart is implicitly defined by the data mapping of the chart, abstracting the specifics of this task away from the user.

Component	Visception
Marks Instantiation and Customization	W: Predefined marks only, referred to as chart type H: Start with default, click chart in outline view to change. Must choose data grouping before seeing marks.
Glyph Composition	H: Combine predefined marks into glyphs as layers. Such layers can be nested within existing charts, enabling a wide range of combinations and mappings.
Path Points and Path Segments	W: Map x and y to data dimensions. H: Drag data dimension or aggregate to the x or y VC-channel of the chart, if the chart type is *line*, *area* or *stream*.
Links between Glyphs	W: Limited availability. H: For example, a line chart can be layered under a plot, with identical x/y data mappings linking the glyphs. However, for future work we aim to introduce a linking tool similar to that of Charticulator
Data Scoping for Glyphs	W: Custom dimensions and groupings: *all*, *identity*, *monolith*, sparse and non-sparse grouping modes. H: An *all* dimension to create one mark per tuple, a *identity* dimension to create a single mark representing the data selection of the parent node (if root, the entire dataset). Grouping by a dimension implicitly aggregates the data by that dimension. Groupings by a dimension are by default sparse, i.e they will not show empty marks when nesting. If non-sparse, empty data is created to populate each nested viewport. To create one glyph per dimension, use the *Monolith* grouping of a numeric dimension.
Mapping Data Values to Visual Properties	H: Drag dimension or aggregate from data view and drop on channel, or select from menu. Available mappings depend on the data scope of the selected chart.
Scales	W: Scales for categorical, temporal, and numerical data H: Implicitly created when mapping data to channels (visual properties)
Axes and Legends	H: Created when a data binding is applied. Hidden by default if chart is nested, except for color mappings. Each legend/axis is customized with channels, the same way a chart is customized.
Relative Layout	H: Use the *Bounds* channel to free-form position a chart in normalized space.
Layout in a Collection	H: Marks are always positioned according to the layout of the selected chart type. Each chart has a set of visual channels, and in most cases a set of *layout* channels, some mappable to data (for example the x and y position of a line chart).
Nested Layout	H: If the chart is nestable (appropriate data and chart type), another chart may be nested within it. Since separate aspects of layouts can be mapped to any eligible data dimension or aggregate, this implicitly changes the space in which the nesting is done. With the *bounds* channel we can edit the bounds of a chart in a normalized space. If the parent space is deformed (i.e an arc) the geometry of the child chart is deformed accordingly (for example, a square to an arc).
Coordinate Systems	W: Cartesian, Polar, extensible to others. H: Each chart is seen as a set of 2D shapes, these shapes are simply transformed to fit within the given parent shape. As such, an arc can deform a rectangle to fit within itself. Each chart type must specify *how* it is to be deformed.

Table 6 summarizes the Visception visual builder in the terms proposed by Satyanarayan et al. and is meant to be compared with Table 1 in their paper. By comparing Visception to the other systems in this manner, it can be seen that Visception achieves many features via nesting and the accompanying data grouping.

Framework: While Visception provides a number of standard charts, there are several types of more complex or specialized types of visualizations that are currently not integrated. Our implementation is designed with tabular data in mind. We support categorical and numerical data, but currently do not provide specific operations for specifying categorical dimensions as ordinals, as well as specialized aggregations for time-oriented data. We also do not provide explicit support for visualizations targeted at graph and network data such as node-link diagrams, and some other common visualization techniques such as parallel coordinates or parallel sets are also currently not implemented. However, we believe that they fit well within our architecture and plan to add these and other relevant chart types in the future. Links and bands between marks of different charts should also be possible to add to the framework, but it proved difficult to find ways to make bands and links work across different levels of nesting, especially given the nature of SVG group hierarchies. The challenge of increasing expressiveness is not in adding the charts themselves, but in adding general structures to support different kinds of charts so that they can leverage the existing nesting behavior.

Rendering and layout calculation: When a chart is fully reflowed, its layout is calculated before it is applied to the corresponding SVG paths. With nesting introduced, it is crucial to only apply the necessary updates to the chart and its child charts. For example, editing the *fill color* of a chart should not cause a reflow of its children. Redundant reflows break the fluidity of the interaction. We use throttling to keep the system responsive, but additional threading could further improve the situation. We rarely encountered performance problems with the SVG rendering itself, except when filter effects like drop shadows are active. This is to be expected, though it would be beneficial to disable filters effects upon zooming and interaction. Specifying which step of the pipeline a VC-channel should trigger has removed a great number of redundant full reflows. Furthermore, we noticed that complex nested visualizations expose some deficiencies in SVG support across different platforms and applications. This is mainly due to numeric instability arising from deeply nested SVG elements. The examples in this paper are screenshots taken in FireFox (version 72), which has not shown these issues.

Data querying: If the dataset is too large, there are potential performance concerns with regards to both data querying, and rendering of the chart itself.

For example, the suicide dataset had about 100,500 rows, and the aggregation at the deepest level *(intent, sex, age, race)* took about 3 seconds to compute on a 2.2 GHz Quad-Core Intel Core i7 with 16GB memory.

The data querying issue could be resolved by using a server for queries. However it is always preferable that the program can be used without a server. Currently, we lazily compute aggregations as well as their domains when querying the data. Whenever an aggregate is retrieved for the first time, all aggregates for that column are computed and cached. We used arrow.js to store the

data in a columnar format, and a recursive data structure to generate queries for each VC-node. Taking a progressive visualization [43] approach might help in addressing this.

Visual builder user interface: The outline view tree has scalability issues if the hierarchy of trees gets too wide or too deep. To counter this, the outline view (and other windows) can be made into a floating window. However, for future work we would like to incorporate more scalable techniques for showing this.

7. Conclusion

In this paper we, presented our framework for nested visualization design. We introduced the VC-tree as a unified framework for the creation and manipulation of nested visualizations and demonstrated how it can be used to flexibly specify a wide variety of data groupings and visual mappings. We showed how the VC-tree provides fine-grained control over data mappings at different hierarchical levels, while providing implicit handling of deformation and nesting behavior. Based on our framework, we contributed a visual builder that exposes the full expressiveness of the framework through a user interface. To demonstrate the expressiveness of our approach, we provided a wide range of examples demonstrating various features achievable via nesting.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Yngve Sekse Kristiansen: Conceptualization, Methodology, Software, Writing - original draft. **Stefan Bruckner:** Conceptualization, Supervision, Resources, Writing - review & editing, Project administration.

Acknowledgments

The research presented in this paper was supported by the MetaVis project (#250133) funded by the Research Council of Norway.

Appendix A. Overview of Charts and VC-channels

We present the full set of VC-channels in the current implementation of Visception in Figs. A.22 and A.23. Fig. A.22 shows all VC-channels that are common to multiple charts, while Fig. A.23 shows all chart types and the VC-channels unique to each type.

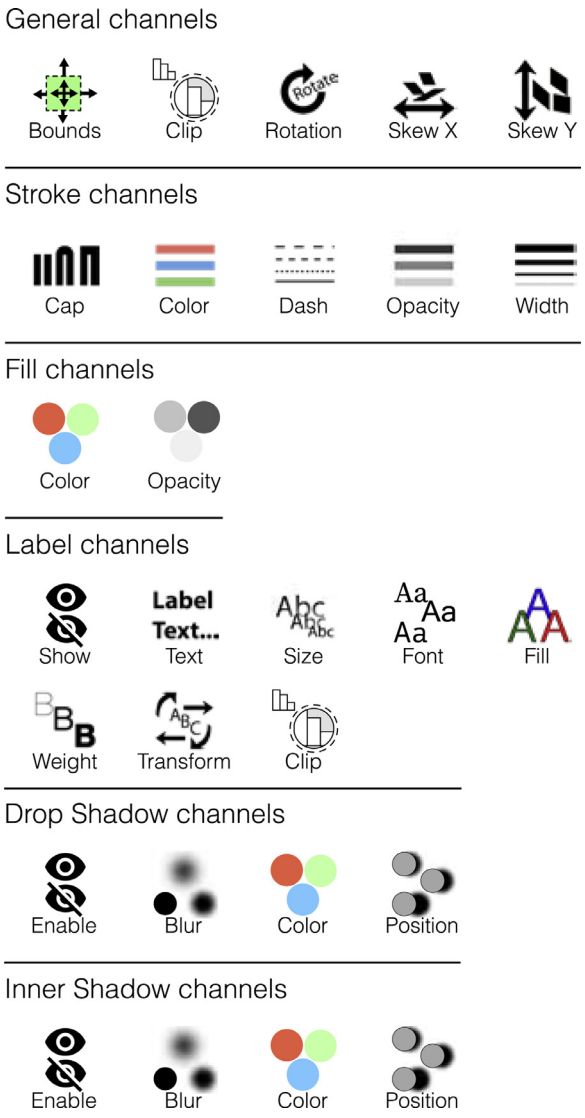


Fig. A.22. All general VC-channels within Visception. These exist for all charts, with the exception of label channels not existing for *area* and *line* charts, and fill VC-channels not existing for line charts.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.cag.2020.08.007.

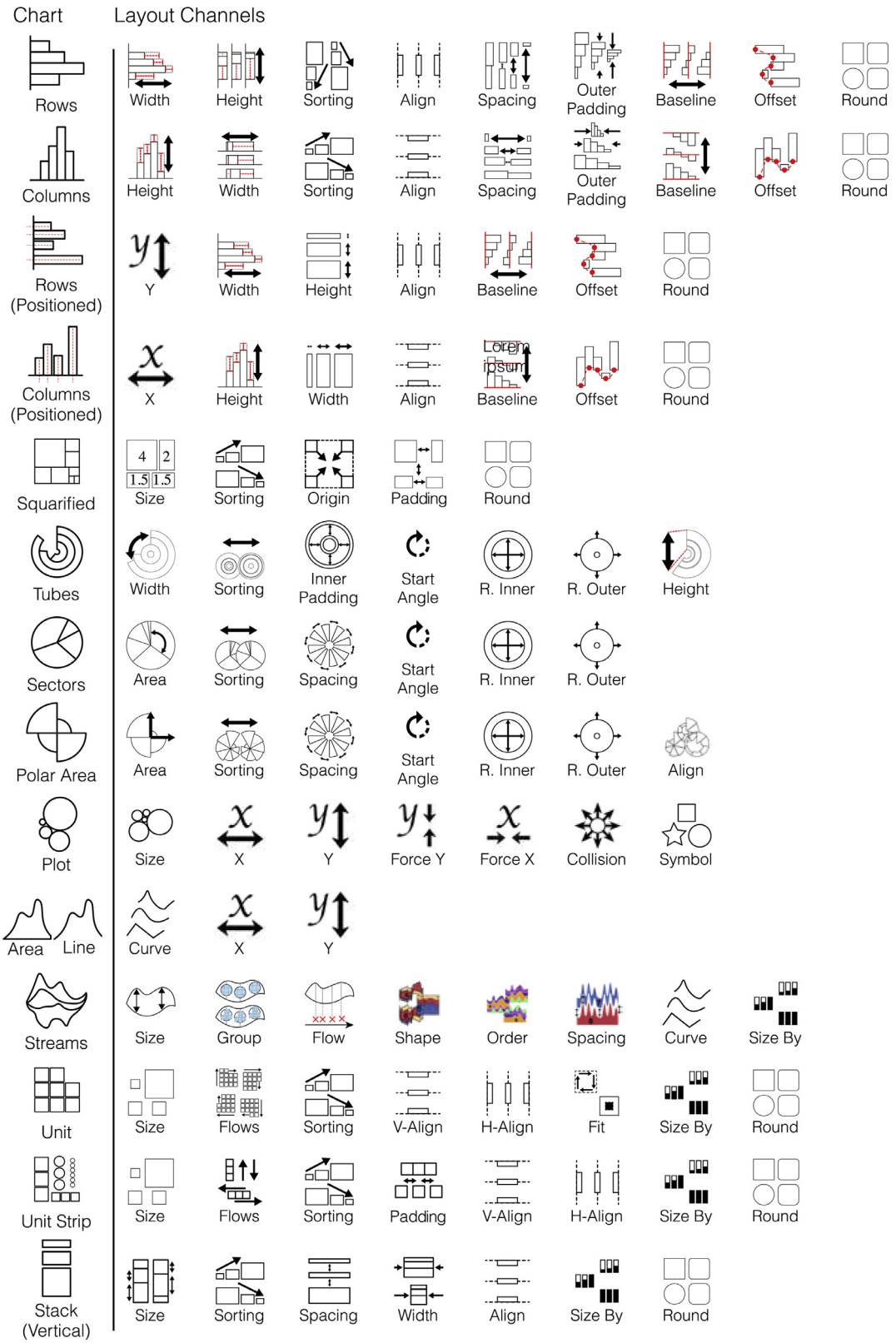


Fig. A.23. All chart types, and VC-channels unique to that chart type within the current implementation of Visception. Each icon represents a channel, and each VC-channel controls the layout or some property unique to that chart, or charts with similar output marks.

References

[1] Liu Z, Thompson J, Wilson A, Dontcheva M, Delorey J, Grigg S, et al. Data illustrator: augmenting vector design tools with lazy data binding for expressive visualization authoring. In: Proc. CHI; 2018. p. 123:1–123:13. doi:10.1145/3173574.3173697.

[2] Ren D, Lee B, Brehmer M. Charticulator: interactive construction of bespoke chart layouts. IEEE Trans Vis ComputGraph 2019;25(1):789–99. doi:10.1109/TVCG.2018.2865158.

[3] Bertin J. Semiology of graphics. University of Wisconsin Press; 1983. ISBN 0299090604.

[4] Wilkinson L. The grammar of graphics (Statistics and Computing). Springer-Verlag New York, Inc.; 2005. ISBN 0387245448.

[5] Munzner T, Maguire E. Visualization analysis and design. CRC Press; 2015. doi:10.1201/b17511. ISBN 9781498759717.

[6] Heer J, Card SK, Landay JA. Prefuse: a toolkit for interactive information visualization. In: Proc. CHI; 2005. p. 421–30. doi:10.1145/1054972.1055031.

[7] Bostock M, Heer J. Protovis: a graphical toolkit for visualization. IEEE Trans Vis Comput Graph 2009;15(6):1121–8. doi:10.1109/TVCG.2009.174.

[8] Bostock M, Ogievetsky V, Heer J. D3: data-driven documents. IEEE Trans Vis ComputGraph 2011;17(12):2301–9. doi:10.1109/TVCG.2011.185.

[9] Wongsuphasawat K, Moritz D, Satyanarayan A, Heer J. Vega: a visualization grammar. https://vega.github.io/; 2013.

[10] Satyanarayan A, Moritz D, Wongsuphasawat K, Heer J. Vega-lite: a grammar of interactive graphics. IEEE Trans Vis ComputGraph 2017;23(1):341–50. doi:10.1109/TVCG.2016.2599030.

[11] Li G, Tian M, Xu Q, McGuffin MJ, Yuan X. Gotree - a grammar of tree visualizations. http://go-tree.info; 2020.

[12] Schulz H, Akbar Z, Maurer F. A generative layout approach for rooted tree drawings. In: Proc. IEEE PacificVis; 2013. p. 225–32. doi:10.1109/PacificVis.2013.6596149.

[13] Park D, Drucker SM, Fernandez R, Elmqvist N. Atom: a grammar for unit visualizations. IEEE Trans Vis ComputGraph 2018;24(12):3032–43. doi:10.1109/TVCG.2017.2785807.

[14] Wickham H, Hofmann H. Product plots. IEEE Trans Vis ComputGraph 2011;17(12):2223–30. doi:10.1109/TVCG.2011.227.

[15] Schulz H-J, Hadlak S. Preset-based generation and exploration of visualization designs. J Vis Lang Comput 2015;31:9–29. doi:10.1016/j.jvlc.2015.09.004.

[16] Vuillemot R, Boy J. Structuring visualization mock-ups at the graphical level by dividing the display space. IEEE Trans Vis ComputGraph 2018;24(1):424–34. doi:10.1109/TVCG.2017.2743998.

[17] Ahlberg C, Wistrand E. IVEE: an environment for automatic creation of dynamic queries applications. In: Proc. CHI; 1995. p. 15–16. doi:10.1145/223355.223381.

[18] Roth SF, Lucas P, Senn JA, Gomberg CC, Burks MB, Stroffolino PJ, et al. Visage: a user interface environment for exploring information. In: Proc. IEEE InfoVis; 1996. p. 3–12. doi:10.1109/INFVIS.1996.559210.

[19] Aiken A, Chen J, Stonebraker M, Woodruff A. Tioga-2: a direct manipulation database visualization environment. In: Proc. international conference on data engineering; 1996. p. 208–17. doi:10.1109/ICDE.1996.492109.

[20] Stolte C, Tang D, Hanrahan P. Polaris: a system for query, analysis, and visualization of multidimensional relational databases. IEEE Trans Vis ComputGraph 2002;8(1):52–65. doi:10.1109/INFVIS.2000.885086.

[21] Satyanarayan A, Heer J. Lyra: an interactive visualization design environment. Comput Graph Forum 2014;33(3):351–60. doi:10.1111/cgf.12391.

[22] Ren D, Hllerer T, Yuan X. iVisDesigner: expressive interactive design of information visualizations. IEEE Trans Vis ComputGraph 2014;20(12):2092–101. doi:10.1109/TVCG.2014.2346291.

[23] Kim NW, Schweickart E, Liu Z, Dontcheva M, Li W, Popovic J, et al. Data-driven guides: supporting expressive design for information graphics. IEEE Trans Vis ComputGraph 2017;23(1):491–500. doi:10.1109/TVCG.2016.2598620.

[24] Nacenta MA, Méndez GG. iVoLVER: a visual language for constructing visualizations from in-the-wild data. In: Proc. ACM international conference on interactive surfaces and spaces; 2017. p. 438–41. doi:10.1145/3132272.3132299.

[25] Schulz H, Hadlak S, Schumann H. The design space of implicit hierarchy visualization: a survey. IEEE Trans Vis ComputGraph 2011;17(4):393–411. doi:10.1109/TVCG.2010.79.

[26] LeBlanc J, Ward MO, Wittels N. Exploring n-dimensional databases. In: Proc. IEEE visualization; 1990. p. 230–7. doi:10.1109/VISUAL.1990.146386.

[27] Wang W, Wang H, Dai G, Wang H. Visualization of large hierarchical data by circle packing. In: Proc. CHI; 2006. p. 517–20. doi:10.1145/1124772.1124851.

[28] Fruchterman TMJ, Reingold EM. Graph drawing by force-directed placement. Softw Pract Exp 1991;21(11):1129–64. doi:10.1002/spe.4380211102.

[29] Baudel T, Broeksema B. Capturing the design space of sequential space-filling layouts. IEEE Trans Vis ComputGraph 2012;18(12):2593–602. doi:10.1109/TVCG.2012.205.

[30] Parker G, Franck G, Ware C. Visualization of large nested graphs in 3D: navigation and interaction. J Vis Lang Comput 1998;9(3):299–317. doi:10.1006/jvlc.1998.0086.

[31] Elmqvist N, Do T-N, Goodell H, Henry N, Fekete J-D. ZAME: interactive large-scale graph visualization. In: Proc. IEEE PacificVis; 2008. p. 215–22. doi:10.1109/PACIFICVIS.2008.4475479.

[32] Javed W, Elmqvist N. Exploring the design space of composite visualization. In: Proc. IEEE PacificVis; 2012. p. 1–8. doi:10.1109/PacificVis.2012.6183556.

[33] Loorak MH, Perin C, Collins C, Carpendale S. Exploring the possibilities of embedding heterogeneous data attributes in familiar visualizations. IEEE Trans Vis ComputGraph 2017;23(1):581–90. doi:10.1109/TVCG.2016.2598586.

[34] Slingsby A, Dykes J, Wood J. Configuring hierarchical layouts to address research questions. IEEE Trans Vis ComputGraph 2009;15(6):977–84. doi:10.1109/TVCG.2009.128.

[35] Henry N, Fekete J-D. NodeTrix: a hybrid visualization of social networks. IEEE Trans Vis ComputGraph 2007;13(6):1302–9. doi:10.1109/TVCG.2007.70582.

[36] Gratzl S, Gehlenborg N, Lex A, Pfister H, Streit M. Domino: extracting, comparing, and manipulating subsets across multiple tabular datasets. IEEE Trans Vis ComputGraph 2014;20(12):2023–32. doi:10.1109/TVCG.2014.2346260.

[37] Pattison T, Vernik R, Phillips M. Information visualisation using composable layouts and visual sets. In: Proc. asia-Pacific symposium on information visualisation; 2001. p. 1–10.

[38] Schlimmer J. 1987. https://archive.ics.uci.edu/ml/datasets/mushroom Accessed: 2019-11-07.

[39] https://www.kaggle.com/hesh97/titanicdataset-traincsv Accessed: 2019-11-07. 2018.

[40] Kim T. 2019. http://tany.kim/best-bookshelf Accessed: 2019-11-07.

[41] Casselman B. 2016. https://github.com/fivethirtyeight/guns-data Accessed: 2019-11-07.

[42] Satyanarayan A, Lee B, Ren D, Heer J, Stasko J, Thompson J, et al. Critical reflections on visualization authoring systems. IEEE Trans Vis ComputGraph 2020;26(1):461–71. doi:10.1109/TVCG.2019.2934281.

[43] Angelini M, Santucci G, Schumann H, Schulz H-J. A review and characterization of progressive visual analytics. Informatics 2018;5. doi:10.3390/informatics5030031.