UNIVERSITY OF BERGEN

Department of Informatics

MASTERS THESIS

A Visual Language for Nested Visualization Design

Author: Yngve Sekse Kristiansen

Supervisor: Stefan Bruckner

Abstract

Significant progress has been made in the field of information visualization. Many programming libraries (like D3) enable the creation of almost any 2D visualization. However, this power of expression is not available for non-coders. In this thesis we show how non-coders can create both simple and complex visualizations using only drag & drop operations. We define a data structure (Visception Tree) that can represent arbitrarily nested and layered charts. This data structure can define a hierarchy of charts embedded within one another, or layered side by side, or a combination of the two. Each chart can be edited separately and intuitively, and selecting a chart is done in an outline view - similar to a file view. Such simple ways of interaction are made possible by the Visception Tree being easily mappable to user interface actions as well as being flexible enough to encapsulate arbitrary hierarchies and layerings of charts. The viability of these ideas is demonstrated by showing how some complex visualizations can be made with just a few drag & drop operations - enabling the creation of visualizations in just a few minutes.

Acknowledgments

The original project is far different from what the end result is. I thank my advisor, Professor Stefan Bruckner for guiding it to become what it is today. Many complex, confusing, subtle problems became clearer throughout our discussions. Without his guidance this project would not have been possible.

Contents

Chapter 1

Introduction

1.1 Motivation

Munzner [38] defines information visualization as a subfield of visualization where the visual encoding is chosen by the designer. With such visualizations we can explore data and gain insight in ways that are difficult (if not impossible) without visualizations. However, creating such visualizations is not trivial and usually requires training or programming skills. Creating visualizations is usually done by coding. Most people who might use visualizations are not able to create the visualizations they need. A system that enables people to create their own visualizations without requiring expert help or extensive training is needed. This problem is partially solved by existing solutions, yet they fail to provide the freedom and power of expression that is achieved by programming toolkits.

Problem and Contribution

Most interactive visualization systems do not for the same power of expression as a programming toolkit. Usually a system will do the basic things, such as bar charts, scatter plots and pie charts. More complex visualizations may be available in some systems, but usually they are rigid and the designer is left with less options than with a programming toolkit. One way to create a system with as much expression would be to implement every single layout by brute force. However, this is practically impossible. When designing visualizations it is not always obvious which variation is the most effective. With limited time, we tend to not test out all different options. Testing out more options increases the likelihood of finding a better design. This thesis will present a way to enable users to nest, layer and transform different charts. By using these operations, a great number of different visualizations can be expressed. However, these operations must be available in a simple and intuitive fashion. Such a simple and intuitive interface with these operations is currently unavailable. To address this, we propose a data structure that maps nicely to an intuitive interface, as well as the interface itself. It is easier to use, and easier to implement. The visual language (Visception) will enable users to intuitively build and manipulate one or more Visception Tree structures and see the corresponding visualizations.

The Visception Tree facilitates the construction of arbitrary nested and layered visualizations. Each node in the Visception Tree encapsulates one chart. For example, if there is a bubble chart with embedded bar charts, the corresponding Visception Tree has two nodes: The root node being the bubble chart, and the child node being the bar chart. To change it to a bar chart with embedded bubble charts, we would have to swap the node positions so that the bar chart is at the root node, and the child node is the bubble chart. In other words, to edit a nested visualization, the Visception Tree must be edited. To edit the tree, a set of operations is needed. This set of operations must enable the expression of any tree topology. The set of operations (*layer*, *nest*, *group*, and *delete*) allow us to express any tree topology - and likewise any nested visualization. These operations are made easily available in an intuitive user interface. This enables the user to intuitively edit and transform nested visualizations.

Each node in the Visception Tree holds a single chart. Editing a single chart is not always trivial, with nested visualizations it may be even harder. We have taken the approach of letting the user edit the charts one chart at a time. In practice, this means the user has to select one node in the Visception Tree to "link" the editor to that single chart. Charts are edited, by editing its *channels*. Building on Munzners [38] notion of a channel, each chart is edited through its set of channels. A channel controls one aspect of a chart - for example **Bar Height** or **Bar Width**. With this setup, each chart has a set of channels that are exposed to the user as a set of icons. Each icon can be clicked and a corresponding control will show. The control can be a slider, text input, color input or something more complex.

To facilitate the exploration of multiple designs, it is also possible to convert one chart into another one, while preserving the mappings. For example, if the user creates a bar chart, it can be converted into a scatter plot or polar area chart in one click.

To illustrate the power and flexibility of our approach we will reconstruct some non-trivial nested visualizations step-by-step in just a few minutes.

Chapter 2

Information Visualization and Visual Analytics

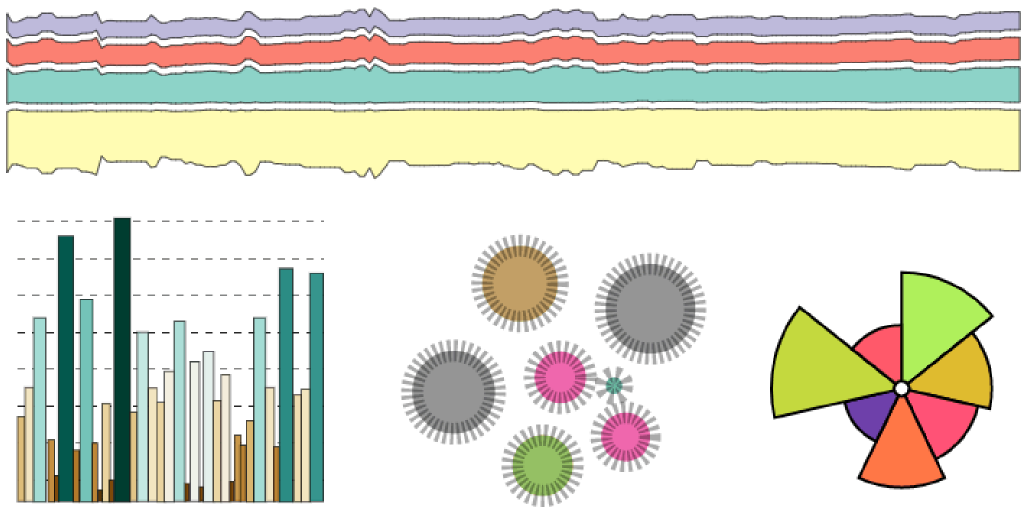


Figure 2.1: Here we see some examples of typical information visualization charts. Different charts tell a different story, picking and customizing the right chart is a crucial task for designers.

In this chapter we will give an overview of information visualization and visual analytics. We will demonstrate how some of the concepts can be improved upon and this will be used as basis for the next chapter. Information visualization is the study of ways to represent data in a way that is easy to understand and manipulate. We will limit ourselves to interaction techniques, visualization techniques, and the general direction of the field. Interpreting the information in data is not simple, and one way to better understand this information is to visualize it. By visualizing data we can help people make better informed decisions, and thus help people carry out their data tasks quickly and efficiently. In the field of information visualization we look for ways of visualizing datasets such as networks, hierarchies and multidimensional datasets, as well as ways of enabling users to conveniently create such visualizations. In visual analytics, we also want to enable the user to interactively explore the data. Creating a static visualization is not always enough. An interactive visualization enables us to gain insight into increasingly complex datasets. Good systems and techniques are crucial in fields where well informed decisions need to be made quickly, as discussed in [19].

2.1 Visualization Techniques

To create a visualization, a technique to turn the data into a chart is needed. Example charts are illustrated in Figure 2.1. Beyond the basics, many systems and techniques have been created or combined to address specific user needs. When the data is high-dimensional we need more sophisticated techniques. Parallel coordinates [27, 26] allows for the exploration of many dimensions, by having one axis for each dimension. If the data is categorical, parallel sets can be used [4]. Sometimes, the datasets are too big to visualize in one visualization. Researchers have proposed pixel-based and very compact layouts [48] and de-cluttering techniques [3] for uncovering clusters.

2.1.1 Basic Techniques

With simple datasets, simple techniques are often good enough. The most basic techniques and layouts include scatter plots, area charts, line charts and bar charts [13]. With these techniques we can visualize a few dimensions in a very effective manner. Other non-trivial techniques are usually less effective because they need to use increasingly unconventional visual channels to convey informations.

2.1.2 Radial Techniques

Radial techniques, while often argued to be ineffective and deceptive [35] are still widely used. Most known is the pie chart, but we also have other variations like elliptical pie charts, and spirals [14]. In some cases these techniques can expose cyclical patterns that bars and scatter plots can not.

2.1.3 Techniques for High Dimensional Data

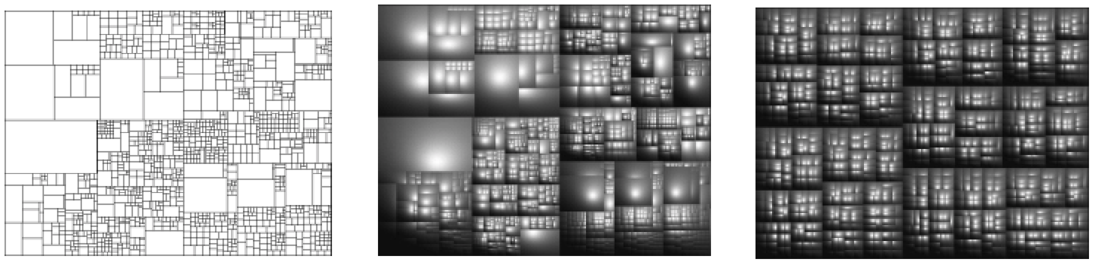
When looking for more complex relationships in data, or multiple dimensions, a simple chart is not enough. Using multiple basic charts, each depicting a small set of dimensions is a commonly used method. By making a selection on one chart, the selection is highlighted in the other charts. This is called linking and brushing [33]. While this is useful, more expression can be gained from using techniques that can express more data with less screen space.

Parallel Coordinates, Sets and Axes Based Techniques

Parallel coordinates [27, 26, 24] allows for visualizing many dimensions at the same time. In a parallel coordinate chart there is one axis per dimension, thus it can visualize a large number of dimensions at the same time. The links between each axis exposes the relationship between the two dimensions. Parallel sets [26] provide the same method for categorical variables. A drawback is that we only see the relationship between adjacent axes. Being able to rearrange axes quickly will solve this problem. Axes based techniques [51, 12] provide a more general form of parallel coordinates that allow us to layout axes radially, and arbitrarily decide which axes are to be linked together.

Hierarchical Techniques

In visualization, hierarchies create order and facilitate understanding. A hierarchy is either strictly defined by the data source, or can be arbitrarily defined by the user. Different visualizations correspond to different aggregation techniques [17].
TreeMaps [31] allows for compactly displaying deeply nested hierarchies (See Figure



(a) A basic TreeMap displaying a hierarchical dataset. TreeMaps give a good indication of what the data contains, but it is difficult to gauge the hierarchical structure of the data.

(b) A cushioned TreeMap. The shading makes it easier to determine the hierarchical structure of the dataset.

(c) A cushioned, squarified TreeMap. The squarification optimizes the layout to display less elongated rectangles, and display more even-sized rectangles.

Figure 2.2: Three kinds of TreeMaps depicting the same datasets. (Source: Figure 6 in Squarified Treemaps by Bruls et al. [8]).

2.2(a)). Cushion TreeMaps [55] do the same thing as regular TreeMaps - but uses shading to clarify the depth (See Figure 2.2(b)), while Squarified TreeMaps [8] optimize the layout to show less elongated thin rectangles appearing to be lines (See Figure 2.2(c)). TreeMaps can also be defined by an arbitrary, user specified hierarchy [11] . Matrices of charts such as the scatter plot matrix [16] reveals the connection between every permutation of a dimension. Dimensional Stacking [36] takes the same concept further by nesting dimensions even more deeply.

Handling Large Datasets

Suppose a dataset is extremely large, then, many techniques will result in a cluttered unreadable visualization. Parallel coordinates turn into "hairballs", and TreeMaps turn into black squares. However, there are techniques that can handle large amounts of data. Some techniques are pixel-based or extremely dense [48], and others augment existing techniques [3].

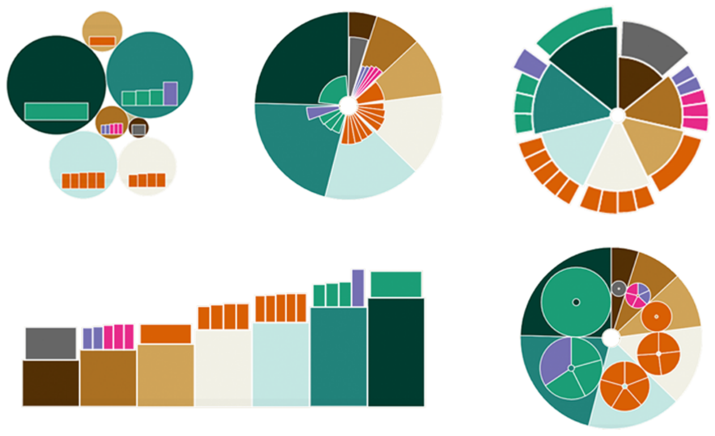


Figure 2.3: Five different visualizations created with nesting operations. By enabling nesting, it is possible to create such charts as a combination of simpler ones, rather than having to explicitly choose one hard coded nesting permutation.

2.1.4 Nesting and Combining

By nesting and combining the above techniques, a wider range of different visualizations can be expressed. Combining these techniques allows for picking and choosing the best techniques for the data. This is what the method in this paper aims to do. This approach turns simple charts into building blocks, that can be used to intuitively and incrementally build complex visualizations. Instead of providing a wide range of hard coded visualizations, like icicle charts and sunburst charts, we can instead provide a small set of building blocks. This will allow the user to nest bar charts within pie charts, and bar charts within bar charts, and much more. Figure 2.3 shows five different visualizations created with nesting, among them an icicle chart and a sunburst chart. With nesting and combining we can intuitively explore new visual representations.

2.2 Challenges and Directions

The direction of visual data analysis involves researchers discovering new layouts and interaction techniques. While discovering new techniques is important, all the methods that have been discovered are not easily available to those who might want to use them. Thus, it makes sense to bring together these already discovered concepts into one system. Designing interaction techniques [60], defining user tasks [47] as well as identifying challenges [34] are important aspects of keeping the research going in the right direction.

2.3 Marks and Channels

In order to reason about visualizations, a language for reasoning about visualizations is needed. A very important part of this language is these two terms: *marks* and *channels*. Munzner introduces the notions of marks and channels [38]. The basic idea is that each visualization consists of a set of marks, and a set of channels. Marks depict items or links. Intuitively, marks can be thought of as the "skeleton" for the visualization, while channels control the appearance of the marks.

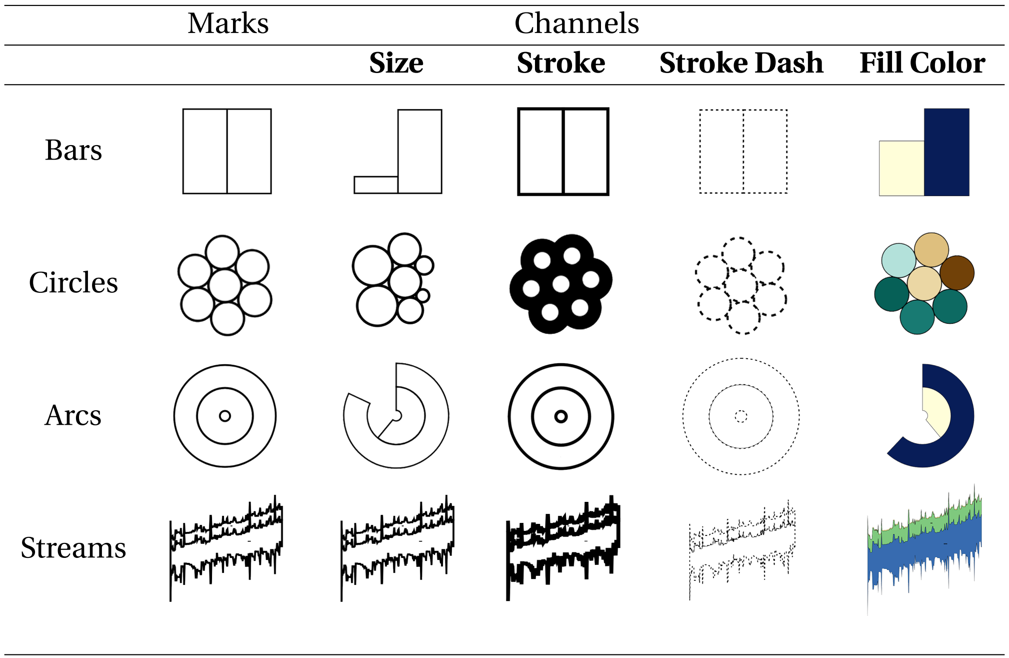
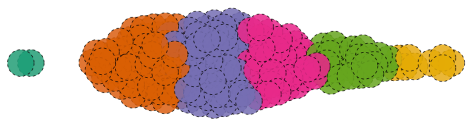


Figure 2.4: Some examples of a marks and corresponding channels. A channel controls the appearance of some marks. In this case, marks are defined as the basic shapes that depict each data item. For example, pie charts, as well as tubes as seen above consist of arc shapes. In some cases, the shape itself may be affected by a channel, then the mark is the deformable path.

With these two terms established, we can already reason about visualizations in a very straightforward manner. For example, a bar chart consists of marks (one rectangle per bar) and channels controlling the appearance of each bar (**Fill Color**, **Bar Width**, **Bar Height**, **Position**, and so on). What we can see after some closer investigation is that while charts may have different marks, they may some completely identical channels. These commonalities are useful when designing a larger space of visualizations. Figure 2.4 shows some examples of marks and channels, and how the different channels affect the marks.

2.4 Categorical and Continuous Attributes

In any dataset, there are different attributes. The attribute types determine how we can visualize it. For example, a numeric value such as height or length can be visualized differently than a set of hair colors. Since "height" is a numeric continuous value, we refer to such an attribute as *continuous*. "hair color", having only a small limited set of values will be referred to as a *categorical* attribute. While we simplify the discussion of attributes to *continuous* and *categorical*, Stevens [49] provides a more detailed classification of different attributes. Figure 2.5 illustrates the difference between a *continuous* and *categorical* attribute mapped to color ranges.


(a) A *categorical* attribute mapped to the **Fill Color** channel. It is clear that there is a smaller amount of distinct colors.

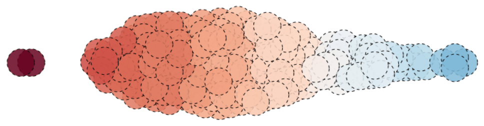
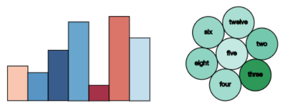
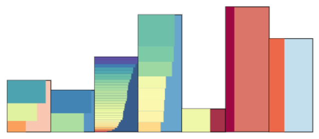

(b) A *continuous* attribute mapped to the **Fill Color** channel. Intuitively we can see the color changes gradually on a continuum, as opposed to the *categorical* mapping.

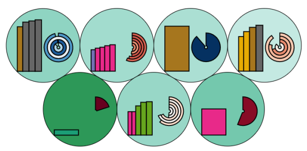
Figure 2.5: An illustration of the difference between a *continuous* and *categorical* attribute.



(a) Juxtaposition (side-by-side).



(b) Nesting (one chart within another).



(c) Two juxtaposed charts nested within a bubble chart.

Figure 2.6: Three different kinds of positioning settings.

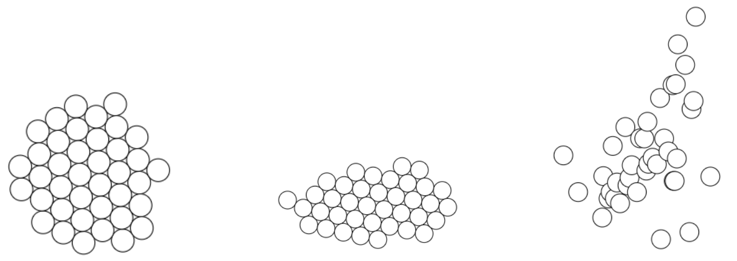
2.5 Nesting and Other View Operations

When creating a single visualization, positioning it is not something that requires a lot of thought and effort. If there are several visualizations, or combined ones, then the way in which they are positioned becomes more important. Visualizations can be combined in different ways. For example, we can put one visualization inside, beside, on top of, under, over, (or something in between) another visualization. This section will cover over a small set of common operations and ways to achieve this. The most common approach to display multiple charts is showing them *side-by-side* (juxtaposed, as seen in Figure 2.6(a)) [28]. By *nesting* charts, one chart is placed within another chart, for example a row chart inside a column chart as seen in Figure 2.6(b). In Visception these operations can be combined or used separately. In other words, it is possible to free-form juxtapose charts, even if they are nested within a visualization. An example of juxtaposition and nesting combined can be seen in Figure 2.6(c).

2.6 Table Arrangements

A table has both *categorical* and *continuous* (numeric) attributes. The most effective way to display an attribute is to map it to a spatial channel. By that, we mean **Position X**, **Size**, **Bar Width**, **Bar Height** and so on. For example, a scatter plot can display two *continuous* attributes - one for the **Position X** and another for the **Position Y**. Consequently, the traditional way of programming and creating visualizations requires these inputs as a bare minimum.

For example, to render a scatter plot a program may require two *continuous* inputs



(a) Starting with an empty viewport.

(b) Dropping one attribute creates one circle per distinct value of that attribute.

(c) Mapping a *continuous* attribute to the **Position X** channel first.

(d) Next, map a *continuous* attribute to the **Position Y** and set **Collision** to 0. This is now a traditional scatter plot.

Figure 2.7: Exposing the steps omitted by traditional visualization input requirements.

to be selected before anything is visible on the screen. This can complicate and slow down the process of exploring multiple designs.

When conceptualizing how the scatter plot is made, there is a mental jump from seeing nothing on the screen, to seeing a scatter plot. If we were to imagine which steps are omitted when going from a blank viewport to a scatter plot, we would (ideally) see three missing steps. To understand these steps, a generic plot chart as will be defined as a force layout. The force layout has a **Collision** channel, which if set to 1 it will ensure the circles do not overlap, if set to 0 it will be like a scatter plot. The first step is the creation of the circles, one circle for each data point. Usually this step is implicit, i.e each row in the dataset is one circle. If aggregating the data, there will be one circle for every row in the aggregated dataset. This first step is depicted in Figure 2.7(a) and 2.7(b). The second and third step is to position the circles on the X-axis, and on the Y-axis. The order of the second and third step does not matter. These final steps can be seen in Figure 2.7(c) and 2.7(d).

Another example is a bar chart. A bar chart traditionally requires one *categorical* and one *continuous* attribute. Instead, we could drop one *categorical* attribute to get a set of equally sized bars. Then, we could map any *continuous* attribute to the **Bar Height**, **Bar Width** or any other channel.

Our contribution provides a solution to these issues by not omitting the steps between and allowing the user to build visualizations more step-by-step.

2.7 Summary

While most people would know what a bar chart is, there is a wide variety of visualization different techniques. The most basic techniques include scatter plots, line charts, area charts, and bar charts. In some cases radial techniques are advantageous, especially if there is a cyclical relationship in the data. For higher dimensional data there are special purpose techniques, some of which are hierarchical like variations of the TreeMap. By nesting and combining basic charts, complex charts can be expressed by combining simpler ones. That is what Visception aims to do.

Marks and channels are two very useful terms when it comes to describing and reasoning about different charts. If a channel is mappable to a data attribute, it may in some cases accept only *categorical* attributes, in some cases only *continuous* ones, or in other cases, both *categorical* and *continuous* attributes. Thinking in terms of marks and channels allows for expressing charts in a more step-by-step manner. The mark takes in a bare minimum data input, and the channels accept varying inputs. This paradigm does not require the user to specify inputs in a certain order, or in clusters. Building on this, we propose a system that brings these concepts together, in order to make them easily available for non-expert users.

Chapter 3

Related Work

This chapter will cover previous work done on information visualization systems as well as underlying concepts these systems are built on. The central part of any system is a formal graphics specification, a field in which much work has already been done. By formally specifying ways of reasoning about graphics, such reasoning can be put into a system to specify different kinds of graphics. Formal specifications have given rise to programming toolkits that allow coders to express increasingly complex visualizations concisely and intuitively. We will go over the different paradigms of programming toolkits, leading to the train of thought that eventually led to D3 by Bostock et al. [7]. D3 was used as a foundation for higher level languages such as Vega and Vega-Lite [44, 22]. The programming toolkits have two main paradigms, the earlier ones were more focused on object orientation and providing interfaces for chart types, whereas Protovis [6] and D3 allowed for more flexibility and "visual thinking" by mapping data to graphical primitives, without having to get steeped in specific chart abstractions. Such "visual thinking" enables intuitive abstractions for charts as well as ways to explore and create new layouts with a minimal barrier of entry.

Early visual database exploration systems were spreadsheet systems, which over time evolved into using visualizations. Since then, visual database exploration systems are typically built on top of some kind of graphics programming toolkit. The graphics programming toolkit powers the rendering of visualizations. Older systems are built on systems more like Chi's Data State Model [10] and Prefuse [23] or some other custom implementation. Newer systems are more frequently indirectly or directly built on top of D3. The visual language presented in this thesis uses D3 for its rendering as well as data binding.

What separates Visception from other database exploration systems is that Visception enables nesting of visualizations. By nesting visualizations, simple charts are used as building blocks for more complex ones. One could argue that this train of thought started with the idea of using multiple views. From this, researchers looked for ways to make these multiple views "work together" – principles like Linking & Brushing [33] and positioning views side by side addressed these issues. Other techniques, like overloading [28] (putting one visualization on top of another) allow us to display more information using less screen space. Finally, the nesting of charts allow for combining simple visualizations into more complex ones intuitively. In some cases, nesting can with more clarity and less screen space, provide the same information as multiple linked views. Since then researchers have tried multiple ways of increasing the power of expression and exploring new ideas by enabling operations such as nesting, juxtaposing and overloading. However, most research results are either specialized interactive systems, or programming libraries exclusively for programmers or computer scientists. With Visception we express all these possible operations by enabling layering, nesting and adjustable bounds for each layer.

3.1 Formal Graphics Specifications

In order to specify multiple charts, an organized way of expressing them formally is needed. A formal specification allows for intuitively reasoning about charts, and more importantly it enables us to specify systems to express different visualizations. Interactive visualization editors are typically built on top of a visualization programming toolkit, and a visualization programming toolkit is usually based on a formal graphics specification.

Bertin [5] was the first to propose such a way of reasoning about graphics. He proposed six basic retinal variables (color, size, location, etc.), each assigned an expressive power and specific use cases. By using these variables we already have a way of talking about different kinds of charts. Notable is the spatial placement variable -

which is the most expressive and effective variable. For example, we can vary the **Fill Color**, or **Size** of a scatter plot. While Bertin exposes different variations of one chart, Wilkinson's Grammar Of Graphics [56] exposes both differences and similarities between different chart types. He breaks a visualization down into a set of components, and details how changing just one of these components can change the entire visualization. A very important observation he made is seeing the coordinate system as a property. This idea is crucial to nesting of visualizations. He pointed out that pie charts and a bar charts are essentially the same, but with different coordinate systems. Knowing these similarities and differences enables us to express a wider range of visualizations more concisely, both intuitively and programmatically.

Munzner [38] provide a framework and language for discussing visualizations. She presents two important terms: marks (geometric primitives) and channels. Every visualization can be expressed as marks (bars, circles, arcs) and channels (color, position, stroke width and so on).

While mathematical or topological differences between visualizations are useful for both reasoning and implementation, it is also crucial to consider how the visualization is perceived by the observer. Different kinds of shapes are perceived differently, by different people – some research has been done into this problem. Ziemkiewicz and Kosara [61] point out that even though there are "equivalent" visualizations (like the pie chart and the bar chart), the shape and overall impression changes the perception of the data, and call for a structural theory of visualization.

3.2 Visualization Recommendation

One crucial step towards exploring multiple visualizations quickly is to have the ability to quickly toggle between different structural representations of a visualizations. One step further would be for the system to simply recommend visualizations to the user. Such recommendation systems do exist and are likely to become more relevant in the future, just like recommendation systems have become popular in other sectors. These recommendation systems are built on a specification saying how different kinds of visualizations relate to one another. One such recommendation system is CompassQL [57]. CompassQL is a general query language for visualization recommendation, sorting, and grouping of different visualizations. It enables for to exploring, sorting, and filtering the space of available visualizations based on a declarative specification. Such programmatic reasoning about the effectiveness of visualizations enables the recommendation of visualizations to users. Vartak et. al [52] predict that visualization tools will have to become visualization recommendation tools. Current visualization tools lack the ability to navigate unexplored areas in the design space, fail to take the interest of the user into account when recommending, and lack an understanding of which kinds of insights the user is looking for. Roopana [32] is another system that attempts to semi-automate visualization recommendations. With Roopana we receive visualization recommendations based on predefined rules based on the best practices from visualization literature, the data types of the columns, and previous user actions. Voyager [58] provides automatic recommendations of visualizations according to statistical and perceptual measures as well as user preferences. In practice, data columns are dropped on shelves to specify which columns are mapped to which channels. Then, the system automatically displays recommended visualizations. Building further on Voyager, Voyager 2 [59] combines both recommendation and manual construction of visualizations. There is one *focus view* showing a selected visualization, and a set of *related views* suggesting related visualizations. Our contribution does not explicitly provide visualization recommendations, though toggling between an icicle chart, sunburst chart, and bar charts nested within circles can be done within seconds. Visception would be a good foundation for a visualization recommendation system.

Recommendation systems greatly simplify researching the problem of how different visualizations are perceived. It makes sense to have powerful visualization recommendation systems in place to make all options easily available.

3.3 Programming Toolkits for Visualization

Programming toolkits for visualization support the creation of visualizations by writing code. Each toolkit aims to be as simple as possible while at the same time being as expressive as possible. There are two main paradigms of such libraries: 1) Object oriented ones where there is a specific set of abstractions for each chart. Example thinking: "Draw a tree map for this data, here are the options." 2) A "visual thinking" approach where the level of abstraction is to what graphical shape we are drawing. Example thinking: "Draw one circle for each row of the dataset, and map data attribute A to the size, and data attribute B to the color".

In the early days of information visualization, visualizations were created by using low level libraries like OpenGL and other rendering libraries. The programmer would have to specify each vertex, transformation matrices, fragment colors and so on. Such low level APIs are not suitable for "visual thinking", thus multiple visualization toolkits that raise the level of abstraction have been developed. The Data State Model [10] is one of the pioneers in building a programming library based on a formal graphics specification. Chi describes the data state model, bridging the gap between formal graphics specifications and the implementation of such specifications as programming toolkits. He introduces the visualization data pipeline split into four stages (value, analytical abstraction, visualization abstraction, and view), as well as three transformation operators (data transformation, visualization transformation, and visual mapping transformation). These steps provide a clear streamlined way of thinking about the steps that turn a dataset into a visualization. Building further on the data state model, the Prefuse toolkit [23] was one of the first programming toolkits designed specifically for information visualization. It provides abstractions to create interactive visualizations, as well as multiple built-in layouts. However, Prefuse is still in the paradigm of providing a specific abstractions for specific charts, requiring the user to become familiar with the abstractions to create visualizations. Protovis [6] addressed this concern by providing a way of designing visualizations by combining graphical primitives. Protovis was one of the first toolkits enabling construction of visualizations by mapping data items to a set of graphical primitives. With this, we can program "visually" and only have to consider one data item at a time. Additionally, this approach is more flexible and allows for a more modular visualization design. Later, the same author published D3 [7] which consists of a small set of operators, allowing even more flexibility, extensibility and expressiveness than previous approaches. D3 enables direct manipulation of the Document Object Model (DOM) based on data items. D3 can concisely express both simple and complex visualizations without losing flexibility. Multiple libraries have been built on top of D3, providing an even higher level of abstraction and simplicity, but usually at the cost of flexibility.

When using D3, the interaction has to be specified manually, and in many cases the designer just wants a few types of interaction that might as well be available "out of the box". Vega [22] (built on top of D3) and projects built on Vega address this by enabling the programmer to connect actions to interactions as concisely as possible. In practice, the programmer can specify interactions in a declarative fashion. Vega provides a novel visualization grammar inspired by previous approaches. It enables us to specify visualizations even more concisely than with D3 – but at some cost of flexibility. Vega allows for specifying visualizations declaratively. Reactive Vega [45] introduced a declarative way of interaction design as well as visualization design. Events are seen as continuous streams, which the designer can specify a behavior for. Vega-Lite [44] provides an even higher level of abstraction, enabling the specification of interactive visualizations, as well as providing a view algebra to layout multiple visualizations on the same page. Currently, D3's full power of expression is limited to programmers only. Visception attempts to abstract over D3 and the Visception Tree data structure to enable users to visually combine and nest different visualizations while retaining as much of the expressiveness as possible. While Visception could be another programming library, we have decided to use drag & drop operations, sliders and other UI components to create a visual language.

3.3.1 Visual Data Exploration Systems

Having established both formal grammar specifications and visual programming toolkits, the next step is to enable users to create visualizations without having to write code. Such systems allow users to explore large datasets by creating their own visualizations. One limitation of the existing systems is the power of expression. Many existing systems still have an impressive range of features, but lack some of the flexibility offered by programming toolkits like D3.

One of the first, and still most widely accepted approaches for exploring large multi-dimensional databases were pivot tables. Pivot tables were first added as an explicit feature to Microsoft Excel, though Pito Salas and his team worked on the concept as early as 1986 [29]. As hardware improved, data visualization became more accessible, and information visualization became more commonly used. Multiple visualization systems have aimed to replace the pivot table, enabling users to visualize their data instead of looking at numbers.

One of the first systems enabling the creation of visualizations is IVEE [1]. IVEE can connect to a database and let the user choose from a rich variety of different visualizations. Among other charts, IVEE can create starfields, cone trees, while also filtering the query using sliders and other query handles. This wide range of features is very impressive given that it was published in 1995. Another database visualization system is Tioga2 [2], Tioga2 tries to build a visual language on top of a database query language, enabling non-coders to visually explore relational databases. While the phrasing is different and in accord with the time of its creation, it is an early visualization system aiming to make it possible to "wire up" visualizations using drag and drop operations. Visage [41] takes an information-centric approach, enabling the dragging & dropping of information between multiple windows/views. For example, it is possible to drag a set of rows to a separate plot view, and instantly see a scatter plot of the selected rows. Furthermore, Visage allows for selecting from a set of generated database queries, and generating visualizations from simple drag & drop operations.

The Polaris [50] interface by Stolte et al. (later commercialized as Tableau) enables rapid exploration of large multidimensional datasets. They introduce a table algebra for performing underlying data operations, a set of graphical operations to depict the query results, as well as a set of interactions to further explore the graphical depictions of the data. With Polaris we can simply drag and drop data columns onto a shelf, and see a corresponding visualization instantly.

Together, the presented visual tools extend the pivot table interface and allow for visually doing what can be done with a pivot table. Visception aims for the same kind of expressiveness as Polaris, but with a different set of operations. By exposing a greater set of channel mappings and combining visualizations, the same visualizations can be expressed with greater intuitiveness and flexibility.

While most editors are "hard-coded" in a sense, Lyra [43] allows for more flexibility. Lyra allows for interactively designing a large variety of visualizations using drag & drop operations. Notably, it has visual data pipelines, enabling advanced layouts and data transformations. The idea of a visual data pipeline enables much expression, but may also be too "programmatic" for non-coders to understand. While Lyra has powerful visual data pipelines, Visception focuses on nested visualizations and the combination of different coordinate systems in a generic manner.

3.3.2 Nesting and Related Techniques

By nesting visualizations we can express many visualizations using a few simple visualizations as building blocks. Being able to resize and move visualizations adds even more freedom to the designer. Ways of combining and editing existing visualizations allow us to express more with less. These operations are usually intuitive and allow for "visual thinking", both when programming or interactively designing visualizations.

If there is too much information to convey in one picture or one simple visualization, using multiple views is a feasible option. Norman et al.[39] were among the first to discuss the idea of using multiple views. They explored how humans interpret information displayed on multiple displays, as well as multiple windows. This idea was general and not specifically targeted towards information visualization, and novel at the time of its writing. More targeted towards information visualization, Schneiderman [47] proposed the visual information seeking mantra: Overview first, zoom and filter, then details on demand. This principle is helpful when designing advanced graphical user interfaces and is used (knowingly or unknowingly) by most systems today. Nested visualizations in particular, are manifestations of this mantra. Importantly, this principle provides a basic way of thinking about arranging visualizations in multiple views. For example, showing a visualization within another visualization can be seen as instantly giving details on demand. Having one visualization within another one already exposes a relationship within the data. Another way of exposing such relationships is juxtaposing (side-by-side) coordinated views. Juxtaposition is the most common way of coordinating multiple views, however it is not the only way. Javed and Elmqvist [28] detail four visual composition operators: juxtaposition, superimposition, overloading, and nesting. While Visception's main focus is on the nesting operation, we provide a flexible layering operation that, combined with movable and resizable bounds, achieves the same level of expression as using four operators. When juxtaposing views, it is natural to want to explore the relationships between the data of each view. A common technique to do this is linking and brushing [33]. When linking and brushing, a selection in one view will appear in multiple other (linked) views. LeBlanc et al. [36] describes the technique of dimensional stacking. Dimensional stacking is a way of embedding many dimensions into one visualization, this can be achieved through nesting or mapping data to a large set of channels. For example, five dimensions can be exposed by mapping each dimension to a separate channel. Such channels can be **Size**, **Fill Color**, **Stroke Color**, **Position X**, and so on. Dimensional stacking can be achieved by simply nesting visualizations by either aggregates or bins of a dataset. Since Visception exposes many channels and operations, the dimensional stacking technique is possible within our system.

ManyVis [42] operates at the program level, applying the principles of nesting, superimposition, and overloading to different application windows. In other words, ManyVis coordinates different windows into one window. For example, it allows for a video editor to be embedded into a PowerPoint presentation.

Nesting is not the only way to combine visualizations, and many different approaches for combining visualizations have been explored. Wickham and Hofmann [54] provide a way of transforming and combining area-based visualizations. They define three 1D primitives: bars, spines and tiles. These three primitives are used as building blocks to express a wide range of both simple and complex visual representations of data. Combining these primitives is intuitively similar to the nesting operation used in Visception. Blending existing visualizations is also a way of expressing new kinds of visualizations. Schulz and Hadlak [46] introduce a way of representing visualizations by blending together existing visualizations defined as presets. Their method allows for transitioning between different visualizations. For example, it enables smooth interpolation from a bar chart to a pie chart, or vice-versa. In the process of describing how to interpolate between different visualizations, they expose connections between different visualizations, such as the polar area chart and the bar chart. What is exposed on a more intuitive high level in formal graphics specifications is exposed in much more detail by Schulz and Hadlak.

Multiple views can also be juxtaposed and linked to display relations. Domino [21] uses overloading and juxtaposition to compare and manipulate subsets across multiple datasets. Domino can show relationships at multiple levels of detail, as well as expose relationships at multiple levels. Figure 3.1 shows a lot of information about artists from various countries using linked visualizations.

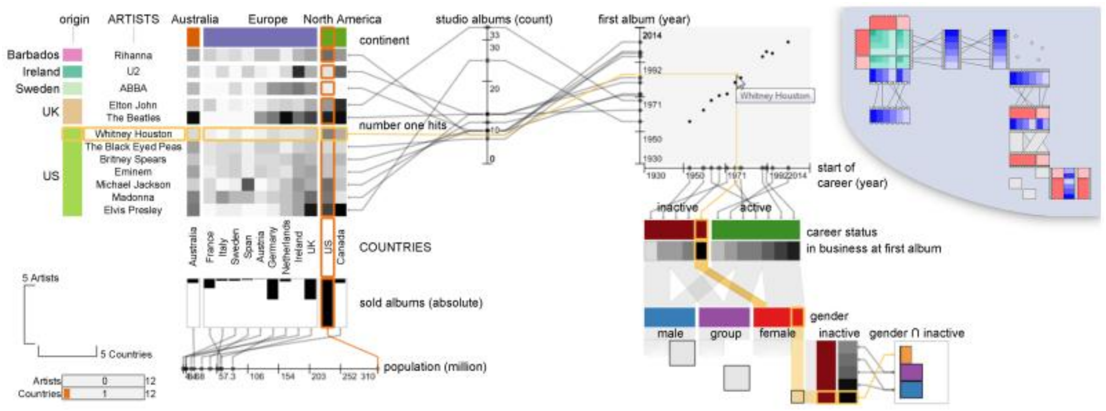


Figure 3.1: Figure 1 taken from [21]. It shows that that Whitney Houston is a female, inactive artist with many hits in English speaking countries, but less than 10 studio albums.

If the dataset represents a very large network, separate techniques may be required. NodeTrix [25] 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 as well as data relations using less visual space.

Nesting does not have to be limited to only 2D. Parker et al. [40], as early as 1998, designed NestedVision3D, allowing for the exploration of nested graphs to explore the structure of computer programs. From a codebase, NestedVision3D will give an interactive 3D graph, giving a very realistic depiction of all the relationships between the different software modules. Another approach to visualize the same kind of data involves showing more information inside cells of an adjacency matrix. ZAME [15] (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 huge datasets, as large as 500,000 nodes and 6,000,000 links.

Wang et al. [53] introduced the Circle Packing layout, nesting circles within circles at arbitrary levels. This layout may be expressed by nesting circles with a force layout [20] within one another. The force layout and the circle packing algorithm achieve very similar results. However, the force layout provides more flexibility, while the circle packing is less computing intensive.

Chapter 4

Creating Visualizations with Visception

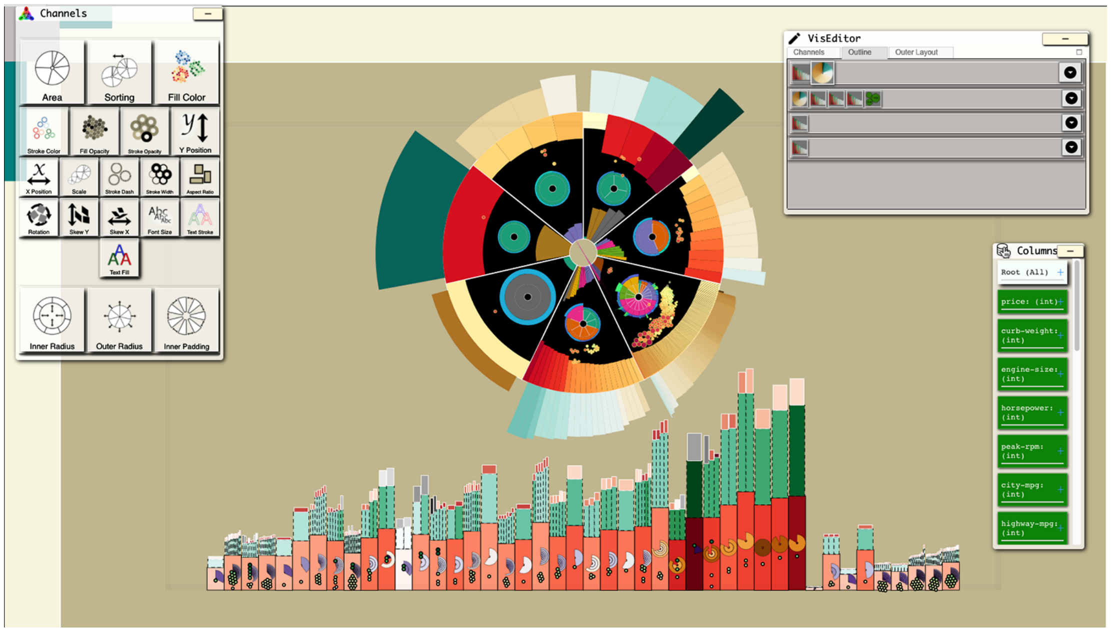


Figure 4.1: A complex visualization created with Visception in about 15 minutes.

Creating visualizations can be time consuming and not always intuitive. If the visualizations are complex (Figure 4.1 shows an example of a complex visualization), creating and customizing them becomes even more difficult. The goal of our approach is to enable users to visually create both complex and simple visualizations.

In order to enable this, an underlying data structure is needed. The constructs of this underlying structure must be flexible enough to enable the expression of arbitrary visualizations. Such flexibility is usually accompanied by complexity. Thus, it is a challenge to create a structure that is flexible and simple enough to be manipulated through a small set of visual actions. Our main contribution, the Visception Tree simplifies the design of visualizations greatly, without stunting expression and flexibility. The Visception Tree and its underlying structures is designed to be easily mappable to a simple user interface, enabling the rapid creation and exploration of nested visualizations.

A natural next step to exponentially increase the expressiveness is to enable the user to perform operations between different charts. A small set of operations can enable the creation of complex hierarchies by simply dragging and dropping. A nested hierarchy of visualizations is equivalent to a Visception Tree – a tree of charts. The child chart owns all the marks placed within each mark belonging to the parent chart. In a Visception Tree, each node can have multiple children, and each child node can have multiple children, and so on. Each visualization consists of one or more Visception Trees.

When interactively designing a visualization, the first step is to pick a chart. If the chart has *N* channels, the space to explore is N-dimensional. The fastest way to explore this space, is to explore one dimension at a time. In other words: One channel at a time. There are far more channels than charts, yet mathematically channels can be very similar. We will argue for a more precise definition of a channel, made up by several components.

4.1 Simplifying Table Arrangements

The insight the user is looking for in a particular dataset is not constant. Furthermore, with high-dimensional datasets, it is even less clear exactly what to look for. Even if knowing what to look for, expressing it in terms of a visualization may be even harder. Typically, visualizations are defined as requiring a set of inputs. For example, while a scatter plot can take in only one key, a matrix requires two (one for each direction). It gets more complicated when a visualization can only take in certain kinds of data columns. This section will go over some different visualizations and their required table arrangements, before showing how these requirements are greatly simplified within Visception.

4.1.1 The Scatter Plot

The scatter plot represents a set of points on a Cartesian grid. Each point is positioned by a given position on the X and Y axis. Thus, the X and Y positioning represents two dimensions. Other adjustable inputs include the size of each point, as well as the color of each point. Data items could be mapped to even more channels, like the size, the stroke width, stroke color and more. Initially it may seem that a set of inputs is needed in order to get something showing on the screen. However, this process can be simplified so it requires only one data column to get started.

Visception can get a bubble chart up on the screen without requiring any initial inputs other than the data column. Initially, if a data column D is dropped on the screen, the data is aggregated by that column. Then, one circle will appear for each distinct value of D. After getting the bubble chart up, attributes can be mapped to the X and Y-position, but it is not a prerequisite. This enables more expression, while requiring less inputs to get started. Traditionally, the scatter plot is not aggregated – this means there will be one circle for every row in the dataset. By taking in the level of aggregation as the first input, both aggregated and unaggregated scatter plots and force layouts can be intuitively expressed as variations of one chart type. Figure 4.3

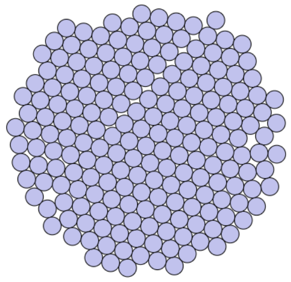
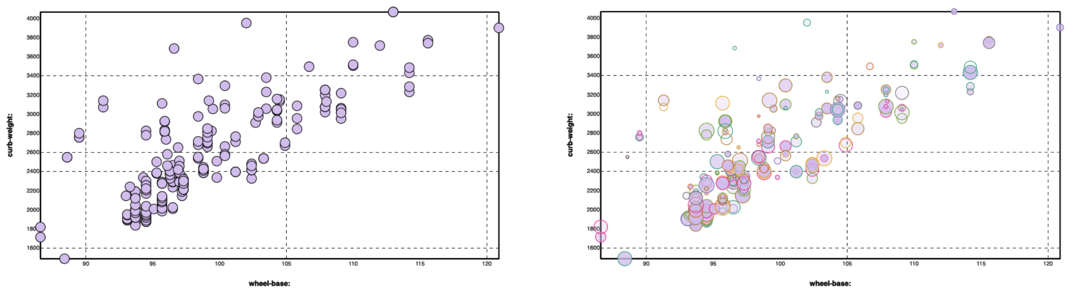


Figure 4.2: A bubble chart created in Visception. When dropping one data column, one bubble is rendered for each distinct value of that column. Intuitively we can see that this is similar to a scatter plot without X or Y-mappings.



(a) The same chart as in Figure 4.2, but with a mapping for the X and Y column and **Collision** set to 0. This makes it equivalent to a regular Cartesian scatter plot.

(b) Here we expose more dimensions by mapping data attributes to the **Stroke Width** and **Size** channels.

Figure 4.3: Incrementally adding more dimensions to the visualization.

illustrates these steps.

4.1.2 The Bar Chart

A bar chart represents a set of values, and traditionally requires one *categorical* and one *continuous* input. One bar is created for each distinct value of the categorical attribute, and the bars are assigned heights according to the continuous attribute. This requires the user to point out one categorical, as well as one continuous column before seeing anything on the screen. This can be simplified by requiring only one data column to render some bars. In other words, simply dropping any data column on the screen will create a corresponding set of bars - all with the same height. Figure 4.4 and 4.5 show how a bar chart changes incrementally as data attributes are mapped to different channels.

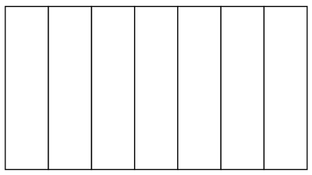
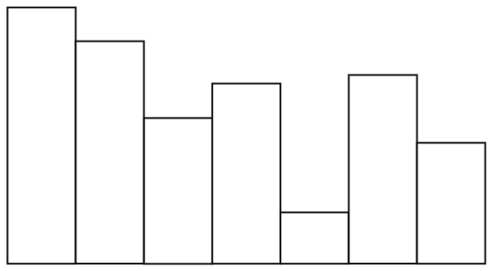


Figure 4.4: A bare minimum bar chart created in Visception. Dropping one data column implicitly renders one bar per distinct value of that column.


(a) The same bar chart as in Figure 4.4 with a *continuous* data column mapped to the **Bar Height** channel.

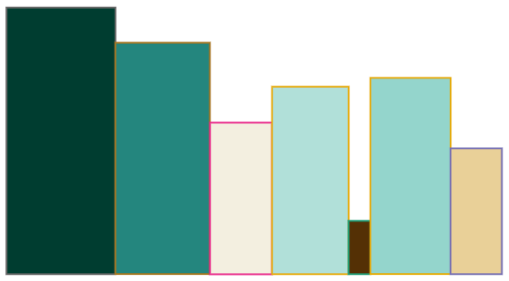

(b) Same bar chart as in *a)*, but with data mapped to the **Bar Width** and **Stroke Color** channels.

Figure 4.5: Incrementally displaying more dimensions with a bar chart.

When the basic bars are rendered, the user is free to keep adding dimensions to the bars one step at a time. For example, dragging a column on the **Fill Color** will expose the dimension of that column through the **Fill Color** channel, and so on.

4.2 Charts

Representing a visualization as one object requires a higher level of abstraction than simply mapping one data item to one mark. A clean encapsulation of an entire visualization is needed. To address this, we will refer to such an object as a chart. A chart represents the mapping of a set of data items to a set of marks. In our approach, a chart transforms a selection of data into a set of marks or one mark.

Charts represent different ways of displaying a selection of data. For example, data can be displayed as a bar chart, a pie chart or a scatter plot. This concept is also in effect when it comes to nested visualizations (See Figure 4.6).

The visual appearance of the marks is a direct result of the input channels of the chart. Each chart has a distinct set of input channels, and some channels may be mapped to data columns. A few examples of channels are **Fill Color**, **Stroke Opacity**, **Stroke Dash** and **Area**. Each channel can be seen as one dimension of a chart. All *N* dimensions of the chart make up the N-dimensional design space. For nested visualizations, the design space is the permutation of the design spaces of every single chart. Visception allows for exploring this space by providing simple controls for

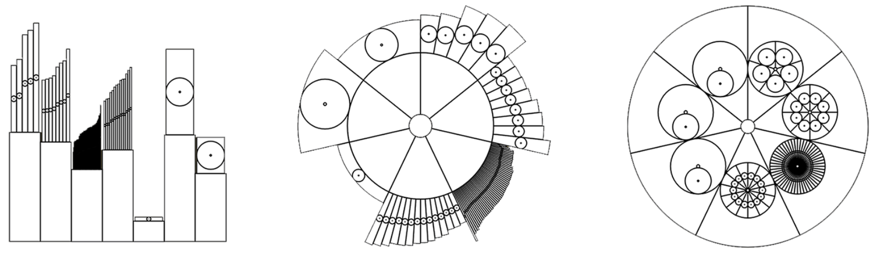


Figure 4.6: Three visualizations depicting the same data. These visualizations have the same underlying tree structure and data mappings, but different charts.

each channel.

Each chart has a distinct set of spaces, a layout space, child spaces, a parent space, and a display space. The nesting behavior of a chart is determined by its own spaces and the spaces of its parent. Every chart "begins" in its layout space – where the normalized coordinates of the marks are calculated. Then, the marks are transformed to fit within the parent space. Finally, the marks are transformed to fit within the display space (the coordinate system used when rendering the shapes). The parent space, is the child space of the parent chart. Figure 4.7 gives a high level overview of the spaces. These spaces are discussed more closely in section 4.3.2. Each of the spaces have a type, which refers to the internal coordinate system of that space. For example, the child space type of a pie chart is an *arc*, while the child space type of a bar chart is *Cartesian*. All marks of a chart share the same child space *type*, yet mark has a unique child space *instance*. For example, a bar chart with 3 bars has 3 child spaces, one for each bar.

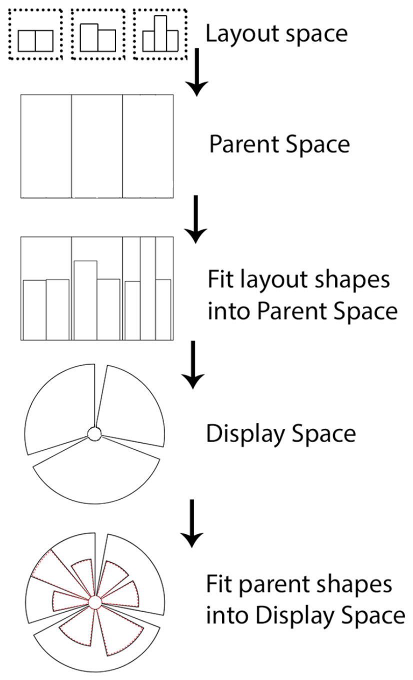

Layout space
Parent Space
Fit layout shapes into Parent Space
Display Space
Fit parent shapes into Display Space

Figure 4.7: A high level overview of the spaces of a chart. Initially, the charts shapes are calculated in layout space. Then, the marks are fit into the parent space. If the display space is not equal to the parent space, the marks are again transformed to fit into the display space. This figure is only meant to give a quick intuition of what these spaces do. They are discussed in more detail in section 4.3.2.

Each chart is a part of a chart class and has a chart cardinality. Charts grouped together are intuitively related and have distinct common channels that are not present in other chart classes.

4.2.1 Chart Cardinalities

In order for nesting to make sense within a mark, the mark must be nestable. To determine whether a mark is nestable or not, one deciding factor is the cardinality of the chart. Considering the dataset, we can not nest a visualization within an area chart or streamgraph. In order to have something to nest within a chart, there must be aggregated data that can correspond to the parent mark. For area charts, streamgraphs and lines, there are no aggregations corresponding to a deeper level of nesting. Intuitively, an area chart, line chart or streamgraph "uses" up all the rows in the dataset. Visually, it is possible to imagine nesting a chart within an area or a stream, but the data depicted in the nested chart would not be nested. Thus, this nesting would rather be a layer fit within the area mark.

Cardinality	Description	Examples
Many-to-one	Maps multiple data points to one path	Lines and areas.
One-to-one	Maps N data points to N paths	Plots, bar charts, pie charts.
Many-to-many	Maps N data points to M paths	Series.

Table 4.1: Chart cardinalities

A chart maps a set of data items $D = \{d_0, d_1, \ldots, d_{N-1}\}$ to a set of marks M. $|M|$ does not necessarily equal $|D|$. For example, if rendering a line, multiple data points are mapped to one path. If rendering a bar chart, or a set of circles, the mapping is one-to-one. We refer to this as the chart cardinality, which is an inherent property of each type of chart. Table 4.2 depicts common chart types and their cardinality.

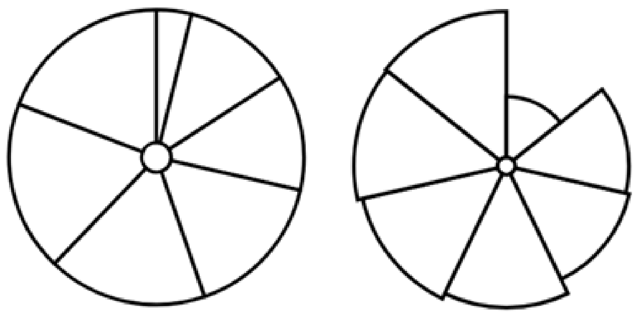


Figure 4.8: An area-based pie chart and a polar area chart. These two charts share the same set of input channels, yet they are different. The only difference between them, is how they depict their areas. The pie chart area is modulated by changing the arc angle, while the polar area chart area is modulated by adjusting the outer radius of the arcs, while the radius is constant.

4.2.2 Chart Classes

There are both commonalities and differences between different charts. For example, there is an intuitive similarity between an icicle chart and a nested bar chart. The purpose of defining and classifying charts is to put the basic building blocks of visualization into a system that is as general, flexible and simple as possible.

A chart class is a grouping of charts based on common properties that distinguish them from others. The most important properties that distinguish one chart from another are its spaces, its possible layouts, and its channels. For two charts to be classified the same, they must have the same child space type. For example, a bar chart and a pie chart have different internal child space types (internal coordinate systems). Finer details also play into the classification of charts. If some charts share a distinct set of channels they can usually be classified similarly. Such charts are usually intuitively related (for example, a pie chart and a polar area chart as shown in figure 4.8). In some cases a new type of visualization can be made available by simply adding one or several channels to an existing chart.

Adding channels to existing charts is preferable, but only if adding the channels does not break the already existing semantics of the chart or introduce too much complexity. For example, an area-based pie chart and a polar area chart are defined as two distinct charts. Yet, they could be defined as one chart, with a channel to toggle between pie and area. There could even be one chart trying to express every sin-

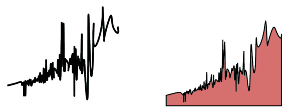
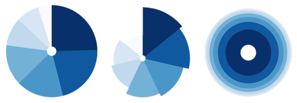
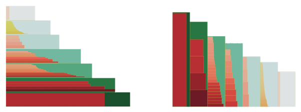
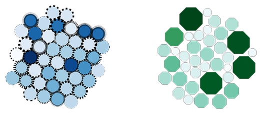
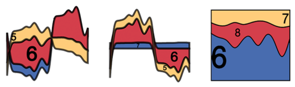
Class	Examples	Chart Cardinality
Line Based		Many-to-one
Circular		One-to-one
Bars		One-to-one
Plots		One-to-one
Series		Many-to-many

Table 4.2: Five distinct chart classes, all nestable except Line Based and Series. Note how bars and plots have a *Cartesian* child space, while the Circular child space is an *arc*.

gle possible visualization. The goal with charts is to classify and simplify, too much generalization may create more complexity than simplicity. Charts are loosely connected by specifying mapping equivalences between channels belonging to different charts. For example, the **Bar Height** channel B of a bar chart is equivalent to the **Area** channel A of a polar area chart. Thus, when toggling between those two charts, the channels A and B would have the same data mappings. Using these concepts, it is possible to specify simple visualizations such as bar charts, pie charts, scatter plots and areas and fully customize them by mapping data columns to channels.

4.2.3 Channel Mappings

When creating a visualization, choosing the chart type is only a part of the equation. It is also needed to control the appearance of the chart, and determine what data the chart shall represent, and exactly how it should be represented. Every chart has a set of channel mappings. The channel mappings control the final appearance of the chart. The task is then to enable the user to edit these channel mappings conveniently and easily. Such a simple paradigm can expose a large range of channels and give the user more power of expression.

Typically, the layout and styling channels are separated, while in Visception they are accessed and edited through the same interface and have meaningful defaults. This allows us to expose a wide range of channels for each chart, increasing the flexibility and range of expression of each chart. In other words, a channel can receive any input, and produce any output. The transformation of input to output may also be specified, or have a different behavior for each channel. For example, a **Size** channel may take in a numeric value, or a data column and a numeric range. The appearance of a chart is a function of all its channels. One channel controls one aspect of the appearance of a chart. When mapping a data column to a channel, the goal is to find the best way to represent the data through that channel. For example, if mapping a data column D to the **Fill Color** channel, we must pick a color scheme, whether the color scheme should be discretized or continuous, how the domain of D should be mapped to the color range, and so on.

A more concise mathematical definition is that every single channel has an input I and an output O. Let $dom(I)$ be the input domain, and $dom(O)$ be the output domain. $dom(I)$ and $dom(O)$ can be sets of *categorical* values (for example, colors and strings), numeric ranges, or custom objects (for example, sorting orders). The transformation function T maps a value of $dom(I)$ to a value of $dom(O)$. While this may seem a very general definition, some channels have distinct common traits. For example, when mapping a *continuous* data attribute to the **Size** channel, T is simply a function mapping one numeric range to another.

Intuitively, a channel mapping can be thought of as an input to a visual channel. The input can be as simple as a single value, or a data attribute. The output can be a set, a *continuous* range, an ordering or a custom object. What is common for all channel mappings is that they have an input and a resulting output, and an arbitrary number of transformation steps in between. Table 4.3 illustrates some different channel mappings, their inputs, outputs and the visual effects on a deformed bar chart. Tables 4.4, 4.5, 4.6 and 4.7 show some (but not all) channel types available in our system.

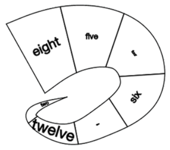
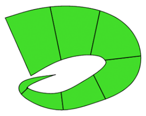
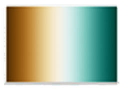
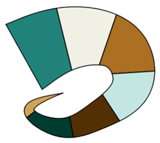
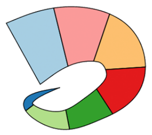
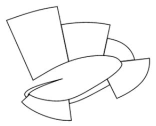
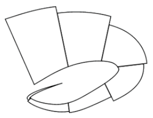
Channel	Input	Output	Illustration
Font Size	*Continuous* Attribute	Numeric Range	
Fill Color	Color	Color	
Fill Color	*Continuous* Attribute		
Fill Color	*Categorical* Attribute*		
Bar Height	*Continuous* Attribute	Numeric Range	
Sorting	*Continuous* Attribute	Order	

Table 4.3: Some example channel mappings on a deformed bar chart. These examples illustrate the wide range of channel inputs, outputs and their results on their respective charts.
*The *categorical* attribute is in this case the same attribute as the aggregation. I.e there is a one-to-one mapping between each bar, and each distinct value in the domain of the attribute.

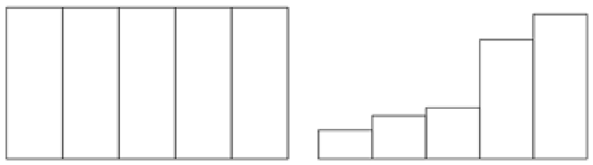
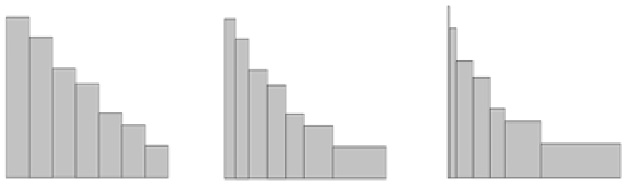
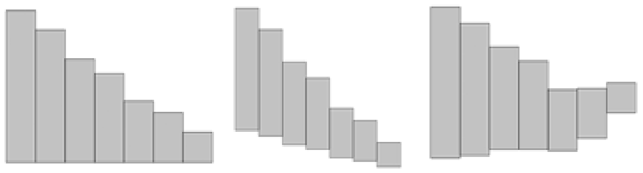
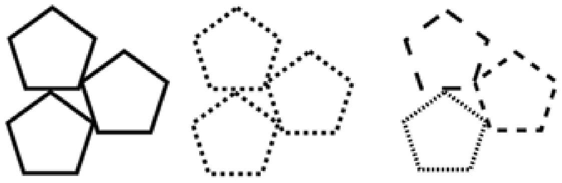
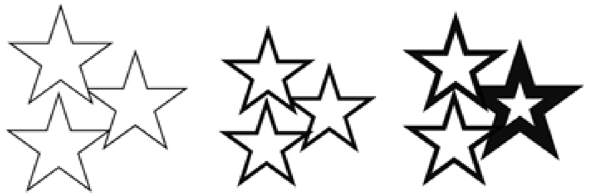
Channel	Illustration
Bar Height	
Bar Width	
Bar Baseline Offset*	
Stroke Dash	
Stroke Width	

Table 4.4: Five different channels. **The **Bar Baseline Offset** channel displaces the bars upwards. For example, it can allow us to place a bar chart on top of a line chart (assuming the **Position X** channels match). This channel is more useful for **Positioned Bars**, where **Position X** is directly mappable to a column.*

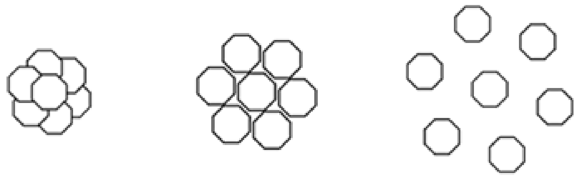
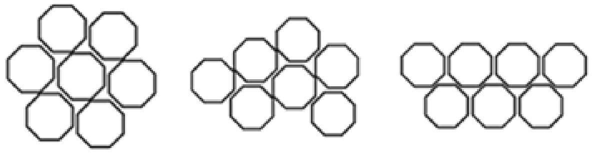
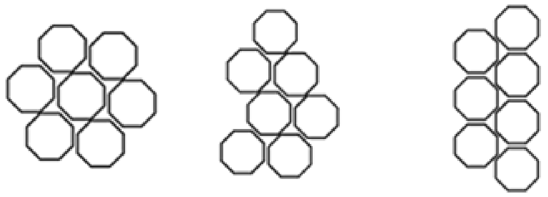
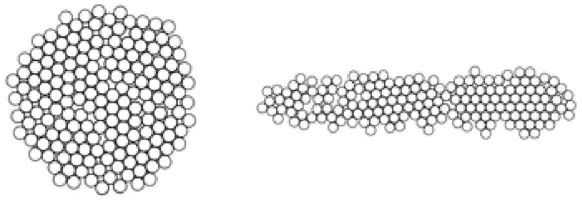
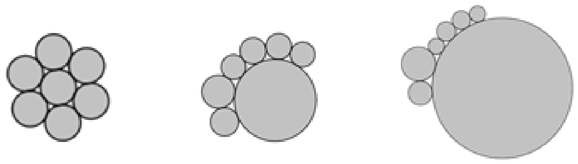
Channel	Illustration
Collision	
Force X	
Force Y	
Position X	
Size (plot)	

Table 4.5: Five different channels. All of these are exclusive to plots. By modulating these channels we can express a wide range of different plots including force layout variations. Note how we can also map data to all of the channels in this table.

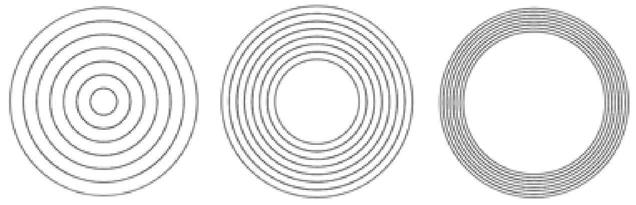
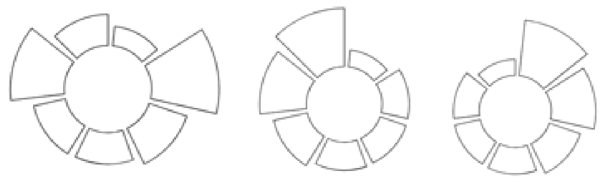
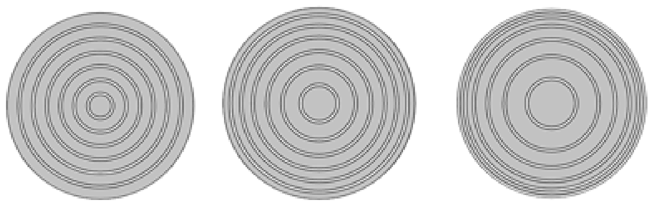
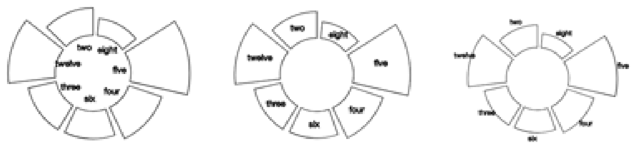
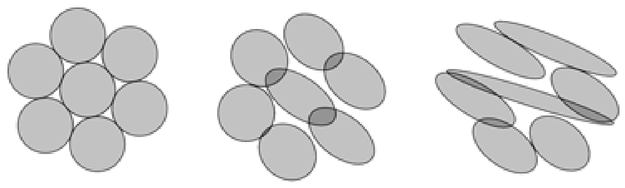
Channel	Illustration
Inner Radius	
Sorting	
Tube Height	
Text Transform*	
Skew X	

Table 4.6: Five more different channels. **The **Text Transform** channel allows us to rotate and translate the text within a Cartesian space, and have Visception automatically apply a transformation corresponding to the parent space type.*

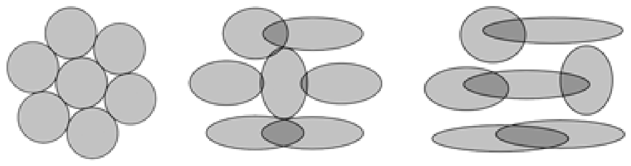
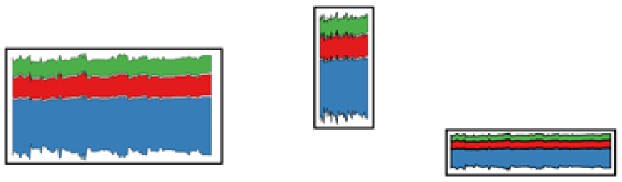
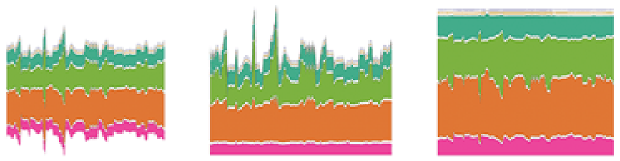
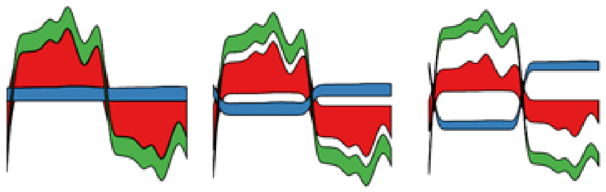
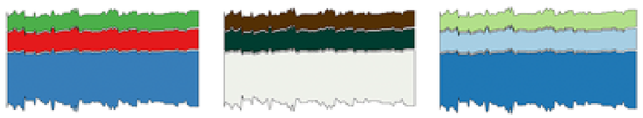
Channel	Illustration
Aspect Ratio	
Bounds*	
Stream Shape	
Stack Inner Padding	
Fill Color	

Table 4.7: A set of five channels. **The* ***Bounds*** *channel enables us to move and resize each chart within any space. The* ***Bounds*** *channel allows us to achieve any size and position, also when a chart is nested within another or has a* non-Cartesian *parent space.*

4.3 The Visception Tree

Every nested visualization corresponds to a tree. For example, a bar chart with two polar area charts inside each bar corresponds to a tree with a root node (corresponding to the bars) and two children (corresponding to the pie charts). We can express arbitrary nested visualizations by building a tree corresponding to the visualization. The first requirement to build such a tree is to be able to encapsulate one visualization into one tree node. Since we established that a visualization can be represented as a chart, creating nested visualizations is as simple as creating a tree of charts. Such a tree will be referred to as a VC-tree (short for Visception Tree). Likewise, a node in a VC-tree will be called a VC-node. Each visualization can contain multiple VC-trees.

4.3.1 Operations

Each node is directly affected by the chart it represents. For example, if the chart encapsulated by the node is a line chart, the node can not have any children, because the line has no child space. A VC-node has no attributes in itself other than a depth and an index. In other words, most of its properties are determined by the chart it encapsulates. Thus, the properties of the chart determine which operations can be performed on the tree, as well as the outcome of these operations. Before we can discuss how chart properties affect operations, we will cover the basic operations of a VC-tree. Other operations are possible, but we will limit the discussion to the operations *Nest* (N), *Group* (G), *Layer* (L), *Move* (M), *Delete*, (D) and the special operation *Push* (P). With these operations any VC-tree can be expressed. Other operations are simply shortcuts that can be expressed as a combination of these operations. Let C be an arbitrary chart with input data D_C, and M_C be the set of marks corresponding to D_C. Furthermore, let D_i be an arbitrary data column, and C_i, M_i be an arbitrary chart, and a set of marks corresponding to D_i. Our VC-tree operations are defined as follows (see also Table 4.8)

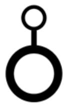
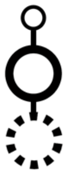
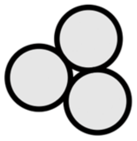
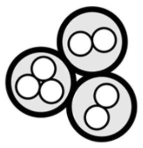
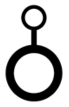
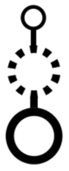
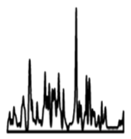
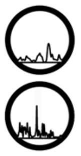
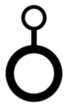
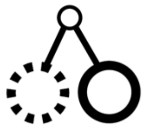
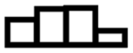
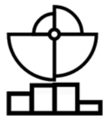
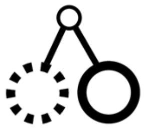
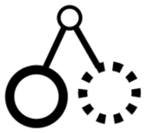
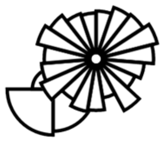
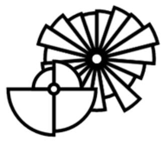
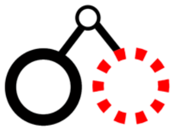
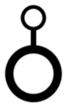
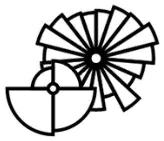
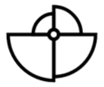
Operations	Visception Tree		Visualization	
	Before	After	Before	After
N (Nest)				
G (Group)				
L (Layer)				
M (Move)				
D (Delete)				

Table 4.8: Visception Tree operations and their corresponding visualizations and tree topologies before and after.

Nest $N(C, D_i)$ is equivalent to $N(C, C_i)$. For each datum d_C, get the aggregated set D_{Ci} – each d_{Ci} will correspond to a mark m_{Ci} to be nested inside m_i.

Group $G(C, D_i)$ is equivalent to $N(D_i, C)$, i.e., it "places" C inside D_i. $N(A, B) \cong G(B, A)$

Layer $L(C, D_i)$ adds D_i as a sibling to C.

Move Changes the tree position of a node, i.e., it moves the node and its children to the new position. Let d be the depth of the node, i be the index of the node, N_e be the node existing at (d, i). Then $M(C, d, i)$ inserts the node at the given depth and index of the VC-tree and shifts any existing node N_e (and its neighbors to the right) at (d, i) to the right. In other words, it layers C on top of N_e.

Delete Removes a node including all its children from the tree.

Push Adds a data column to the node, but does actually not alter the VC-tree. This operation is only useful and available for series charts (e.g., streamgraphs).

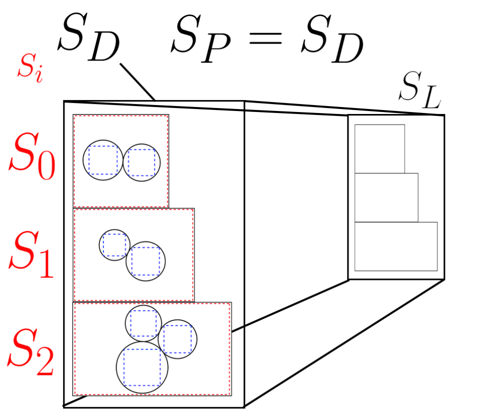
For the *Nest* and *Group* operations, the following conditions must be met:

Nest $N(P, D_i)$. D_P must be an aggregate because it is not meaningful to nest a visualization into a node that only represents a single datum.

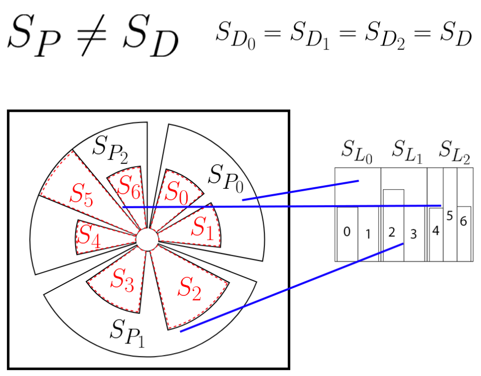
Group $G(P, D_i)$ D_i must be an aggregate (for the same reason stated above).

4.3.2 Nesting and Spaces

Having established charts, the VC-tree and its operations, we now propose a generic way of sizing and positioning child marks within parent marks. A chart has several spaces to consider (see Figure 4.9): Its parent space S_P, layout space S_L, and its display space S_D. S_P denotes S_L of the parent chart, or a default space if the node is a root. A single nestable chart must consider the space of its parent S_P, which also affects its display space S_D. The layout space S_L is the space that is used when calculating the layout of the charts marks. Finally, if we are to nest something inside a chart, each mark on the chart has its own space S_i, which will be the parent space for



(a) A depiction of four spaces types. In this case S_P is the same as S_D, this is not necessarily always the case. For example, if S_P was not Cartesian, then S_D would be the inherited from its parent. In other words, it would take the first-hit *Cartesian* space as its display space, recursively up towards the root of the Visception Tree.



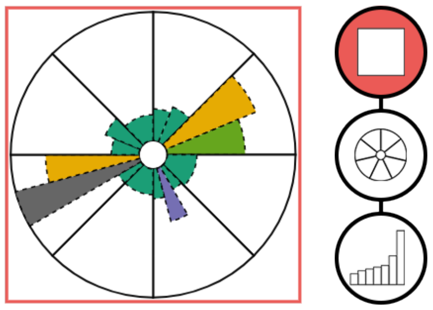
(b) A depiction of four spaces types where $S_P \neq S_D$. It is crucial to note we are looking at it from the perspective of the innermost bars, and more crucial – one bar at a time. Each bar maps its layout space to the corresponding parent space, however since the parent space is deformed, it will instead be applied as a transform before being rendered in the display space of the parent's parent.

Figure 4.9: Two examples of the four spaces of a chart.

child charts. Together these spaces determine how the layout of the chart is done, and how the layout of the child chart is done.

All spaces $\{S_P, S, S_C\}$ share the same domain. The domain of these spaces include different grid types. Common examples are: The regular *Cartesian* grid and the *polar* grid. The full domain includes all possible grids that are not imaginary and that can have their coordinates converted to a regular *Cartesian* grid.

A vital step to enable the nesting of charts is to be able to convert a coordinate of a grid G_1 into the coordinates of a grid G_2, where $G_1 \neq G_2$. Each root node has a layout space S_L. The layout space defines the coordinate system in which the marks are placed. For example, a scatter plot has a *Cartesian* layout space. In order to fit a scatter plot inside an *arc*, we would have to transform the coordinates from its layout space S_L to its parent space S_{P_j}, where j is the index of parent datum. Furthermore, if we were to fit a column chart with a *Cartesian* layout space S_L into an *arc* we would have to transform the columns from *Cartesian* to *polar* as shown in figure 4.9(b). Such transformations enable nesting of arbitrary charts.


(a) Since the immediate parent space of the bar chart is an *arc*, the *arc* deforms the space before it is transformed to the *Cartesian* coordinates corresponding to the inner space of the square.

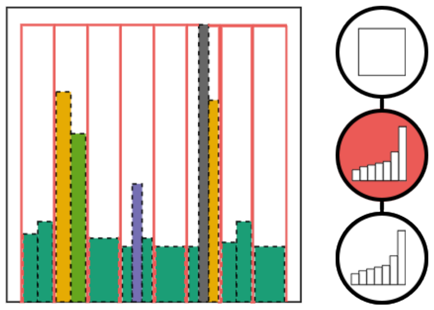

(b) The immediate parent's child space is *Cartesian*, hence there is no intermediate deformations. The bars are fit into the **Cartesian** coordinate system of the immediate parent.

Figure 4.10: Deformation of layout coordinates according to parent spaces. The red VC-node is the node that contains the display space of the leaf. Note how the node in-between can still deform the coordinates before they are transformed to fit the display space of the marked node.

For the sake of brevity, we will limit ourselves to two grids. The general principle is valid for any other arbitrary 2D space S_a, as long as its grid coordinates can be transformed to 2D *Cartesian* coordinates and vice versa. A *Cartesian* coordinate is defined as (x, y), denoting longitude and latitude on a 2D rectangular grid. A *polar* coordinate p is defined as (r, ϕ) where r is the distance from the origin and ϕ is the angle. When nesting charts, we need to determine which space to calculate the layout in, and which space the calculated layout will be positioned within. These two spaces will be denoted as S_L and S_D. When a chart C's layout space S_L has a different coordinate system from its immediate parent space S_P, the chart's layout in S must be transformed to fit in its display space S_D. S_D is the space S will be fitted into. In other words, P will calculate its layout in S_L-coordinates. S is then transformed to fit into S_D. The nature of this transform varies, and depends on the nature of the chart C, and its parent chart C_P. For example, to fit a scatter plot into inside an arc, only the coordinates are transformed. Another example is when fitting a column chart inside an *arc* (a circular chart), the entire shape is deformed, and not just the coordinates. Each VC-node has one such transformation, and each child will handle the transformation from the parent in a unique way.

Let VT be an unbranched VC-Tree with d nodes. Let T_0 be the transformation applied to the root node, and T_{d-1} be the transformation applied to the node at depth $d-1$. Here T denotes a very general transformation, a function that transforms a set of coordinates. The transformation applied to a VC-node at depth $i, 0 < i < d$ can be expressed as $T_0 \circ T_1 \circ \cdots \circ T_i$. The way these transformations are received is specified by the chart and its parent charts. Some charts will deform their center coordinates and scale to fit within the parent, some will deform their entire shape, some will deform their children. It is up to the designer to specify exactly how the charts should be transformed by their parents, as illustrated in Figure 4.10.

4.4 Summary

This chapter has covered the main underlying principles of Visception. It began by showing ways to simplify table arrangements by requiring less initial inputs, and showed examples of how this is advantageous when used visually. Requiring less initial inputs enables the user to do things more step-by-step.

Visception is built on a framework for classifying charts. This classification system is based on the channels and properties of each individual charts. In particular, the internal coordinate system (synonym to child space) of the chart is crucial to classifying charts. For example, a pie charts and a bar charts are intuitively different.

To build these charts, operations must be made available. The operations within Visception, *Move* (M), *Nest* (N), *Layer* (L), *Delete* (D) and *Push* (P) enable the user to express a wide range of visualizations, and modify the Visception Tree topology freely. Some operations, specifically the *Nest* operation can only be done on charts with a chart one-to-one chart cardinality. With the ability to build a Visception Tree and create an arbitrary hierarchy, the final step is to modify each chart one by one. This is done by editing the channel mappings. Channel mappings encapsulate most ways of editing a chart.

It has also been presented how the nesting of spaces work. Four different spaces have been suggested: The layout space, the parent space, the display space, and the child space. These four spaces provide a framework for creating nested visualizations, even with varying internal coordinate spaces.

Chapter 5

Interaction

The Visception Tree data structures enables for programmatically editing nested and layered visualizations. However, it should also be possible to edit the Visception Tree, without having to write any code. In this chapter we will demonstrate how the Visception Tree is mappable to simple user actions, enabling the highly flexible interactive design of visualizations

A Visception Tree provides a discrete, flexible representation of arbitrary visualizations. Most visualization programming libraries are not trivially mapped to a fully expressive user interface. We propose a mapping from user actions in the form of a visual language, to VC-tree operations. On a high level, we can break down all possible operations into four categories:

1. Map data to charts
2. Map data to channels
3. Modify channel mappings
4. Modify the Visception Tree topology

These four categories indicate what the user interface must do, and what it must look like. The user interface must be able to operate on chart types, channel mappings and the Visception Tree topology, without being too complex to the user. In order to map data to charts, we first need a pane from which we can drag data columns.

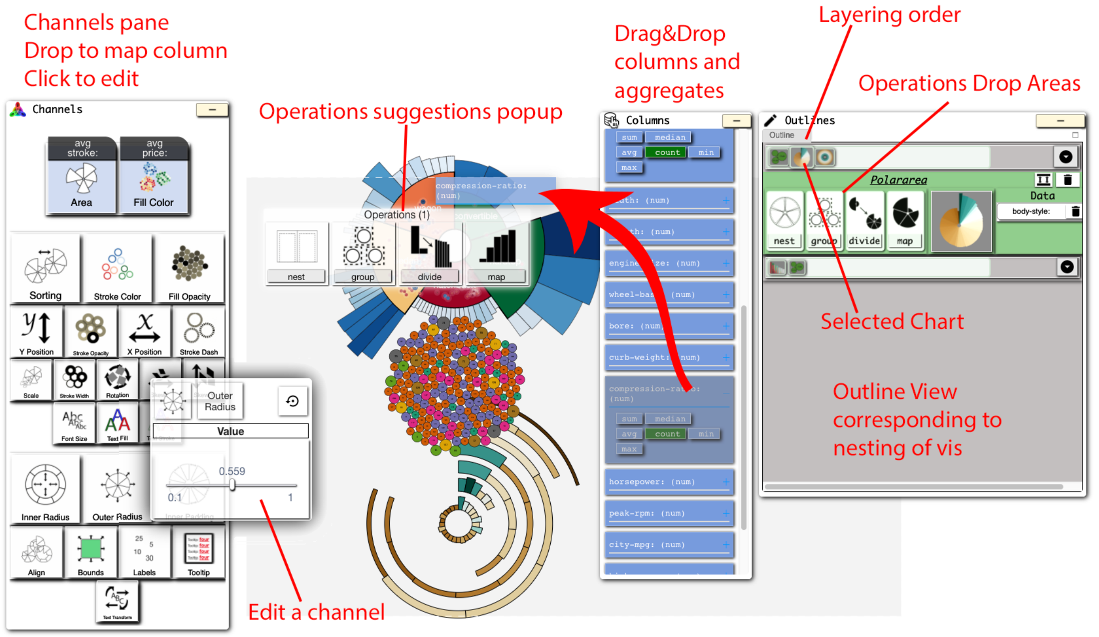


Figure 5.1: An overview of the main user interface functionality. The outline view on the right represents the structure of the Visception Tree, and enables the user to do nesting operations, as well as change the chart on any node. From the columns view we can drag columns, or aggregates and map them to charts or channels. In the channels view we can edit the channel mappings further.

For this we have a simple columns pane embedded in a floating window. From the columns pane we can drag data columns, as well as aggregates to the middle of the screen, or other designated drop areas. Designated drop areas correspond to VC-Tree operations, as well as channels that are mappable to the dragged columns. The Outline View provides a file-tree-like overview of the Visception Tree topology as well as the active charts per node, and exposes available operations. By clicking a node in the Outline View, we select that chart. When a chart is selected, the Channels View will display all available channels. Likewise, the Guides View show available channels for guides (axes and legends). In order to provide an easily accessible and customizable interface, all controls are placed into in floating windows that can be minimized, resized and moved around. A screenshot of the essential parts of the user interface is shown in Figure 5.1. The four most important windows for building a visualization are: The Columns View, the Channels View, the Outline View and the Guides view. With only these windows the user is able to create any nested visualization.

Each view will be explained in greater detail in the coming sections. The views provide fully expressive interaction, while being highly scalable and flexible.

5.1 Controller Views

With highly expressive and flexible design tools there are many options. Usually this leads to options being deeply nested within drop-down menus, making it more difficult to learn how to use the program. In Visception we have exposed all controllers in floating minimizable, resizable, and movable windows. These windows provide more flexibility and available screen space for designing. There are five views: The Columns View, the Channels View, the Outline View, the Guides View, and the Dataset View. Each view, as well as the synergy between them, will be explained in closer detail. Figure 5.2 shows the most important controller views.

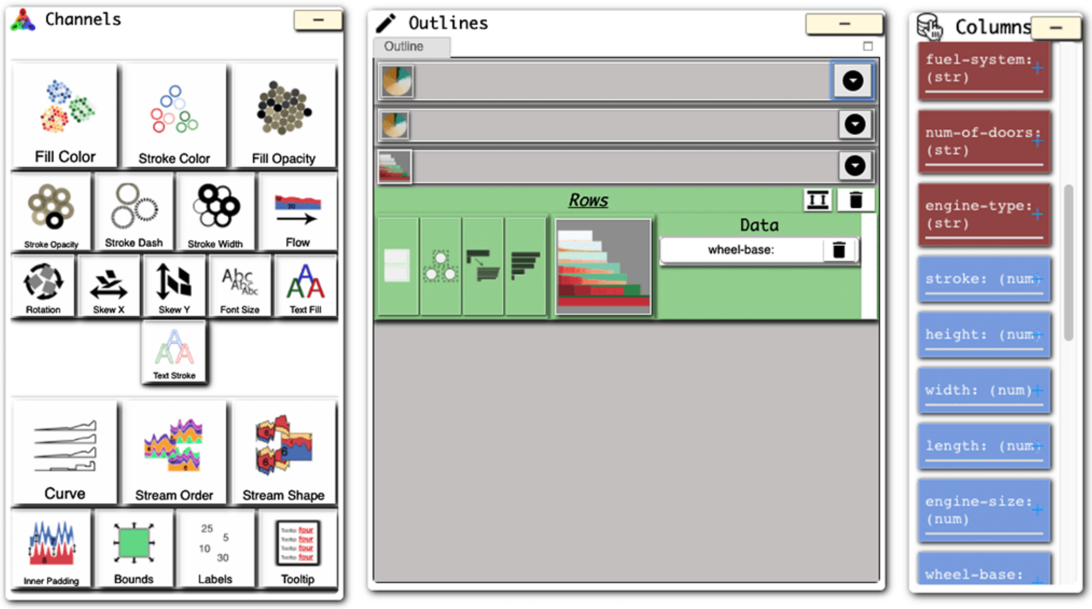
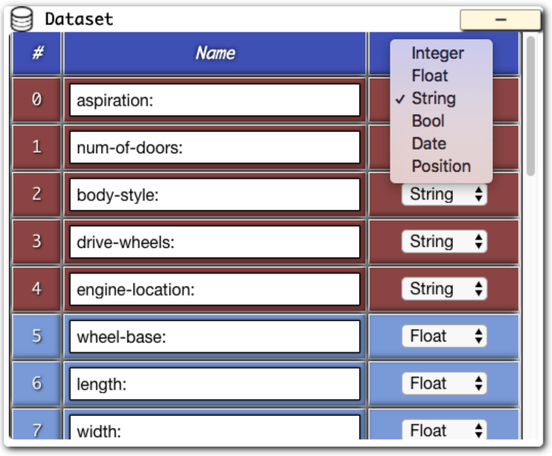


Figure 5.2: An overview of the most important controller views. With these three views we can create basic visualizations. We can drag from the columns view and drop on the Outline or Channels View.

5.1.1 The Dataset View



#	Name	
0	aspiration:	
1	num-of-doors:	
2	body-style:	String
3	drive-wheels:	String
4	engine-location:	String
5	wheel-base:	Float
6	length:	Float
7	width:	Float

Figure 5.3: The dataset view

When the user loads a dataset, Visception automatically infers the types of each attribute. This is done by creating a histogram of types for each data column, then picking the data type according to a rule set. By default the data type occurring the most frequent is picked. One example of a custom rule, is that if a majority integer variable has a few floats, it will be treated as a float. Also attributes such as longitude, latitude, date have a higher precedence than floating numbers or integers. However, the initial guess may be wrong, and the user may want to rename the column. Thus, the Dataset View enables renaming and re-typing of data attributes.

5.1.2 The Columns View

After a dataset has been loaded, the available attributes must be displayed somehow. The columns pane displays all available data columns and is central to the application. Clicking on one data attribute will show all available aggregates for that attribute. Each attribute is an accordion, where each item is an available aggregate. The column and its aggregates may be dragged. Every data mapping action begins with a drag operation from this view. Figure 5.4 shows a screenshot of this view.

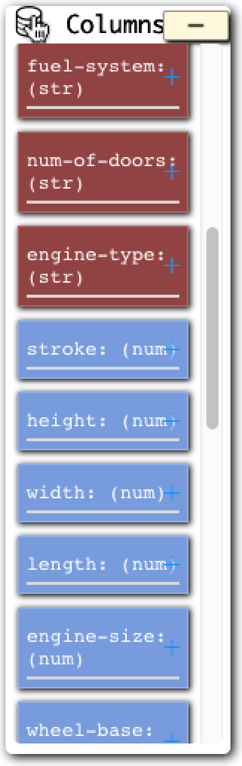
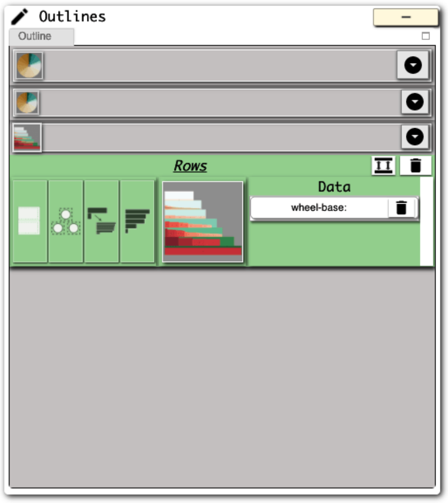


Figure 5.4: The columns view

5.1.3 The Outline View

The Outline View (See Figure 5.5) provides a high-level glance at what charts a visualization consists of. By clicking on a chart in the Outline View, we select that chart. When a chart is selected, the Channels View and the Guides View will be updated accordingly. Furthermore, it shows the current chart type, its available operations (*Nest*, *Group*, *Divide*, and so on) and its children.



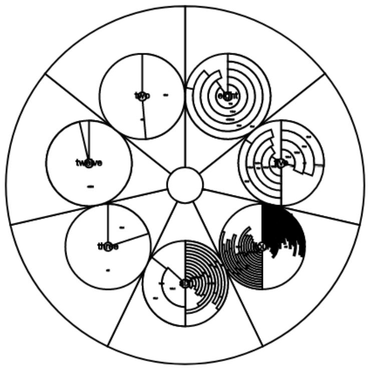


Figure 5.5: The outline view and a corresponding visualization.

5.1.4 The Channels View

The Channels View exposes all channels for the selected chart. Each channel has an icon that gives the user an intuitive idea of what the channel does. Even if the icon does not give a clear enough indication, the user can find out what the channel does within just a few seconds by adjusting it. The channels view consists of a tiled layout, where each tile is a channel. Channels that are mapped to data have a header indicating which data column they are mapped to. Figure 5.6 shows two examples of the Channels View, for a stream and an area chart.

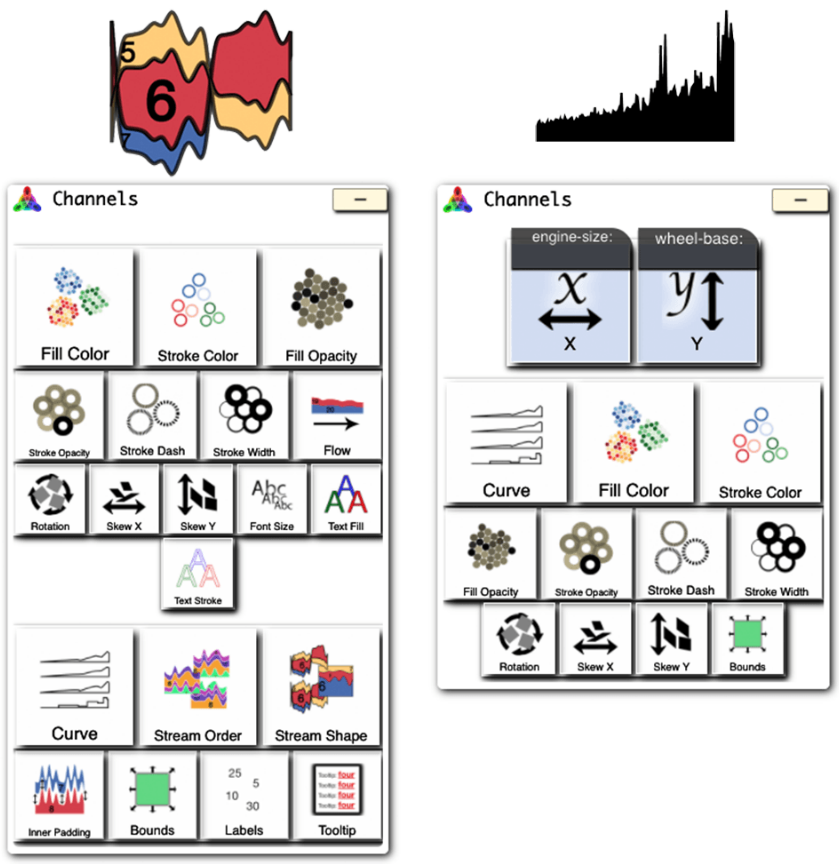


Figure 5.6: The channels view for a ***stream*** chart, and an ***area*** chart. We can see that they have some unique channels, and some channels in common.

5.1.5 The Guides View

Guides are synonym for legends and axes. These are a crucial part of the visualization, and should also be customizable for the user with great flexibility. One problem with guides is that one chart may have two axes - an X and a Y axis. Then, each axis has an identical set of inputs. To solve this, the user must be able to select an outer asset before adjusting it. Thus, there is an outline of the visualization as well as its guide on the top view. In this outline the user can click the guide and get access to its corresponding channels, as shown in Figure 5.7.

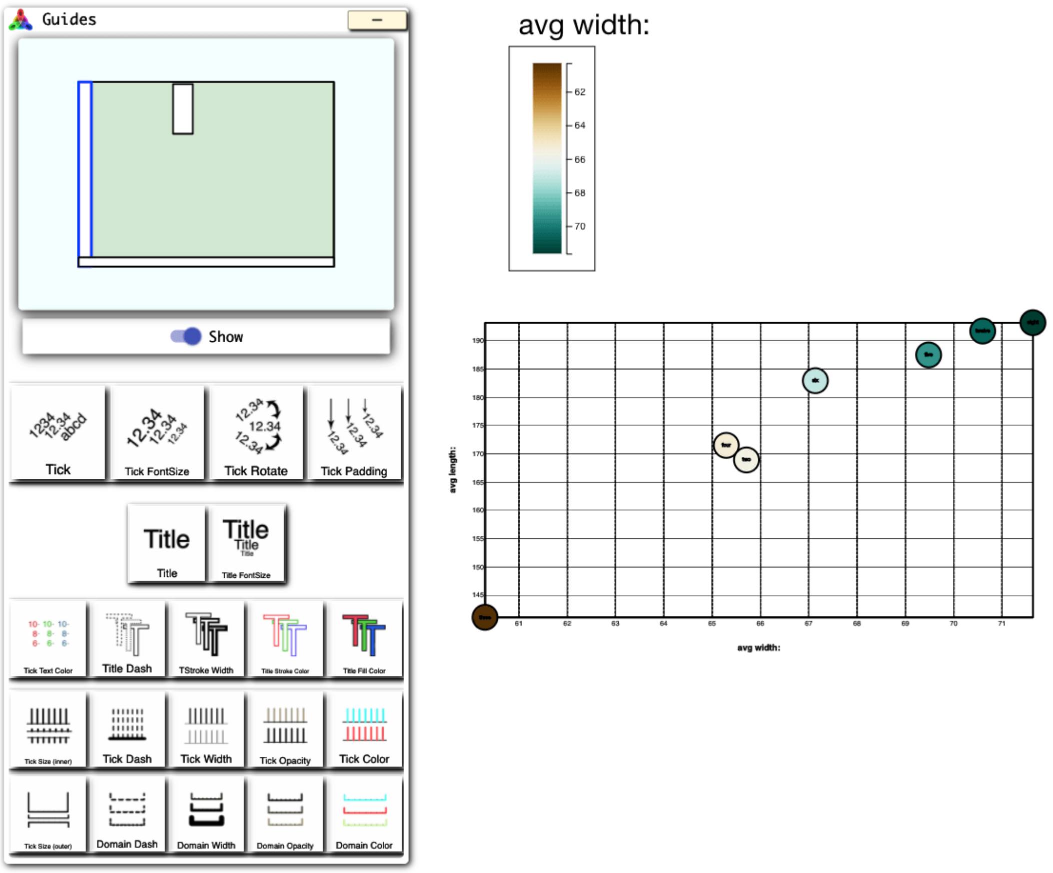


Figure 5.7: The Guides View and a corresponding visualization. Here we enable the user to edit axes and legends at any level of nesting. Each guide has its own set of channels, all editable in the same manner as channels tied to the chart itself.

5.2 Mapping Data

The most basic operation of Visception is to initiate a column drag. When initiating the drag, every potential drop area will be highlighted. This includes possible VC-tree operations, channels, and the viewport.

5.2.1 Mapping data to charts

The first step to create a visualization with Visception is to add one VC-node. In order to add a VC-node, only one input is required: a data column. To express creating a VC-node as well as mapping a data column *D* to it, we drag a column from the columns pane and drop it on the middle of the screen. This action implicitly creates a chart with default settings. The chart type can be changed in the outline view.

5.2.2 Mapping data to channels

To customize a chart, the user most likely wants to map data columns to certain channels. The user needs to see all of the possible channels that can be mapped to a column without getting overloaded. To facilitate this, every channel that accepts data is highlighted on drag start. This instantly exposes all channels that can accept a data column. When a column is dragged over a channel, the chart will show a preview of the mapping. This mechanism enables the previewing of a great number of potential channel mappings within seconds.

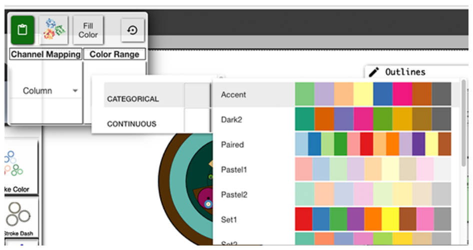
5.3 Modifying the Visception Tree

In order to create a nested visualization, one chart must be nested within another. In order to accomplish this, a new VC-node is created, and appended as a child to the target node. Visually, we achieve this by dropping a column on the *Nest* operator area. By dropping a column, a VC-node with the chart type of the parent node is implicitly created. The chart is then edited in the outline view. Grouping a chart by a data attribute *D* will show one instance of that chart for each distinct value of the domain of D. Dropping a column on *Group* operator area will group a node by a column. Implicitly, a new VC-node is created, mapped to the new column and set to the chart type of the child (unless the child is not nestable, then it defaults to a nestable chart). The *Layer* operation is used to overlay visualizations. It is performed in the same way as the *Group* and *Nest* operations. In order to delete a node, there is a delete button which will delete the node as well as all of its children. The outline view allows the user to browse the tree topology and select a single node. The layout is similar to the conventional way of browsing a file system. This allows for easily inspecting and editing the contents of one VC-node at a time. The layer order can be rearranged by dragging the nodes in the Outline View.

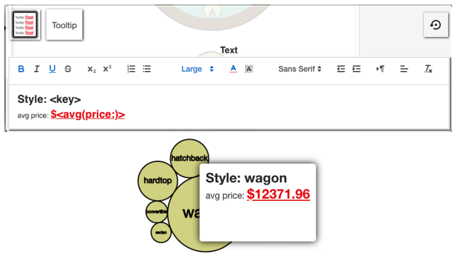
5.4 Editing Channel Mappings

When programming, certain values will be specified for certain attributes. For example, for the **Fill Color** channel the output domain is all valid colors, while the **Stroke Width** channel the output domain is a certain numeric range, or a numeric value. The channels view partially automates this by having a preset domain and an adjustable range. So, instead of having to type in a color value or numeric value, it is adjusted by a value slider or range slider.

Customizing a single chart is done by editing its channels. By simply dropping a column on a channel in the Channels View, the channel will be mapped to that column. A drop-operation will be automatically previewed by simply dragging the column over the channel. This enables the user to look through outcomes of mapping the column to all channels within seconds. In order to edit the finer details of the channels, the user can click a channel icon to see all the available controls. Some channel mappings are controlled by a single slider, while some have more complex visual controls. For example, numeric ranges are edited through range sliders, color values are chosen with a color picker, and so on (See Figure 5.8).



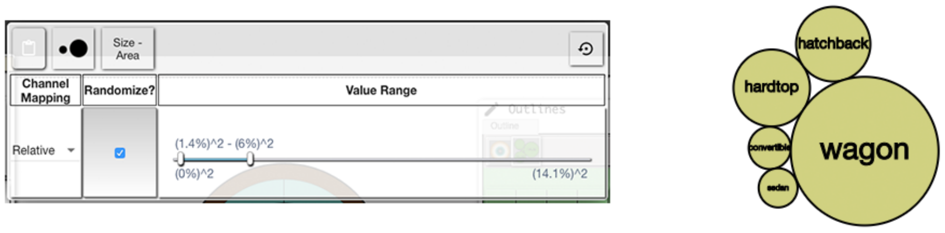
(a) Choosing a color range.



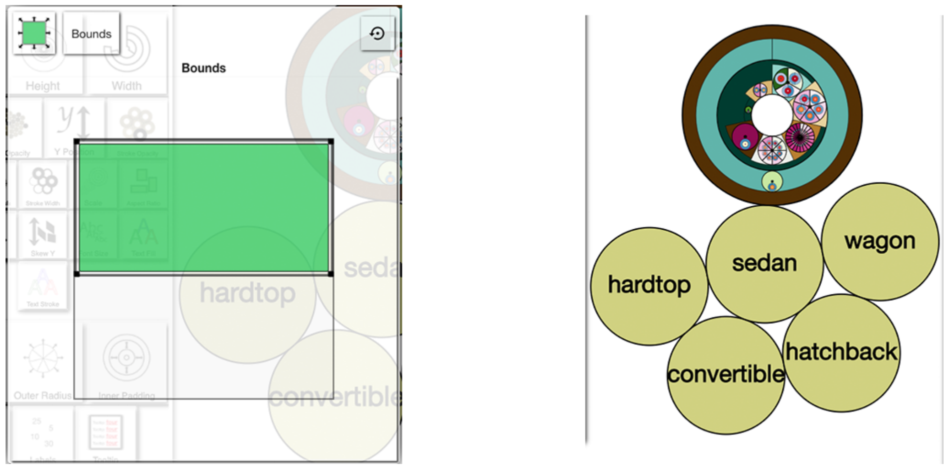
Style: <key>
avg price: $<avg(price:)>

Style: wagon
avg price: $12371.96

(b) Editing the tooltip text and styling.



(c) Editing the **Size** channel of a *Cartesian* ***circles*** chart.



(d) Editing the bounds of a chart.

Figure 5.8: Some examples of channel editing cards. The editor card appears when clicking an icon.

Summary

In this chapter it has been demonstrated how a set of controller views can provide full, flexible access to charts and their individual channels, as well as the tree topology. The Dataset View both enables and assists the user to set the types of each data column. After the types are set, the Columns View presents columns, as well as their available aggregates to the user. The user may drag aggregates, or column from the columns pane and instantly see the possible drop targets highlight. If a column is dropped on the middle of the screen, a corresponding chart is created. The Outline View shows the available operations per chart, as well as a file-tree like outline of the tree topology. In the Outline View, the user can select a VC-node, edit or change the chart, as well as rearrange the tree topology and delete nodes. When a single chart is selected, the Channels View will display all available channels for that chart. Each channel has an icon, indicating what it does, and displays a custom controller card when clicked. Channels that accept data are highlighted on drop. Likewise, the Guides View lets the user control the appearance of axes and legends. Each guide has a set of channels, and is edited in the same way charts are edited. The Columns View is the source of most drag actions, while the other views are the receivers.

Chapter 6

Implementation

At the time of writing, Visception is about 30,000 lines of code without whitespace and comments and there are plenty of improvements that can be made. The biggest challenge with this system is that it grew very large and had to be refactored many times for it not to become unmanageable and cluttered.

6.1 Architecture

The architecture has been an important part of making all the components work together without becoming too complex to manage. The user interface and the underlying logic is fully separated. In practice, the Visception core could be exported as a standalone library. In broad terms, the Visception core consists of a data management part, as well as a language part – encompassing charts, channels, layouts as well as the Visception Tree. The Visception core allows for creating new visualizations, and provides hooks for adding, deleting, fetching, querying and editing charts. These hooks connect to the user interface - providing a clean separation of logic. Figure 6.1 depicts a diagram of this overview.

The architecture must also facilitate optimizations addressing performance bottlenecks. Major potential performance bottlenecks in a system like Visception are: constant querying of data and re-rendering cycles. To still enable rapid interaction, Visception only lazily evaluates transforms and channel changes. With this minimal

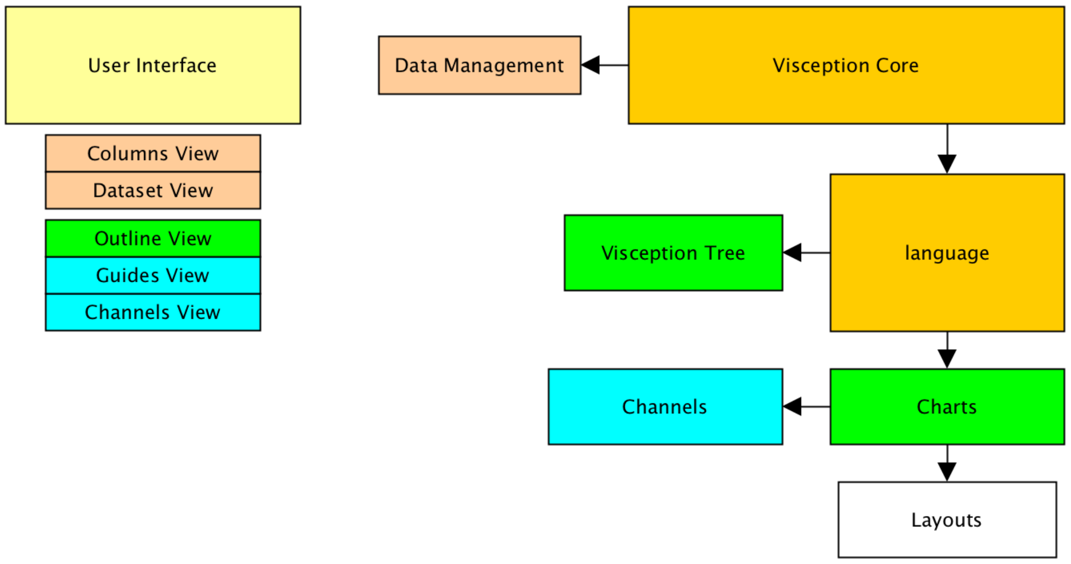


Figure 6.1: A high level outline of the architecture of Visception. Boxes that are colored similarly are related. For example, the Columns View and the Dataset View "connect" to the Data Management part of the core, while the Channels View and Guides View are "connected" to the channels.

evaluation, data querying and rendering cycles are kept to a minimum.

Running nested layouts can be highly computing intensive. To address this we defined each layout as a pipeline with certain steps. Our Visception Tree is stateful, meaning we only render one chart at a time as we iterate through the parent data. In the pipeline we refer to *local steps* as steps dependent on the selected parent datum, and *global steps* as steps independent of the selected parent datum. Each pipeline consists of local and global steps, varying for each layout. With this pipeline architecture we only have to re-run the absolutely necessary part of the layout, thus saving more computing power.

6.2 The Visception Tree

The Visception Tree itself is implemented as a plain tree structure, where each node is enumerated by a depth and an index from doing a level-order tree traversal. Nodes are put in a 2D table indexed by depth and index. Whenever the tree changes the node table is updated. Each node has read-access to its parent node as well as its children. Figure 6.2 illustrates the general topology of a Visception Tree, as well as managers encapsulated the contents of each node.

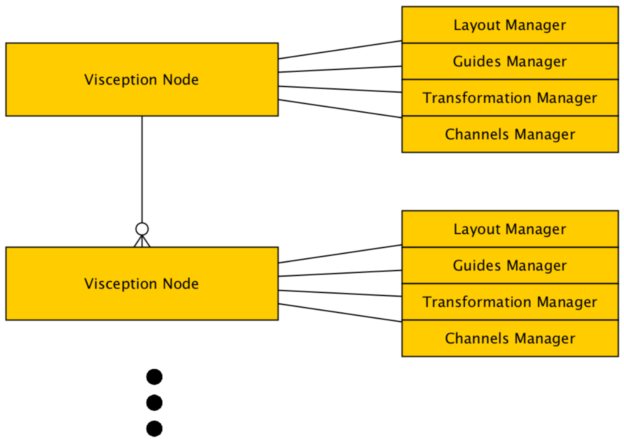


Figure 6.2: The architecture of the Visception Tree. Each node has a manager for its layout, guides, transformations and channels.

Since many channel mappings have much in common, they are reduced to a smaller set of general channels. For example, a channel may be numeric, and take in a numeric range plus a column or a numeric value. Such numeric channels are coded as a general channel.

A specified set of equivalence groups between charts enables for transferring channel states when changing charts. For example, bar length, circle area, sector area size are in the same group, so if bar length is mapped to a column, the mapping will be transferred to the channel in that equivalence group.

6.3 User Interface and External Libraries

The user interface code is written in AngularJS, and the Visception library itself is written in ES6. Consequently, the interface and Visception library is loosely coupled. The AngularJS controllers fetch relevant information from the Visception Tree data structure.

To enable undo/redo mechanism for all actions, each action performed through the user interface is stored in an action manager class. Each action is represented as a state change to the visualization. The action manager automatically removes duplicate/overlapping actions, and trims away redundant information from the objects incrementally. The action history is exportable as a plain string. This allows for users to save their progress.

For data query processing we used CrossFilter [18], Reductio [30] and Vega Datalib [22]. When a column is dropped, a corresponding query is created and fetched. Visception relies heavily on D3's built in data binding system, especially for looking up relevant parent items when doing nested layouts. Lodash was used for basic object and array operations. To facilitate a modular design, we used Browserify for the bundling of Javascript modules. Hence, the distribution is one JS file referred to by the browser.

Chapter 7

Results

Using Visception enables us to create visualizations quickly and easily. To demonstrate this we will show how some non-trivial visualizations are made and what steps are involved.

7.1 At the National Conventions, the Words They Used

This visualization was published by The New York Times in 2012. It shows which words were mentioned at which conventions, but also how many times they were said, and which party said it the most. To recreate this, we took a similar dataset from a transcript of one speech by Donald Trump, and one by Barack Obama. Figure 7.1 shows the end result.

1. Looking at the NYT visualization, we see that there is one bubble for each word. The first step is thus, to create one bubble for each word. To do this we can simply drop the *word* column mid screen. This creates one circle for each distinct value of the *word* column. (See Figure 7.2(b))

2. Looking more closely at the NYT chart, there are two mappings that can be inferred intuitively. Each word is sized proportionally to its number of mentions, and the more republican-dominated words are further to the right. The next step is to recreate these mappings. To position the circles, we drop *percentage(trump)* on **Position X**. Then, to size the channels we drop *sum(count)* on

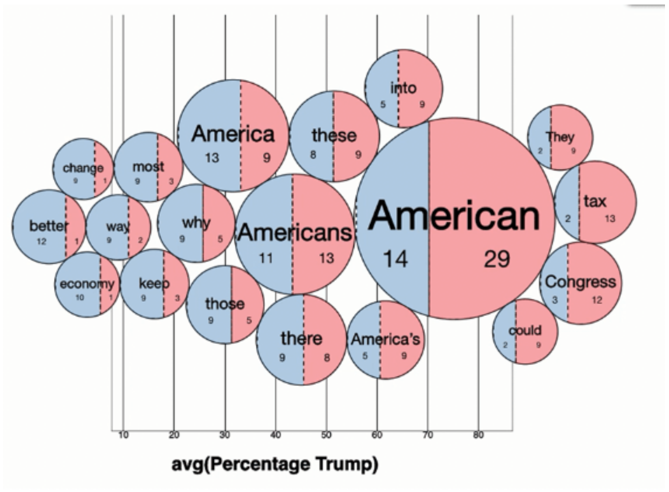


Figure 7.1: The final result made with Visception. Each circle represents one word. Inside each circle we see clipped bars, with a width proportional to the number of mentions. By displaying the same dimensions using multiple channels, the data becomes much clearer.

Size. *sum(count)* is the same as the number of mentions for each word. (See Figure 7.2(c))

3. Now we have a set of circles that are sized and positioned as desired. Somehow we need to subdivide these circles by who has mentioned it the most. To approximate this we will use the ***columns*** chart. The rationale behind this, is that if we can put a column chart inside each circle, and size it so that it "covers" the entire circle - we can clip it and map the **Bar Width** channel to the number of mentions. Hence, we begin by nest the column *name* within the current chart, then we set the chart type to ***columns***. (See Figure 7.2(d))

4. To expand the columns to cover their parent circles, we edit the bounds of the column chart. Intuitively this can be thought of as "stretching" the column chart to cover the circles. (See Figure 7.2(e)

5. Next, we map *name* to the **Fill Color** channel, and *sum(count)* to the **Bar Width** channel. Additionally we edit the **Text Label** channel to display *sum(count)* – the number of mentions of each word, implicitly broken down by party. (See Figure 7.2(f))

6. Currently the column chart is overflowing the circle bounds, in order to clip it we enable the **Clip** channel. (See Figure 7.2(g))

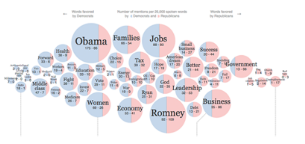
7. Next, we see that the colors are much brighter than in the original. To achieve a more similar color scheme, we adjust the **fill opacity** channel. The visualization is now close to identical to the original. (See Figure 7.2(h))

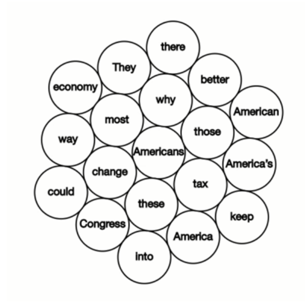
8. Instead of leaving it at this, we will look at some other variants of this visualization. We decide that we want to see a pie chart inside each circle, instead of clipped bars. Thus, we change the chart type to ***sectors***, and map *sum(count)* to the **Area** channel. Then, we see that the text of parent circles is overlapping the text of the pie chart slices. To fix this, we edit the **Text Transform** channel and move the parent labels slightly up. (See Figure 7.2(i))

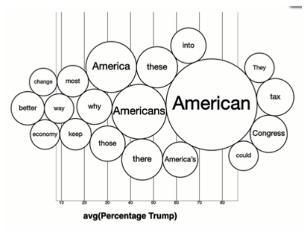
9. To see what a ***polar area*** within each circle might look like, we set the chart type to ***polar area***. (See Figure 7.2(j))

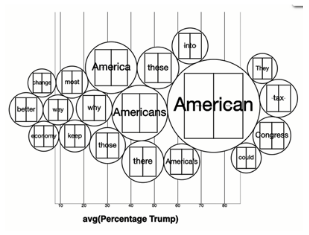
10. Since there are more options, we now try change the chart type to ***rows***. (See Figure 7.2(k))

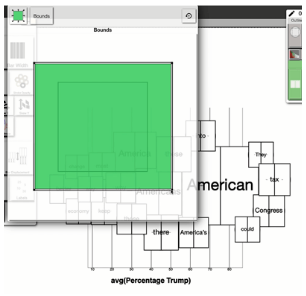
11. We keep exploring, and change the chart type to *tubes* with one click. (See Figure 7.2(l))

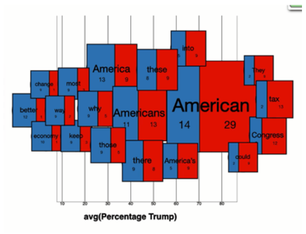

(a) The original visualization by The New York times

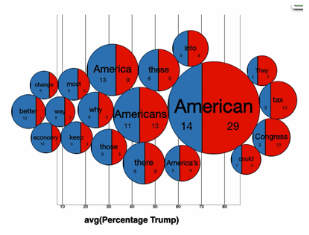

(b) Drop column *word*

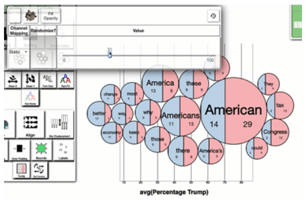

(c) Drop *percentage(trump)* on **Position X**, and *sum*(*count*) on **Size**.

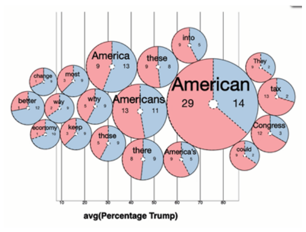

(d) Nest *name* within current chart. Change chart type to ***columns***.

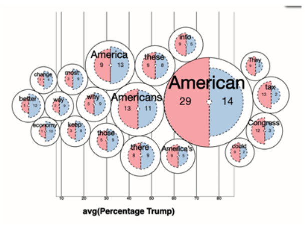

(e) Edit columns **Bounds** channel, expanding to cover the parent circles.

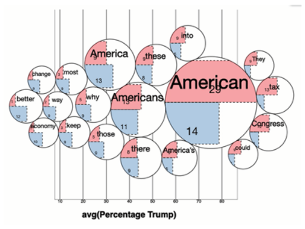

(f) Map *name* to **Fill Color**, and *sum(count)* to **Bar Width**.


(g) Edit (enable) **Clip** channel


(h) Edit **Fill Opacity**


(i) Change leaf chart to ***sectors***. Edit **Text Transform**


(j) Change leaf chart to ***polar area***


(k) Change leaf chart to ***rows***

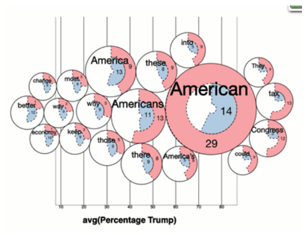

(l) Change leaf chart to ***tubes***

Figure 7.2: These visualizations were made in 2 minutes and 30 seconds (includes loading the data).

7.2 Dimensional Stacking

This visualization will depict a snapshot of the Visception code base from late 2017. It will show which folders contain most lines of code, as well as the amount of code, comments and blank lines per file. To achieve this, ***streams*** will be nested within ***squares***, ***bars*** and ***arcs***.

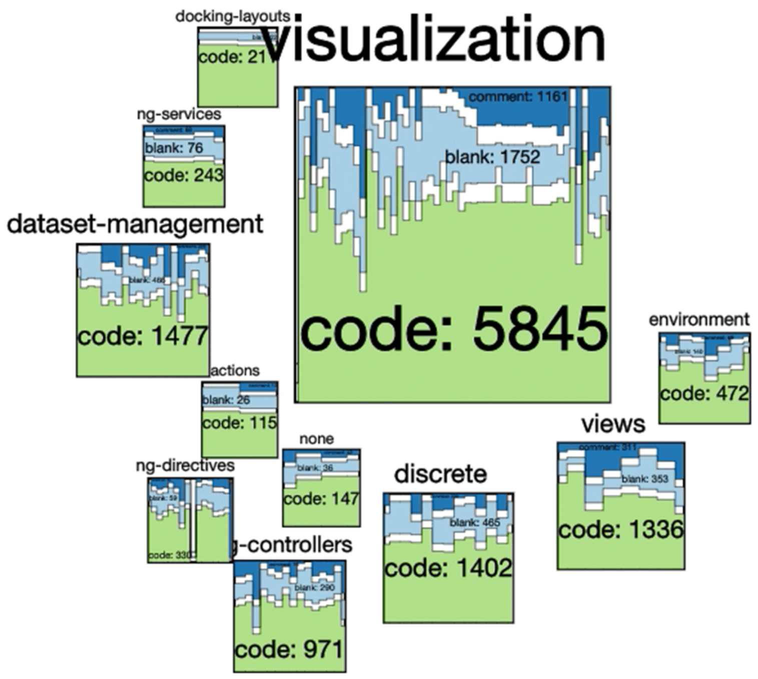


Figure 7.3: Visualization of an old Visception source code snapshot. Each square represents a folder, sized by the amount of code its files contain. Within each square, we see the distribution between code, comments and blank spaces. Using a stepped curve interpolation also gives us a rough indication how many files are in each folder.

1. Knowing that the "top level" of the visualization is the folders, we will create one square for each folder. This can be achieved by dropping the *folder1* column mid screen, then setting the chart to ***squares***. (See Figure 7.4(a))

2. The squares are all equally sized. To illustrate how much code is within each square, we can map *sum(code)* to the **Size** channel. We also want the labels to

not occlude the child charts, this is addressed by editing the **Text Transform** channel. (See Figure 7.4(b))

3. The next thing we want to see is ***streams*** within each square. To achieve this, we must first create a nested node. To achieve this, we nest the *code* column within the current chart. This only creates one square for each distinct value of *code*, which is not what we want, but it is simply an intermediate step. (See Figure 7.4(c))

4. We set the chart type to ***streams***, which creates one stream for *code*, where each step in the stream represents one file. (See Figure 7.4(d))

5. Since the goal is to see the amount of code, comments and blank lines, we must create one stream for each of these attributes. We push the columns *blank* and *comments* onto the stream, using the push operation. The push operation is only available for series charts and is equivalent to divide. This creates one stream for $blank$, and one stream for $comments$. (See Figure 7.4(e))

6. With only white streams, it is difficult to differentiate between them. To address this, we edit **Fill Color** to show one color per column. This channel mode is only available for streams. Next, we edit the **Text Label** input to *'<key>: <sum(key)>'*, and set the **Stream Shape** to **step**, which exposes the amount of files more clearly. (See Figure 7.4(f))

7. While the size does indicate the total amount of code, it can be made more clear. Thus, we horizontally sort the squares by *sum(code)*. To achieve this, *sum(code)* is mapped to **Position X**. (See Figure 7.4(g))

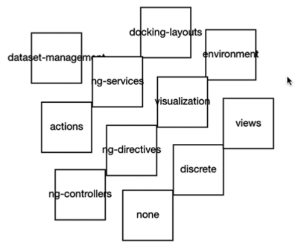
8. To exploring some more options, we will replace the squares with columns. Thus, we set the parent chart to *columns*. Note how the **Size** channel propagates its mapping to the **Bar Height** channel, and **Position X** to **Bar Order**. (See Figure 7.4(h))

9. Next, we check if ***rows*** display it better. The parent chart is set to ***rows***. The mappings are automatically transfered from the previous chart. (See Figure 7.4(i))

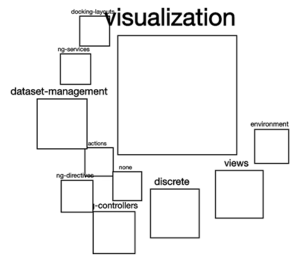
10. Exploring more, we set the parent chart type to ***sectors***. Note how the ***Size*** and ***Order*** is transferred. Out of curiosity, we also set the **Stream Shape** to **minWiggle**[9] (See Figure 7.4(j))

11. Since there are more pie charts, we test the ***polar area***, though it is immediately clear that this is harder to read than the previous chart.
(See Figure 7.4(k))

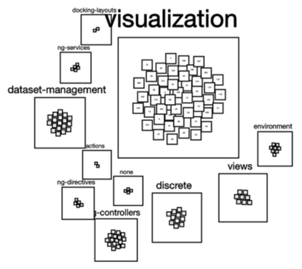
12. To make it more readable, we set the **Stream Shape** to **expand**. (See Figure 7.4(l))



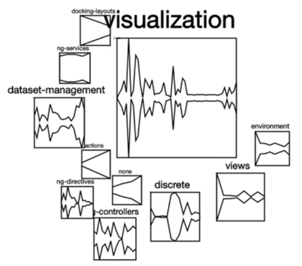
(a) Drop column *folder 1* and set chart to ***squares***.



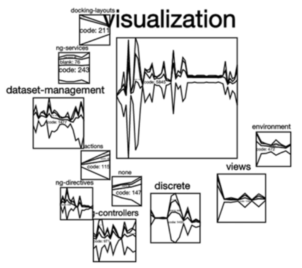
(b) Drop *sum(code)* on **Size**, and edit **Text Transform** channel.



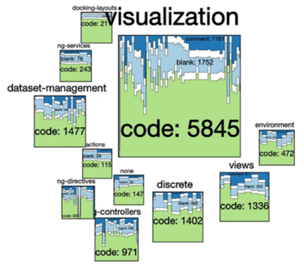
(c) Nest *code* within current chart.



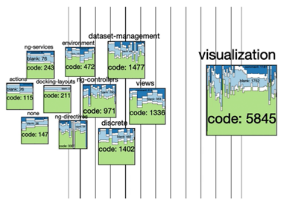
(d) Change chart type to ***streams***.



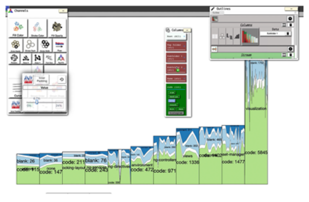
(e) Push columns *blank* (lines of white space) and *comments* onto the leaf node.



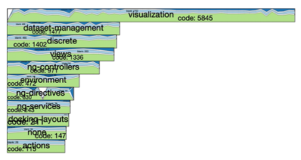
(f) Edit **Fill Color, Text Label, Stream Shape** (**Stream Shape** maps to a d3.stackOffset).



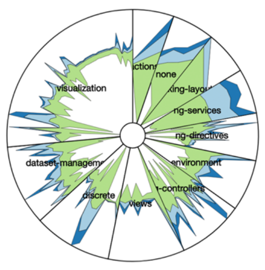
(g) Map *sum(code)* to **Position X**



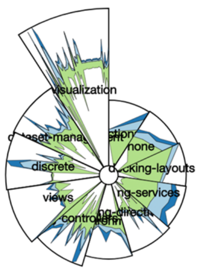
(h) Change parent node chart to ***columns***



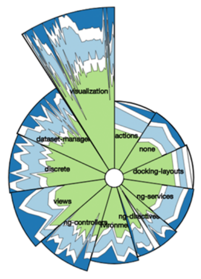
(i) Change parent node chart to ***rows***



(j) Change parent node chart to ***sectors***. Note how $sum(code)$ is mapped to **area**



(k) Change parent node chart to ***polar area***.



(l) Edit leaf node **Stack Padding Stream Shape**

Figure 7.4: These visualizations were made in 5 minutes.

7.3 Loans

Here we will analyze a loans dataset from Kaggle (www.kaggle.com) without using the *nest* operation, but rather only the *group* operation. Here we will compare how loan recipients pay down their loans, broken down by education and gender. The final result can be seen in Figure 7.5

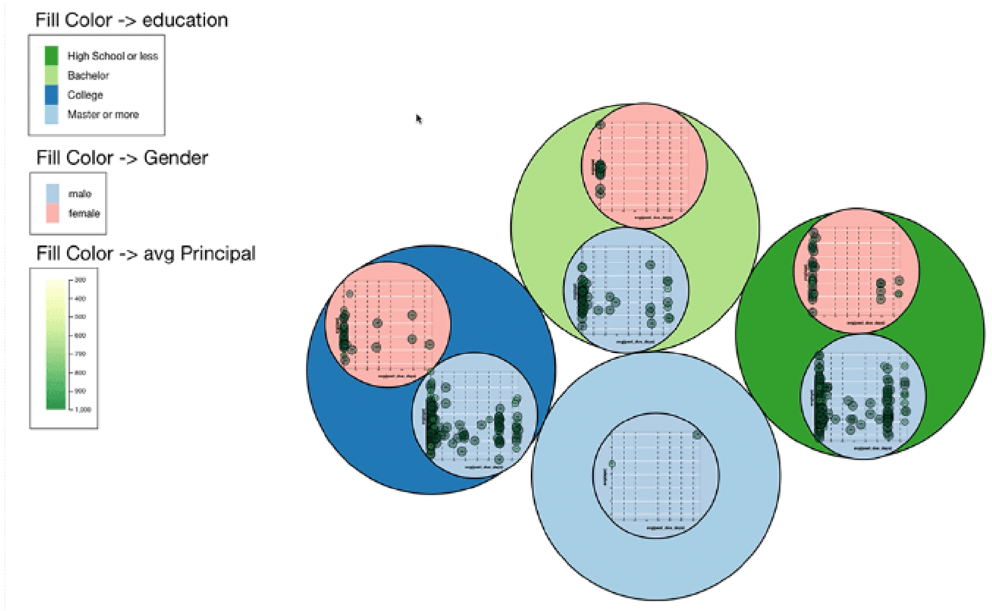


Figure 7.5: A breakdown of loan payments by education and gender. The outermost bubbles represent level of education. The innermost bubbles represent the value of the loan, while the bubbles "in the middle" represent genders. From this we can see that men tend to go overdue more often. It is also apparent that we have a very small sample of highly educated people in this dataset.

1. First we will create one circle for every loan. To do this we use the *LoanID* column. Though we could also use the *root* column which is equivalent if there is only one row per *LoanID* value. After dropping the *LoanID* column on the screen we see a bubble chart (See Figure 7.6(a))

2. Next, we want to group all the loans by level of education. Intuitively, we can imagine separating the bubbles and grouping them by level of education. By dropping the *Education* column on the group operation we are creating one bubble for every education level, and also showing one bubble per *LoanID*

inside every *Education* bubble. At the tree topology level this is equivalent of inserting the node representing the *Education* chart as the parent of the node representing the *LoanID* chart. (See Figure 7.6(b))

3. Now that we have grouped it by *Education* and *LoanID*, we want to see the breakdown by gender. To do this, we do a *group* operation on *LoanID*. This is equivalent of inserting a node between the two already existing nodes so that the grouping level is *Education*, *Gender*, *LoanID*. We can see that the sizing of the bubbles is not optimal, which we can fix by adjusting the **Size** channel of the *Gender* chart. (See Figure 7.6(a))

4. Now we can adjust the colors. For the root node, i.e *Education* we map the *Education* column to **Fill Color**. For the middle node we map *Gender* to the **Fill Color**. We now have the color blue representing males, and red representing females. For education levels we have one color for each education level. (See Figure 7.6(d))

5. Next we want to see how large each loan is. For this we have the *Principal* column representing the size of the loan. On the leaf node representing *LoanID*, we map the *Principal* column to the **Fill Color** and **Size** channel. Larger, greener circles represent larger loans. (See Figure 7.6(e))

6. Not everyone pays their loans on time. We want to see how this plays out for the current grouping, as well as how this relates to different age groups. To achieve this, we map *PastDueDays* to **X Position**, and *Age* to **Y Position** (See Figure 7.6(f))

7. By default Visception uses a force layout for ***circles***, however we can adjust it to a plain Cartesian layout by adjusting the **Collision** channel to 0. Doing this we have a Cartesian layout with the nodes at their exact positions. With a force layout the **Position X** and **Position Y** represent the centers of gravity of each node. (See Figure 7.6(g))

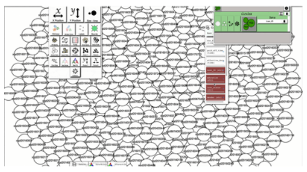
8. One problem with the current layout is that circles occlude one another, so we cannot see all of them or get an idea of how many circles there are. To solve this we adjust the **Fill Opacity** and **Stroke Width** channels. (See Figure 7.6(h))

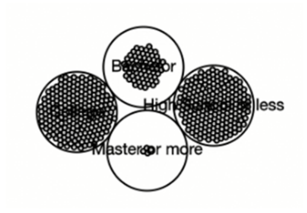
9. With nested axes it is easy for the ticks to become too apparent. To fix this, we adjust the **Tick Width** of the x-axis down. (See Figure 7.6(i))

10. The Y-axis ticks are also too apparent, and not as distinguishable from the X-axis ticks as we would like. To fix this we adjust the **Tick Width** and **Tick Dash** channels of the Y-axis, as well as the **Tick Color** - making the ticks white. (See Figure 7.6(j))

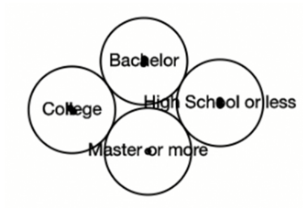
11. Now we can zoom in and look at every nested chart separately. (See Figure 7.6(k))

12. We adjust the **Tooltip Text** channel to display the principal of the hovered circle in a nicely formatted manner. (See Figure 7.6(l))

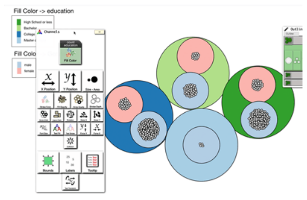

(a) Drop column *LoanID* mid viewport.

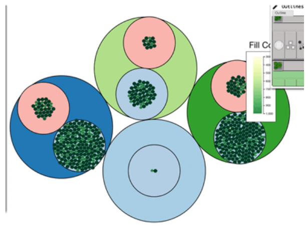


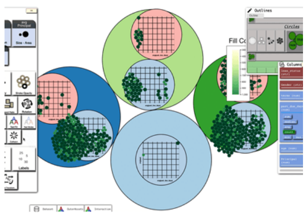
(b) Group *Education*

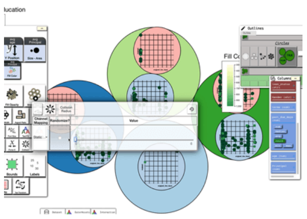


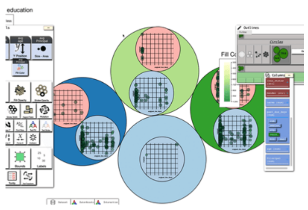
(c) Group *Gender*.

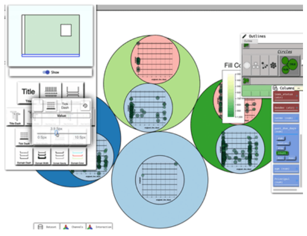

(d) Root node: Map *Education* to **Fill Color**. Mid node: Map *Gender* to **Fill Color**.

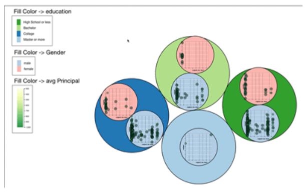

(e) Leaf node: Map *Principal* to **Fill Color, Size**.

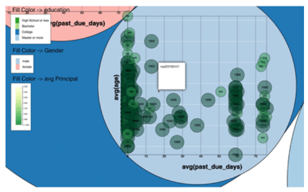

(f) Map *PastDueDays* to **X Position**, *Age* to **Y Position**.


(g) Edit **Collision**, set to 0. Equivalent to using a plain Cartesian layout

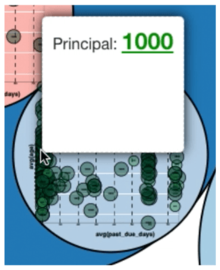

(h) Edit **Fill Opacity, Stroke Width**


(i) Change **Tick Width** and **Tick Dash** for x axis


(j) Change **Tick Width**, and **Tick Color** for y-axis.



(k) Zooming in to inspect further.



(l) Edit **Tooltip channel**

Figure 7.6: This visualization was made in 5 minutes and 10 seconds.

Chapter 8

Discussion and Limitations

While we have demonstrated that Visception is capable of creating visualizations really fast, performance is still something to consider. For example, with larger datasets, interaction will only stop up and slow down the process of exploring visualizations. This could be addressed by doing "soft updates" only on a part of the visualization or for a part of the dataset, but this is not implemented at the moment.

The two major factors that affect the performance is dataset querying and rendering. Rendering is easier to optimize than the dataset querying – with our pipeline setup the rendering is already quite fast. The dataset querying is a major issue for larger datasets and requires further work to improve. For example, if an attribute has 10,000 distinct values, the user may try to create a bar chart for that row. Rendering 10,000 rows as rectangles will be disadvantageously slow on most computers. A mechanism to address this could be implemented by displaying a warning message when the user is about to make actions that may crash the browser.

If the dataset is really large, it will take up a lot of memory, as well as time to query. Caching the query results becomes a lot harder if we run out of memory and have to delete already cached queries. One way to address this is to implement a cache management mechanism, or to run the data processing on a separate server. However, we believe that the best solution for this would be to use a GPU database [37] for querying the dataset. With this setup we could simply load a 1GB file into GPU memory and have the main memory remain close to untouched, and have high performing queries.

While performance is important, in most cases Visception still enables users to prototype visualizations far faster than they can with other solutions or by coding. Usually, once the user has made one visualization that is adequate, it is not a big deal if it takes a few seconds to render, especially if it is static.

Adding new charts to the system was not always trivial. The pipeline greatly simplified and streamlined the process of adding nestable layouts, however we still need to specify how the layout is nested within different kinds of spaces. Our current solution has much room for improvement and simplification. However abstracting this to a too high level (e.g. a generic nesting behavior for an arbitrary coordinate system) may introduce more complexity than it removes.

On flexibility, there are still things that can only be done with code and not with Visception. One way we can address this is to allow users to write their own D3 code to be executed on the internal D3-selections after rendering. This would introduce a bit of complexity to advanced users, but may still be a viable solution to providing full flexibility combined with rapid prototyping.

Chapter 9

Conclusion and Future Work

In this thesis we have presented a novel way of designing nested as well as non-nested visualizations. By providing a data structure that is easily mappable to user interface actions while still allowing for great flexibility, we have enabled non-coders to design previously unavailable visualizations.

By building on the previous work done on formal graphics specification, programming toolkits and information visualization we have developed a formalism for creating visualizations by using drag and drop operations. Our method builds on all of this previous work. We have shown how we can simplify different table arrangements by requiring less inputs to get started, and how these intermediary steps can help simplify the process of creating charts.

We have gone through the different kinds of spaces that are needed to make different chart layouts nestable within other spaces. These different spaces are crucial to mathematically place one layout within another one. With nesting in place, we have shown how the user can interactively construct and edit a Visception Tree, and how these changes look in terms of a visualization as well as topologically. We have also shown how nested visualizations can be complicated, but also how we can simply edit one chart at a time by selecting one VC-Node in the Outline view. To facilitate all of the operations and interactions, we have gone over the crucial parts of the user interface, how it works and the rationale behind the design.

The project has been implementation-heavy, and would be suitable for future work. Especially within visualization recommendation, Visception would be a good system to use as a basis, since it could be built on top of the already existing structures.

One unsolved problem, however, is to abstract the nesting to an even higher level. If we had fixed definitions for multiple coordinate spaces, we could specify nesting behaviors declaratively. However, we are not sure if this would introduce more complexity than it would solve, since there are unexplored, possibly complex edge cases for this problem.

Bibliography

[1] C. Ahlberg and E. Wistrand. IVEE: An environment for automatic creation of dynamic queries applications. In *Proc. CHI*, pp. 15–16, 1995. doi: 10.1145/223355.223381

[2] A. Aiken, J. Chen, M. Stonebraker, and A. Woodruff. Tioga-2: a direct manipulation database visualization environment. In *Proc. International Conference on Data Engineering*, pp. 208–217, 1996. doi: 10.1109/ICDE.1996.492109

[3] A. O. Artero, M. C. F. de Oliveira, and H. Levkowitz. Uncovering clusters in crowded parallel coordinates visualizations. In *Proc. IEEE InfoVis*, pp. 81–88, 2004. doi: 10.1109/INFVIS.2004.68

[4] F. Bendix, R. Kosara, and H. Hauser. Parallel sets: visual analysis of categorical data. In *Proc. IEEE InfoVis*, pp. 133–140, 2005. doi: 10.1109/INFVIS.2005.1532139

[5] J. Bertin. *Semiology of Graphics*. University of Wisconsin Press, 1983. doi: 10.1080/00690805.1987.10438353

[6] M. Bostock and J. Heer. Protovis: A graphical toolkit for visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1121–1128, 2009. doi: 10.1109/TVCG.2009.174

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

[8] M. Bruls, K. Huizing, and J. van Wijk. Squarified treemaps. In *Proc. VisSym*, pp. 33–42. Press, 1999.

[9] L. Byron and M. Wattenberg. Stacked graphs – geometry & aesthetics. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1245–1252, 2008. doi: 10.1109/TVCG.2008.166

[10] E. H. Chi. A taxonomy of visualization techniques using the data state reference model. In *Proc. IEEE InfoVis*, pp. 69–75, 2000. doi: 10.1109/INFVIS.2000.885092

[11] G. Chintalapani, C. Plaisant, and B. Shneiderman. Extending the utility of treemaps with flexible hierarchy. In *Proc. International Conference on Information Visualisation*, pp. 335–344, 2004. doi: 10.1109/IV.2004.1320166

[12] J. H. T. Claessen and J. J. van Wijk. Flexible linked axes for multivariate data visualization. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2310–2316, 2011. doi: 10.1109/TVCG.2011.201

[13] W. C. Cleveland and M. E. McGill. *Dynamic Graphics for Statistics*. CRC Press, Inc., Boca Raton, FL, USA, 1st ed., 1988.

[14] G. M. Draper, Y. Livnat, and R. F. Riesenfeld. A survey of radial methods for information visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(5):759–776, 2009. doi: 10.1109/TVCG.2009.23

[15] N. Elmqvist, T.-N. Do, H. Goodell, N. Henry, and J.-D. Fekete. ZAME: Interactive large-scale graph visualization. In *Proc. EEE PacificVis*, pp. 215–222, 2008. doi: 10.1109/PACIFICVIS.2008.4475479

[16] N. Elmqvist, P. Dragicevic, and J. D. Fekete. Rolling the dice: Multidimensional visual exploration using scatterplot matrix navigation. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1539–1148, 2008. doi: 10.1109/TVCG.2008.153

[17] N. Elmqvist and J. D. Fekete. Hierarchical aggregation for information visualization: Overview, techniques, and design guidelines. *IEEE Transactions on Visualization and Computer Graphics*, 16(3):439–454, 2010. doi: 10.1109/TVCG.2009.84

[18] M. B. Ethan Jewett, Jason Davies. Crossfilter, fast n-dimensional filtering and grouping of records. https://github.com/crossfilter/crossfilter, 2018.

[19] B. Fisher. *Illuminating the Path: An R&D Agenda for Visual Analytics.* 01 2005.

[20] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software Practice and Experience*, 21(11):1129–1164, 1991. doi: 10.1002/spe.4380211102

[21] S. Gratzl, N. Gehlenborg, A. Lex, H. Pfister, and M. Streit. Domino: Extracting, comparing, and manipulating subsets across multiple tabular datasets. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2023–2032, 2014. doi: 10.1109/TVCG.2014.2346260

[22] J. Heer. Vega: A visualization grammar. https://vega.github.io/, 2013.

[23] J. Heer, S. K. Card, and J. A. Landay. Prefuse: A toolkit for interactive information visualization. In *Proc. CHI*, pp. 421–430, 2005. doi: 10.1145/1054972.1055031

[24] J. Heinrich and D. Weiskopf. State of the art of parallel coordinates. In *Proc. Eurographics - State of the Art Reports*, pp. 95–116, 2013. doi: 10.2312/conf/EG2013/stars/095-116

[25] N. Henry and J.-D. Fekete. NodeTrix: a hybrid visualization of social networks. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1302–1309, 2007. doi: 10.1109/TVCG.2007.70582

[26] A. Inselberg. *Parallel Coordinates*, pp. 2018–2024. Springer US, Boston, MA, 2009. doi: 10.1007/978-0-387-39940-9_262

[27] A. Inselberg and B. Dimsdale. Parallel coordinates: a tool for visualizing multidimensional geometry. In *Proc. IEEE Visualization*, pp. 361–378, 1990. doi: 10.1109/VISUAL.1990.146402

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

[29] B. Jelen and M. Alexander. *Pivot Table Data Crunching.* Que Corp., 2005.

[30] E. Jewett. Reductio: Crossfilter grouping. https://github.com/crossfilter/reductio, 2018.

[31] B. Johnson and B. Shneiderman. Tree-maps: a space-filling approach to the visualization of hierarchical information structures. In *Proc. IEEE InfoVis*, pp. 284–291, 1991. doi: 10.1109/VISUAL.1991.175815

[32] W. A. D. Kanchana, G. D. L. Madushanka, H. P. Maduranga, M. D. M. Udayanga, D. A. Meedeniya, and I. Perera. Semi-automated recommendation platform for data visualization: Roopana. In *Proc. Moratuwa Engineering Research Conference*, pp. 117–122, 2017. doi: 10.1109/MERCon.2017.7980467

[33] D. A. Keim. Information visualization and visual data mining. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):1–8, 2002. doi: 10.1109/2945.981847

[34] D. A. Keim, F. Mansmann, J. Schneidewind, and H. Ziegler. Challenges in visual data analysis. In *Proc. International Conference on Information Visualisation*, pp. 9–16, 2006. doi: 10.1109/IV.2006.31

[35] R. Layton. Your data deserve better than pies and bars: An r graphics workshop for the timid. In *Proc. IEEE Frontiers in Education*, pp. 1–3, 2014. doi: 10.1109/FIE.2014.7043983

[36] J. LeBlanc, M. O. Ward, and N. Wittels. Exploring n-dimensional databases. In *Proc. IEEE Visualization*, pp. 230–237, 1990. doi: 10.1109/VISUAL.1990.146386

[37] T. Mostak. Using gpus to accelerate data discovery and visual analytics. In *Future Technologies Conference (FTC)*, pp. 1310–1313, 2016. doi: 10.1109/FTC.2016.7821771

[38] T. Munzner and E. Maguire. *Visualization Analysis and Design*. CRC Press, 2015.

[39] K. L. Norman, L. J. Weldon, and B. Shneiderman. Cognitive layouts of windows and multiple screens for user interfaces. *International Journal of Man-Machine Studies*, 25(2):229–248, 1986. doi: 10.1016/S0020-7373(86)80077-3

[40] G. Parker, G. Franck, and C. Ware. Visualization of large nested graphs in 3D: Navigation and interaction. *Journal of Visual Languages and Computing*, 9(3):299–317, 1998. doi: 10.1006/jvlc.1998.0086

[41] S. F. Roth, P. Lucas, J. A. Senn, C. C. Gomberg, M. B. Burks, P. J. Stroffolino, A. J. Kolojechick, and C. Dunmire. Visage: a user interface environment for exploring information. In *Proc. IEEE InfoVis*, pp. 3–12, 1996.

[42] A. Rungta, B. Summa, D. Demir, P.-T. Bremer, and V. Pascucci. ManyVis: Multiple applications in an integrated visualization environment. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2878–2885, 2013. doi: 10.1109/TVCG.2013.174

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

[44] 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. doi: 10.1109/TVCG.2016.2599030

[45] 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. doi: 10.1109/TVCG.2015.2467091

[46] H.-J. Schulz and S. Hadlak. Preset-based generation and exploration of visualization designs. *Journal of Visual Languages And Computing*, 31(PA):9–29, 2015. doi: 10.1016/j.jvlc.2015.09.004

[47] B. Shneiderman. The eyes have it: a task by data type taxonomy for information visualizations. In *Proc. IEEE Symposium on Visual Languages*, pp. 336–343, 1996. doi: 10.1109/VL.1996.545307

[48] B. Shneiderman. Extreme visualization: Squeezing a billion records into a million pixels. In *In Proc. ACM SIGMOD*, pp. 3–12, 2008. doi: 10.1145/1376616.1376618

[49] S. S. Stevens. On the theory of scales of measurement. *Science*, 103(2684):677–680, 1946.

[50] C. Stolte, D. Tang, and P. Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Transactions*

on Visualization and Computer Graphics, 8(1):52–65, 2002. doi: 10.1109/2945.981851

[51] C. Tominski, J. Abello, and H. Schumann. Axes-based visualizations with radial layouts. In *Proc. of the ACM Symposium on Applied Computing*, pp. 1242–1247, 2004. doi: 10.1145/967900.968153

[52] M. Vartak, S. Huang, T. Siddiqui, S. Madden, and A. Parameswaran. Towards visualization recommendation systems. *SIGMOD Record*, 45(4):34–39, May 2017. doi: 10.1145/3092931.3092937

[53] W. Wang, H. Wang, G. Dai, and H. Wang. Visualization of large hierarchical data by circle packing. In *Proc. CHI*, pp. 517–520, 2006. doi: 10.1145/1124772.1124851

[54] H. Wickham and H. Hofmann. Product plots. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2223–2230, 2011. doi: 10.1109/TVCG.2011.227

[55] J. J. V. Wijk and H. V. de Wetering. Cushion treemaps: visualization of hierarchical information. In *Proc. IEEE InfoVis*, pp. 73–78, 147, 1999. doi: 10.1109/INFVIS.1999.801860

[56] L. Wilkinson. *The Grammar of Graphics (Statistics and Computing)*. Springer-Verlag New York, Inc., 2005. doi: 10.1002/wics.118

[57] K. Wongsuphasawat, D. Moritz, A. Anand, J. Mackinlay, B. Howe, and J. Heer. Towards a general-purpose query language for visualization recommendation. In *Proc. Workshop on Human-In-the-Loop Data Analytics*, pp. 4:1–4:6, 2016. doi: 10.1145/2939502.2939506

[58] K. Wongsuphasawat, D. Moritz, A. Anand, J. D. Mackinlay, B. Howe, and J. Heer. Voyager: Exploratory analysis via faceted browsing of visualization recommendations. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):649–658, 2016. doi: 10.1109/TVCG.2015.2467191

[59] K. Wongsuphasawat, Z. Qu, D. Moritz, R. Chang, F. Ouk, A. Anand, J. Mackinlay, B. Howe, and J. Heer. Voyager 2: Augmenting visual analysis with partial

view specifications. In *Proc. CHI*, pp. 2648–2659, 2017. doi: 10.1145/3025453.3025768

[60] J. S. Yi, Y. a. Kang, and J. Stasko. Toward a deeper understanding of the role of interaction in information visualization. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1224–1231, 2007. doi: 10.1109/TVCG.2007.70515

[61] C. Ziemkiewicz and R. Kosara. Beyond Bertin: Seeing the forest despite the trees. *IEEE Computer Graphics and Applications*, 30(5):7–11, 2010. doi: 10.1109/MCG.2010.83