Direct Manipulation Techniques for Creation of Multiple-View Visualizations

by

Katharine E. Bacher

S.B. in Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 2020

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2021

Author ....
Department of Electrical Engineering and Computer Science
May 20, 2021

Certified by..
Arvind Satyanarayan
Assistant Professor
Thesis Supervisor

Accepted by ....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Direct Manipulation Techniques for Creation of Multiple-View Visualizations

by

Katharine E. Bacher

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2021, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

I propose an extension to Lyra, a tool for authoring interactive data visualizations, that introduces novel processes for creation and laying out of multiple-view (MV) visualizations. The existing tools for designing data visualizations lack specific support for MVs, making the process of creating such visuals cumbersome for users. I draw on the concepts of direct manipulation, in which users interact with objects in the visual to enact desired changes, to develop new methods of creating MVs that are more intuitive to users and have less of a learning curve. With my proposed changes, users can iteratively add new groups in Lyra simply by dragging and dropping visual marks onto dropzones. To facilitate this new method for adding groups, I implement a layout concept within Lyra to allow for easier creation of gridded layouts for MVs. Additionally, this concept is extended to data-driven layouts that can be automatically generated for small multiples, or trellis charts. The system is evaluated by demonstrating the new mechanisms and showing an example gallery of MVs that can be created with these methods.

Thesis Supervisor: Arvind Satyanarayan
Title: Assistant Professor

Acknowledgments

I would like to thank Professor Arvind Satyanarayan and Jonathan Zong for their mentorship, direction, and support throughout the research process, and Ebenezer Sefah for the camaraderie and collaboration on our MEng projects.

I am grateful to the entire MIT Visualization Group for being such an inspiring, insightful, and thoughtful group of people and for making me feel welcome in the community, in spite of everything being virtual for the past year.

Contents

List of Figures

Chapter 1

Introduction

As people visualize more complex datasets, a single-view graph often does not afford enough expressivity to capture the data. However, multiple-view visualizations — those made up of several charts — allow for more complex representations of data and can showcase different aspects of high-dimensional data in a single graphic. Examples of some MV visualizations can be found in Figure 1-1.

Despite widespread acceptance of MVs in the visualization community, there has been little work, to date, on developing methods that facilitate creating them. Current data visualization tools instead focus primarily on designing and creating single-view charts. As a result, the process for creating MVs using current tools is less user-friendly, and often, inconsistent with direct manipulation principles. While visualization applications like Lyra, Charticulator, and Data Illustrator make it possible to create MVs, they lack specific features to help create such visuals. Previously in Lyra, adding charts required button clicks in inspector menus, and there was no way to create a structured layout. Users instead had to manually drag and position charts. Thus, in this thesis, I extend the existing Lyra program with new features targeting the layout and creation of multi-view visualizations.

I describe a new layout concept to define the arrangement, positioning, and sizing of groups within a MV visualization. While there is an exponentially increasing number of possible layouts as the number of groups in a visualization increases, I chose to focus my attention on gridded layouts with a set base unit that determines a default

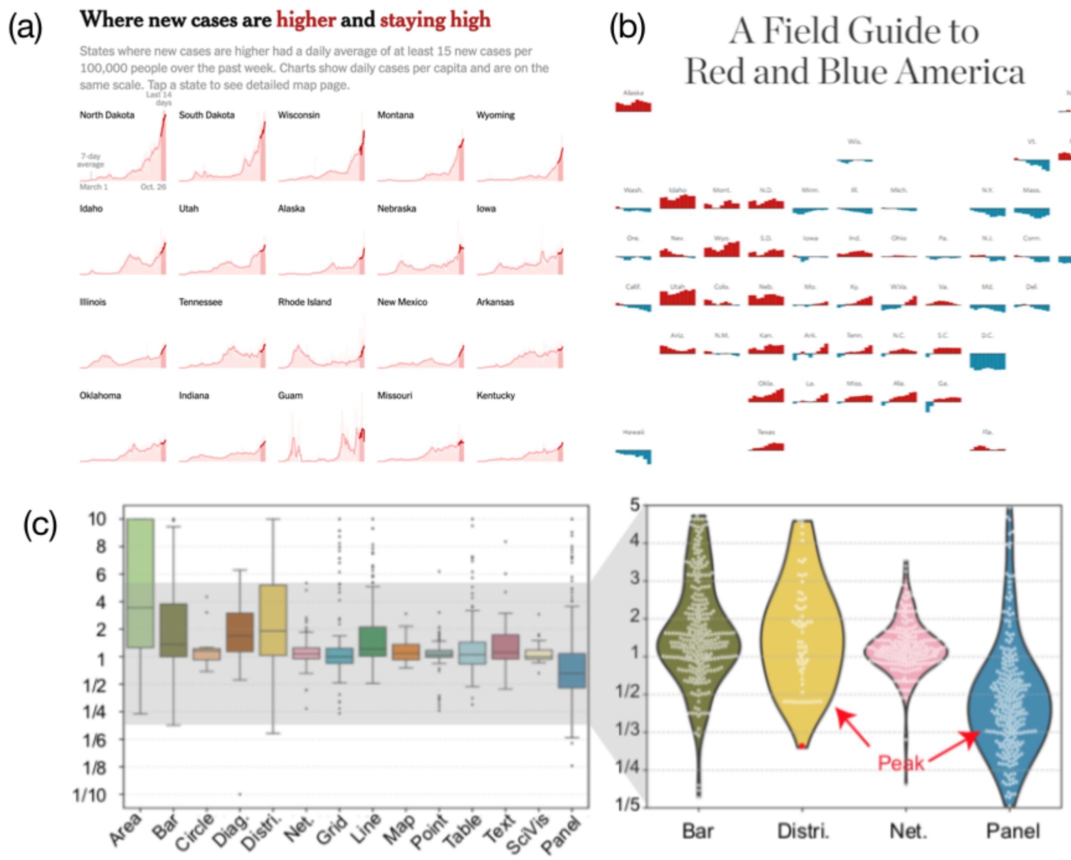


Figure 1-1: Examples of MV visualizations: (a)Small multiples from the NYT showing state-by-state COVID cases [14], (b) a MV with state-by-state election data positioned by geography [15], and (c) a MV with two different types of charts looking at the same data [2].

row and column size. This decision reduces the scope for the layout implementation but still provides for a broad diversity of possible layouts. With the new gridded layout system, users can drag-and-drop graphical marks directly onto positional dropzones to create new groups in the desired arrangement within the current layout. Grid-specific resize features allow for dragging a vertical or horizontal line between columns or rows, respectively, to resize an entire row or column of the layout at once and automatically adjust the size and positioning of affected charts accordingly. These concepts lay the foundation for creating groups and laying them out in a structured way.

Under the umbrella of multiple-view visualizations, small multiples are a particular subset of MVs that contain several similar charts with the same axes and scales but that are faceted, or binned, by a particular field of the data. Since small multiples contain numerous similar charts, the process to create such visualizations is very repetitive, requiring users to manually set up several groups with the same axes and scales and then filter the data separately for each one. Instead, I propose a method to automate this process once a user builds the base chart and chooses a field to facet out the data. The faceted groups are then automatically laid out in the chosen arrangement.

To help users understand how to use the new features, the methods to add charts and create small multiples were developed largely in-line with the direct manipulation techniques already existing in Lyra. These tasks are done by dragging and dropping marks and fields onto designated dropzones, processes users of Lyra would already be familiar with but that were not previously applied to MV creation and layout. Thus, in this thesis, I propose extending previous direct manipulation techniques for creating individual visualizations in Lrya to now also handle creation of multiple view visualizations and data-driven layouts.

Chapter 2

Related Work

The main principle behind my proposed contribution is direct manipulation. Direct manipulation is an approach to interaction where users demonstrate and interact with objects directly to enact their desired edits and changes. As opposed to alternative approaches that rely heavily on button clicks and editor menus, direct manipulation proposes to be easier to learn and a more intuitive method of user interaction. Hutchins et al. propose that direct manipulation interfaces are only successful when a feeling of *directness* is established [3]. One critical aspect of *directness* is minimizing *distance*, which they describe as the gulf between the user's intentions and the way those goals must be specified to the system. Their concept of *articulatory distance*, which defines how closely the meaning of intention maps to physical form, is particularly relevant to my work as I move away from buttons in inspector menus (which have a large *articulatory distance* as they provide no physical alignment with the users' goals) to implementing actions that more closely mimic the user's intention. More recently, there has been increased research on applying direct manipulation to graphical encodings for visualization construction. Most research in this area focuses on how users can manipulate the size, position, or color of encodings or how they apply data to marks [6]. There has been limited research, however, applying direct manipulation to the creation or layout of multiple-view visualizations.

Numerous tools for authoring data visualizations employ direct manipulation principles, including Lyra [8], Data Illustrator [4], and Charticulator [5]. In Lyra, users drag-and-drop data fields to bind them to graphical marks, and users can directly resize and move marks using handles. A recent addition to the Lyra application allows for authoring interactive visualizations by directly demonstrating the desired interaction on the visual [16]. In a different approach, Adobe's Data Illustrator treats data visualizations first and foremost as vector graphics, which offers a different paradigm for interaction and layout mechanisms. The interaction mechanisms supported in Data Illustrator — using tools for drawing, selecting, and manipulating vector graphics — are more familiar to designers' existing workflows [4]. Charticulator from Microsoft, like Lyra, uses a drag-and-drop mechanism for binding data but adds the concept of mathematical layout constraints, allowing more freedom and flexibility in determining layout composition [5]. As explored by the creators of these three applications in their *Critical Reflections*, none of these tools have a particularly smooth process for handling MVs, and Lyra, in particular, is noted as having a "highly viscous" user experience for dealing with nested layouts, a process necessary for creating small multiples [7]. In the current Lyra implementation, groups are added with the click of a button and are created with a default size and position. Users can then drag the group to position it and click and drag resize handles to change the dimensions. To create a small multiples plot, users would have to manually filter the data by properties of a particular field and then create a new group for each of the filtered properties. While creating MVs is possible in all three of these applications, there are few to no specific features to aid in the process of creating such visuals.

Recent work analyzing the design space of MVs served as insight for my thesis. Xi Chen et al. provided an overview of MV composition, which defines the types and number of charts that comprise MV visualizations, and configuration, which determines the spatial layout of the constituent charts [2]. They found that most charts only have a few visuals (fewer than 5), although there were a sizable portion of visuals with 10+ charts, which indicated we would ideally like to support larger compositions. They also developed the concept of a hierarchy for visual layout construction to handle small multiple compositions, in which views and small multiples make up the first level of a gridded layout. The subplots of the small multiples then form the second level in the hierarchy. The MV visualizations they analyzed did not go deeper than 2 levels. Working off this hierarchical concept of layout, the groups comprising a small multiples plot together could be treated as a single visualization in a cell of a MV layout. While this work only analyzed visualizations included in academic papers submitted to various visualization conferences (e.g. IEEE VIS, EuroVIS), their findings are still relevant to several of the design decisions that went into my work. Additional analysis, however, could be done to see whether these results hold across other visualization platforms, such as news and government publications.

Chapter 3

Background

Before diving into the new additions to Lyra, it is important to understand the Lyra and Vega terminology. Lyra is built on top of Vega [9], a visualization grammar for defining graphics. Consequently, all visualizations created in Lyra are represented as Vega specifications [8]. The main building block of multi-view visualizations in the Lyra interface are *groups* [11]. In Vega, groups act as containers for other marks and can include data references, axes, legends, scales, marks, and properties such as position and size for a chart. In Lyra, groups are managed in the main inspector sidebar on the left side of the interface. Groups are currently the highest level mark in Lyra, but my implementation adds the concept of a layout, which contains one or more groups.

Only a subset of the Vega language is exposed through the Lyra interface, and one aspect of Vega currently not supported is the *layout* specification, which enables a simple way of positioning multiple groups in a visualization [12]. In the Vega layout specification, users dictate the number of columns and their alignment. As seen in Figure 3-1, by changing the number of *columns* in the layout specification, users can achieve four different layouts with four groups. Additionally, the *align* property in the layout definition can be used to specify the spacing for the rows and columns, as seen in Figure 3-2. While not providing a significant amount of flexibility to users, the layout specification does allow for automatic laying out of groups at runtime, which turned out to be beneficial for faceted groups. In this thesis, I developed a custom

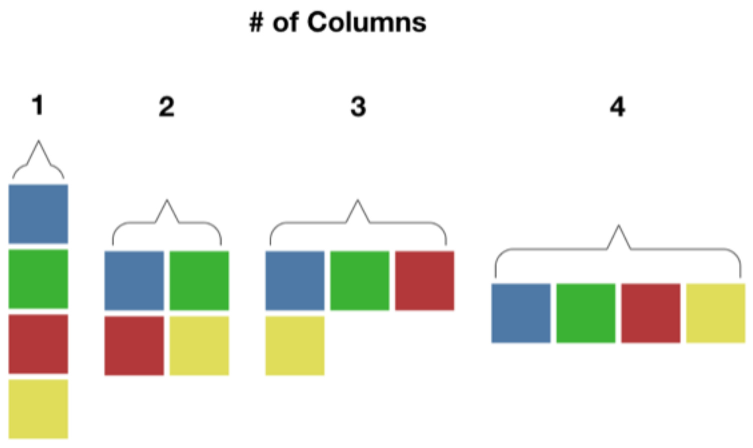


Figure 3-1: Each colored box represents a group, which together makes up a four-view visual. Changing the number of columns in the Vega layout specification creates four different layouts.

concept of a gridded layout that does not utilize the Vega layout specification for the general case to provide more expressiveness and give more autonomy to users.

In Vega, the data source behind groups can be specified using the *from* property, which can either take a dataset reference name for the entire dataset or a *facet* definition, which allows for partitioning the data across groups [10]. A data-driven facet has a groupby property, which accepts a data field by which to partition the dataset. The number of partitions, and thus number of resultant groups, are determined at runtime. Defining a facet is the basis of my small multiples implementation.

Another key concept in Vega and Lyra is *signals*, which are reactive variables that can update based on input events or changes to upstream signals [13]. Signals in Vega help to create interactive visualizations that can update on events, such as clicks or mouseovers. Lyra makes extensive use of signals, beyond interactive visualizations. In the Lyra interface, almost every supported property that can be defined in a Vega specification is enumerated in a signal. This enhances performance so that when a chart property is changed in an inspector or by direct manipulation, the signal reacts to the change and the visualization that appears in the Lyra canvas updates automatically. When exporting the Vega specification for a visual created in Lyra, however, most of these properties do not need to be reactive in the resultant chart, and

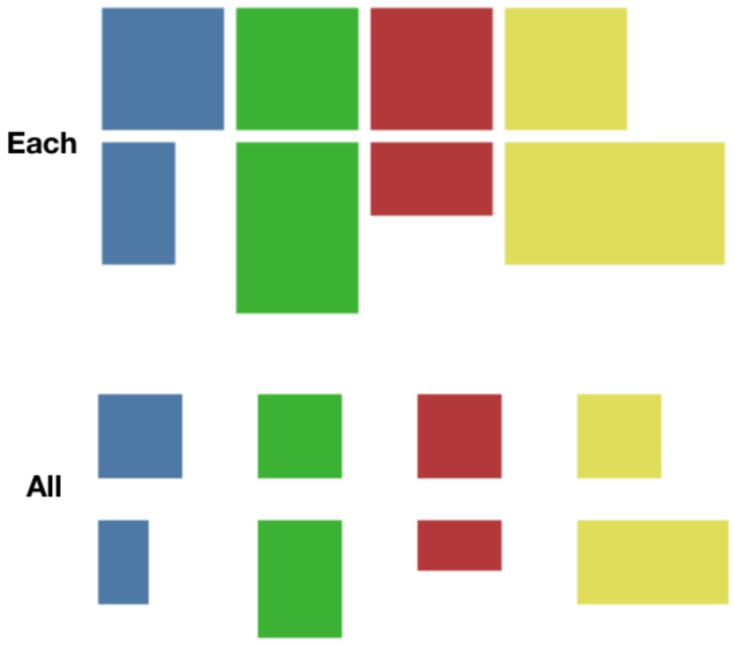


Figure 3-2: Setting the align value in the Vega layout specification for these eight differently sized groups to “each” and “all” creates for two separate layouts, one with variable column and row width and height and the other with identical row and column width and height, respectively.

thus, are converted to their value at the time of export. My layout implementation takes advantage of the *update* property of the signal definition to create a network of signals that update when an upstream signal changes. The *update* property takes an expression, such as another signal name or an expression containing multiple signals. Then when the value of signals referenced in the update expression change, the signal value updates as well. Similar to most other signals in Lyra, these are converted to their value and do not appear in the resultant Vega specification.

Chapter 4

Methods

4.1 Design Process

I started by performing a literature review of the prior work on visualization systems, direct manipulation, and multiple-view visualizations. After getting a better understanding of the current limitations with MV creation, I developed some high-level design goals to inform my research. Next, I began an extensive prototyping phase, in which I iterated on different approaches and interaction mechanisms to meet those objectives. Each prototype was evaluated on how successfully it met the design goals, and the most successful concepts from the prototyping phase were carried over into the final implementation in Lyra.

4.2 Design Goals

Informed by a literature review and initial prototyping, I developed the following design goals for my proposed system:

- **Expressive.** The process for creating MVs should be generalizable to accommodate a wide breadth of possible layouts. Chen et al. document the design space of MV composition and configuration, observing a non-trivial number of compositions with 10+ views, a broad range of gridded layouts, and multiple chart types within a single visualization [2]. The system should be expressive enough to handle the common layouts and designs they document. From my prototypes, I found ones that were less expressive felt lacking and restrictive.

- **Easy to use and learn.** Designing interactions for creating MVs that more closely mimic the users' intentions reduces *articulatory distance* and helps to create the feeling of directness [3]. Hutchins et al. assert this feeling is necessary to achieve the benefits of direct manipulation, which proposes to make processes more intuitive and to reduce the cognitive load on users.

- **Visibility when needed.** Components to aid in MV design should be visible when needed and hidden when not. Users designing a single-view chart need not be bombarded with interface elements specific to MVs. Conversely, hiding buttons in inspector panels decreases visibility and makes the processes for creating MV more viscous [1]. Towards the end of my prototyping phase and after developing several new features, I found it necessary to only present components when users could reasonably use them to avoid detracting from the visualization itself or cluttering the view.

- **Consistent.** The new processes should be in-line with existing methodologies. Users that already have experience building single-view visualizations in Lyra or similar visualization tools likely already have preconceptions about affordances of the platform or visualization construction by direct manipulation, in general. Existing methods of interactions in direct manipulation visualization authoring tools include techniques like drag-and-dropping data fields to bind data and dragging handles to resize [10]. If similar concepts for MV construction are presented in different forms, their purpose is obscured, and usability is compromised [12]. Developing new processes that are similar to existing paradigms makes those methods more learnable and consistent with users' expectations.

4.3 Prototypes

With these design goals in mind, I started by prototyping different interactions and methods for laying out multiple views and easily creating new groups. The goal at this stage of the process was to make several low-fidelity prototypes via design sketches and simple code that allowed for quick iteration and evaluation before moving into the actual Lyra codebase. Each prototype presented different trade-offs between ease of use, breadth of possible creations, and how much flexibility was afforded to users. These trade-offs were evaluated before deciding on the final implementation plan.

4.3.1 Layout

The first couple prototypes tackled different ways of creating layouts for multiple groups. I compared snap-to-grid functionality for freeform placement of groups to rigid, gridded layouts that restricted movement.

Snap-to-grid

The first prototype implemented snap-to-grid functionality. In this prototype, users could freely drag groups around the screen, and the groups would "snap" into place if dragged near an alignment line with another group. In this case, a dotted line would appear to show that a side of the currently dragged group lines up with a side of another chart on the screen (e.g. they have the same x or y position, respectively). See Figure 4-1 for an example.

The primary advantage with this method of laying out groups is expressiveness. A user can make any arbitrary layout desired and have complete control over the size and position of each group. Additionally, users can already freely drag groups around in Lyra, so implementing snap-to-grid on top of the dragging functionally would likely be intuitive and familiar to users. This expressiveness and consistency, however, come at the expense of requiring more manual work on the part of the user. For example, if a user wants to create a gridded layout, they would first have to resize all the charts and then align them one by one. If they then want to increase the size

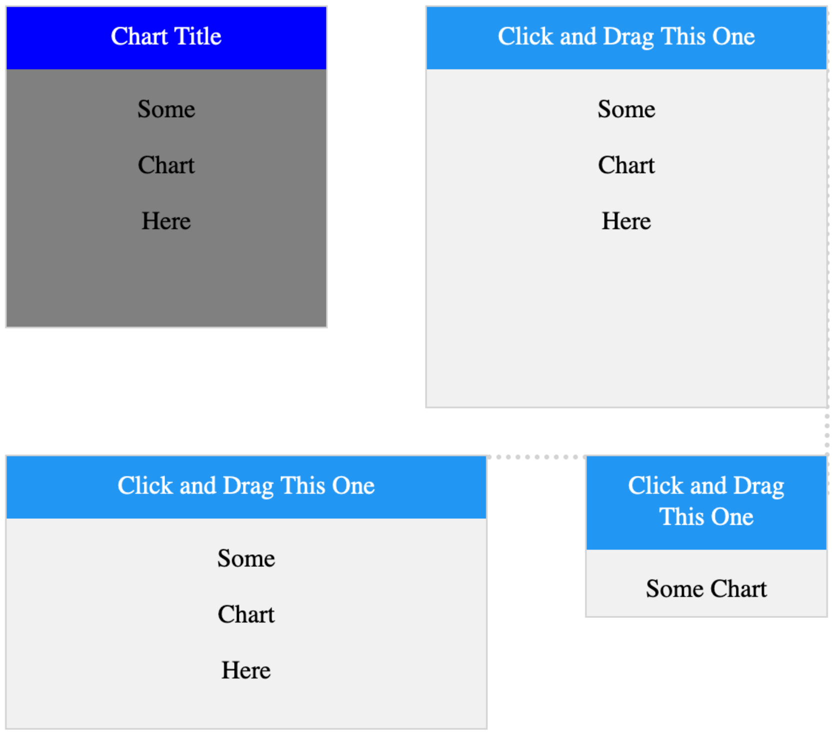


Figure 4-1: An example layout with four charts in the snap-to-grid prototype. The bottom right chart is currently being dragged and the user can see that the top and right sides align with other charts in the layout.

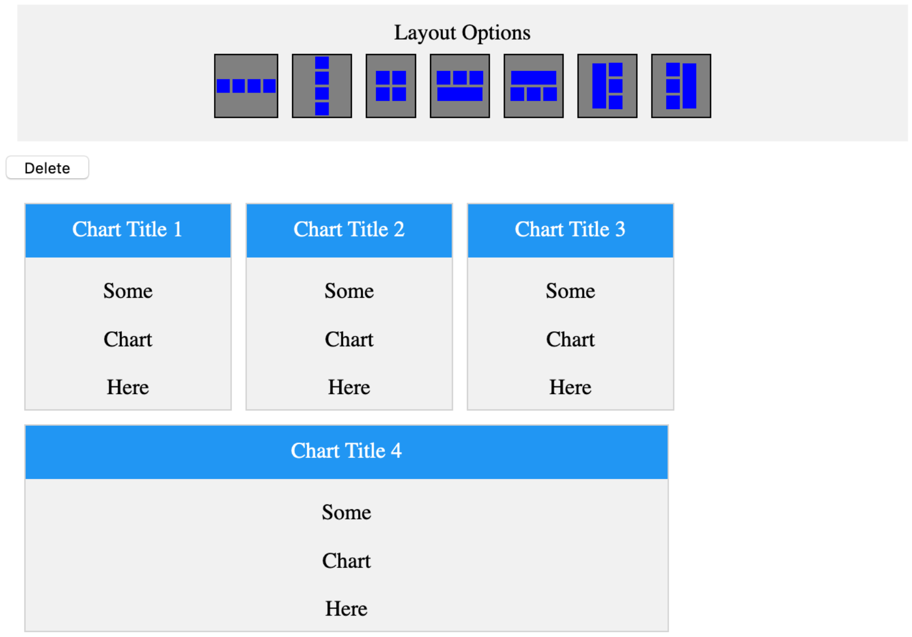


Figure 4-2: Gridded layout with 4 groups and 7 layout options.

of the first column, for example, they would have to manually increase the width of all the charts in the first column and then adjust the position of all the other charts. For these reasons, I next prototyped a gridded layout method.

Predefined Layout Options

The next batch of prototypes centered around creating gridded layouts. I iterated on different methods of adding groups to layouts and how users could choose a desired layout. In one prototype of the gridded layout, users relied on predefined layout options to pick a desired arrangement and could click buttons to apply different layouts to their visualization. Here, the buttons would show pre-specified arrangements of groups with the same number of groups as currently existed in the visualization. For example, if the user had a visualization with 4 groups, one button would show the groups laid out in a row, one in a column, another as a 2-by-2 grid, and so on (see Figure 4-2).

The advantage of this method is that it allowed users to quickly try out different layouts based on the number of charts they created. However, an “Options Panel” and

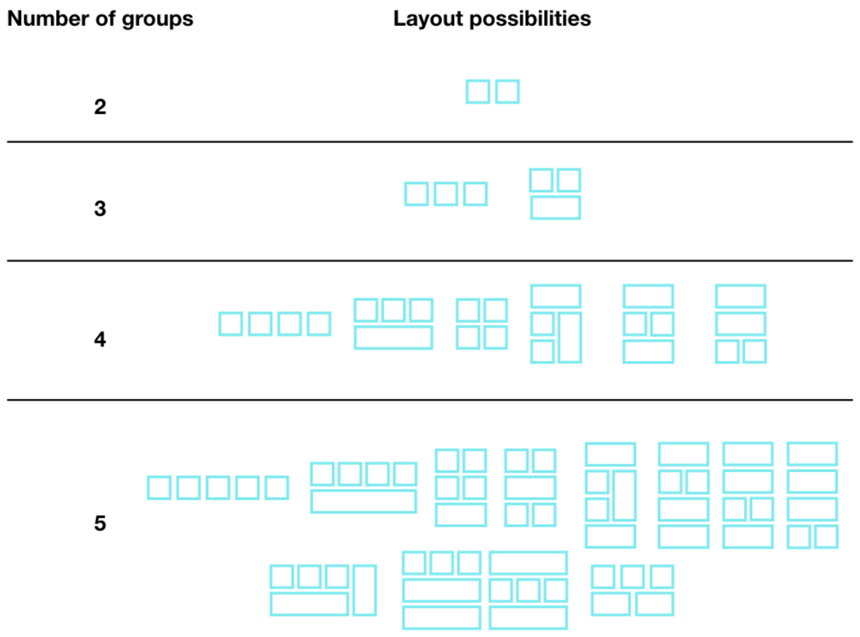


Figure 4-3: Layout possibilities for given number of groups (does not include rotation or flips of these possibilities).

requiring button clicks is not in-line with direct manipulation principles. Specifically, the option panel increases the *articulatory distance* as there is a representational disconnect between the user's goal (changing the layout) and the physical form of the input (pressing a button) [3]. This disconnect makes the interaction less intuitive for users. Furthermore, the number of possible options explodes exponentially with the number of groups, making it not feasible to enumerate the full options space for large layouts (see Figure 4-3) and resulting in a loss of expressiveness.

To solve some of these issues, the next iteration of this prototype placed a subset of predefined layout options around the visualization as dropzones, drawing on the principles of direct manipulation for both layout and adding new charts. Unlike the previous iteration, the layout options would not be for the current number of groups, but for the number of groups plus one additional one that would get added on drop. The layout options also were restricted to only the layouts that could be created by adding one group to the current layout, without repositioning the current groups. Thus, users were presented with only 4 layout options, which corresponded to the

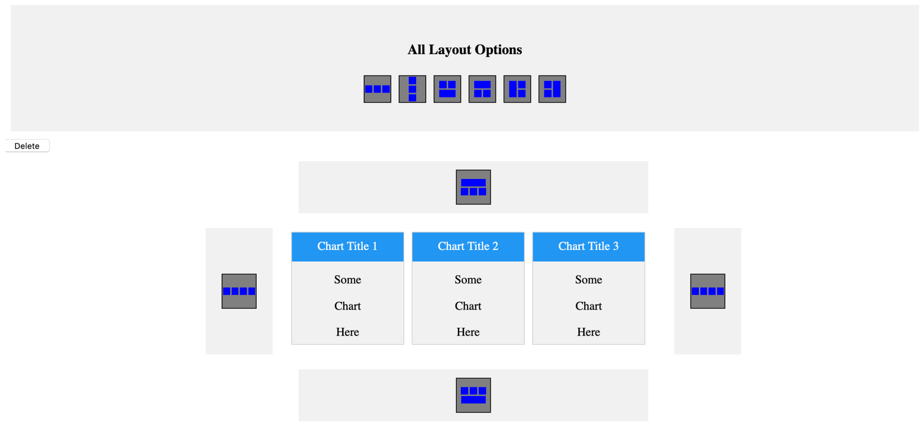


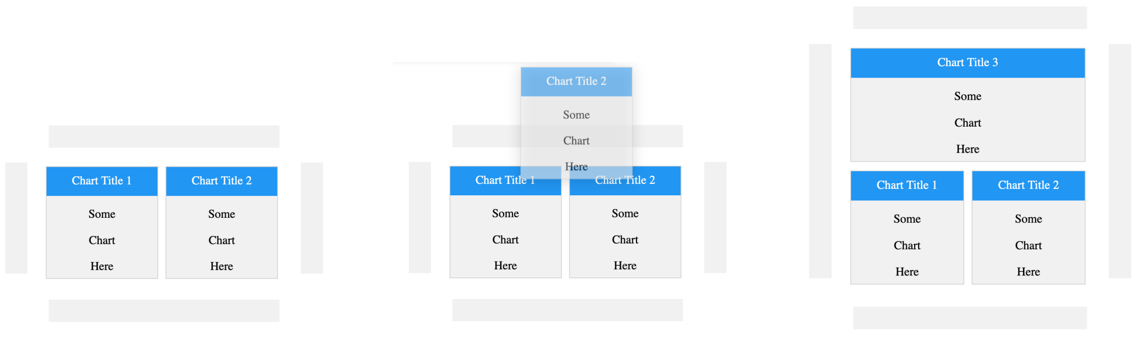
Figure 4-4: Reduced-options dropzone prototype.

possible layouts that could be created by adding a group to each side, respectively. See Figure 4-4. The system kept track of the current layout and had a mapping for the next reachable layouts from that state, which were presented to users in the dropzones. Alternative options for the current layout could still exist as buttons in an inspector panel, like the previous iteration, if users desired, but would not be required to build layouts.

While adding dropzones did help solve some of the concerns regarding direct manipulation, several issues remained. The largest issue with this implementation is that developers would still have to precompute all the layout options and create the mapping from each layout option to the next valid states. This presents a huge overhead, and ultimately, users would be limited by how many layouts were implemented and would likely be restricted to a relatively small number of groups allowed in layouts. Thus, this method of creating gridded layouts from precomputed options was rejected.

On-the-fly Layout creation

In the next iterations of prototypes, I moved to “on-the-fly” gridded layout creation. Instead of relying on precomputed layouts, users could iteratively build up a gridded layout by adding groups to valid grid positions. Here, generic dropzones on each size of the visualization allow users to add a group to any side to build up a layout. Now



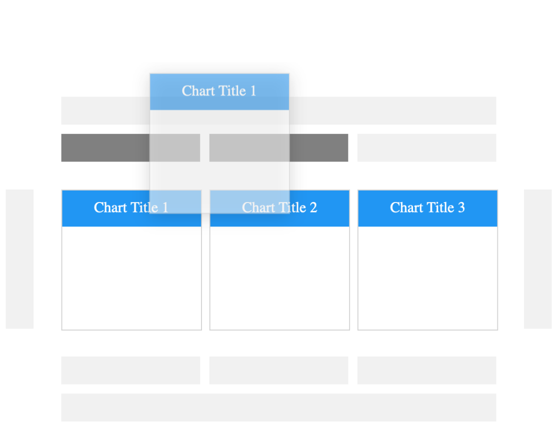
(a) (b) (c)

Figure 4-5: Process for on-the-fly layout creation. (a) Starting from a row of two views, (b) a user drags onto the top dropzone, which (c) results in a new group added above the original two.

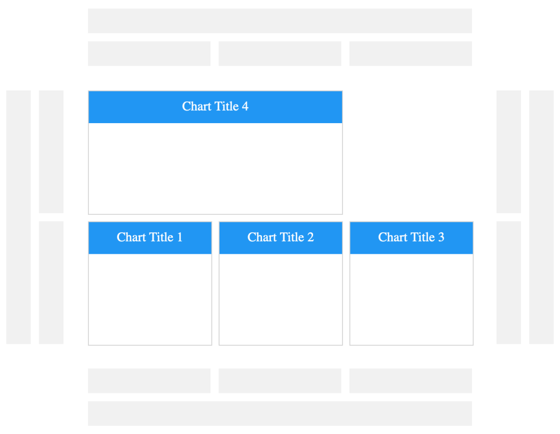
instead of precomputing chart sizes and positions for a given layout, these values would be computed in real time once a user chooses where to add a group. If the top dropzone is selected, for example, a new group is added above the existing charts in the layout with unit height and width equal to the current width of the visualization. See Figure 4-5.

This method could create almost all the same layouts as with the predefined buttons but was not limited to the options implemented by a developer. As I continued to develop this prototype, I included split dropzones for users to create different sized new groups, instead of the new group filling the full width or height of the side to which it was added. Instead of a single dropzone on each side, there were separate dropzones for each row and column. Now, users could drag over multiple dropzones along one side and the resultant group would be the cumulative size of those dropzones. For example, in Figure 4-6 a user drags over the first two segmented dropzones on the top to select them and the result is a new group that has a width of two units.

While enabling greater expressiveness in possible layouts, this change also created the need for placeholder dropzones for regions where new groups did not take up a full width or height of the row or column, respectively. These new dropzones appear as unit sized whitespace in the layout where a new group could be placed so as to not disrupt user's view of their created layout. When a group is dragged over a placeholder, however, the placeholder dropzone turns dark grey to indicate that it is



(a)



(b)

Figure 4-6: User drags over multiple split dropzones to create a new group 2 units wide.

selected. Like with the split dropzones, multiple in a row can be chained together to create a group the cumulative size of all the selected placeholders. See Figure 4-7.

After iterating on these layout concepts, I compared the snap-to-grid, predefined options, and on-the-fly methods in terms of usability and expressiveness and determined the "on-the-fly" gridded layout would be preferable for the final implementation. While lacking some of the flexibility of the snap-to-grid method, this implementation provided easy methods for users to quickly add groups and create structured layouts. I judged that the gridded formation would be sufficiently expressive enough to cover most use cases for MV visualizations, and if users wanted to create more complex layouts that did not fall into a grid, they could still follow the original manual method that already existed within Lyra. The on-the-fly method proved to be more effective than the predefined layouts in that users could create arbitrarily large layouts, and there was less development overhead required for precomputing desired layout options.

4.3.2 Resize

Compared to the freeform snap-to-grid method, the gridded layout concept I developed is more rigid. Built on a default unit size for groups and uniform rows and columns, this implementation makes group sizing more constrained. To remedy this,

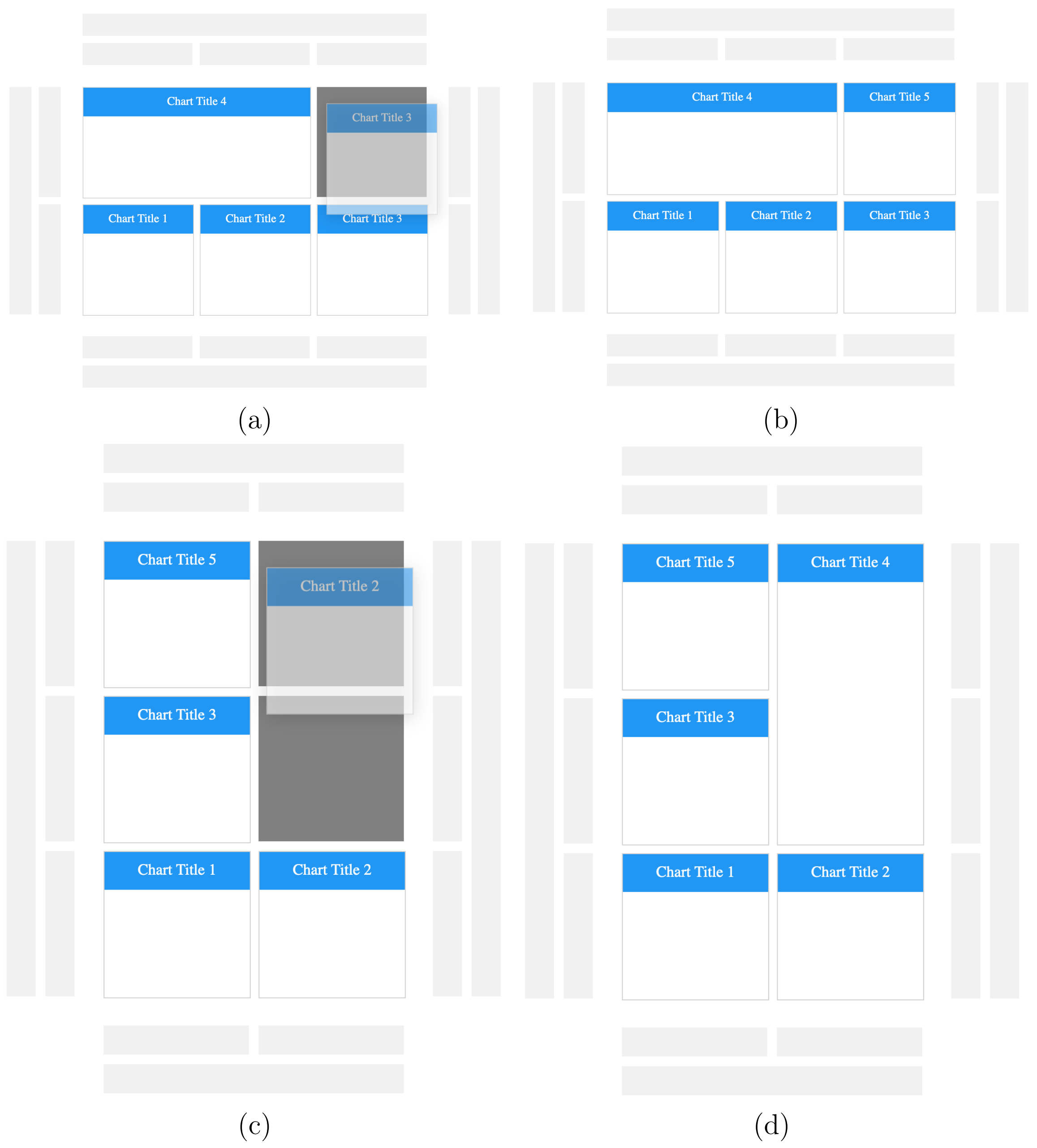


Figure 4-7: Placeholder dropzones: (a and b) show the result of dropping on a single placeholder dropzone, while (c and d) show the result from multiple selected placeholders.

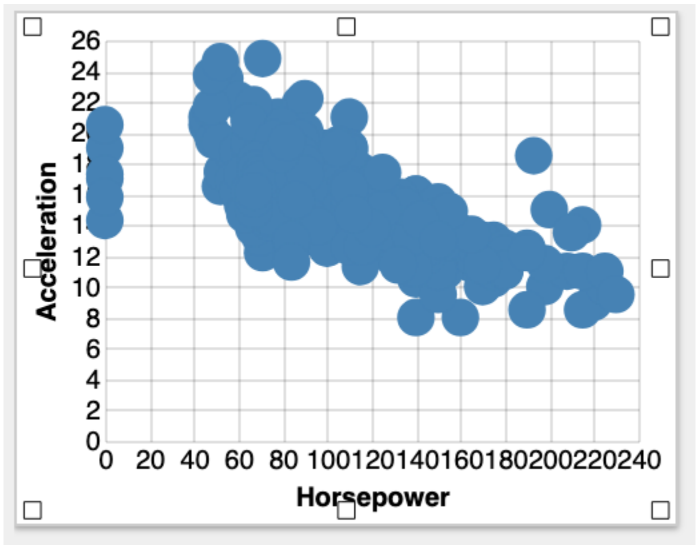


Figure 4-8: Lyra handles for continuous resize of individual groups.

the next prototypes centered around how to implement resizing across different scales to provide more flexibility on this front. To this end, I developed a hierarchy of resizing at different scales (single chart, whole row/column, and entire layout) and with different levels of precision (discrete vs. continuous).

Single Chart Resize

At the lowest level of the chart resize hierarchy is single chart continuous resize, which is what already exists within Lyra. Users can click and drag corner handles on a group to change the width and height to any desired value (See Figure 4-8). This method, however, is not compatible with discrete, gridded cells in a layout, and thus I wanted to transition to other methods of resizing.

First, I prototyped a discrete resize for single charts. This allows a chart to increase or decrease in size only by the width or height of an entire column or row, respectively. In my prototype, handles on each side of a group will appear when a group is selected, and a user can click and drag the handle in the desired direction of change. Dragging outward will increase the size of the group and bump any other charts in that direction into the next cells, while dragging inward has the opposite effect (note that a chart cannot be shrunk any further when it spans only a single

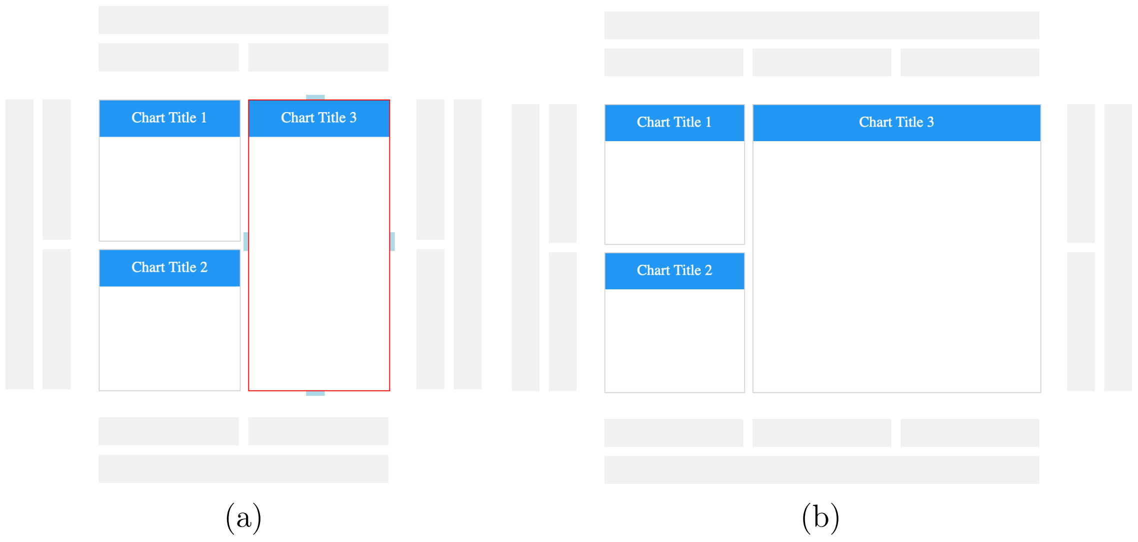
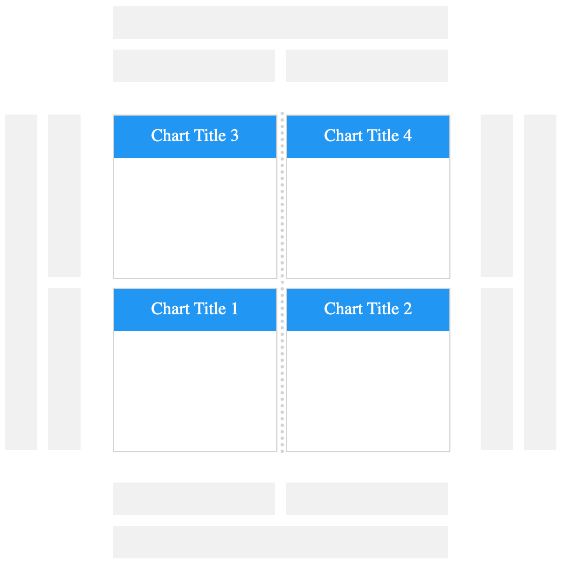


Figure 4-9: Discrete resize of individual group. A user drags the right-most handle on chart 3 to the right to enlarge it by the width of a new column.

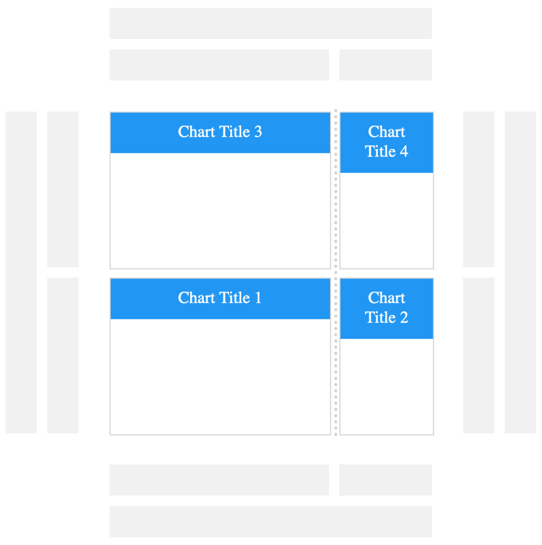
row and column). See Figure 4-9. Since this process is discrete, there is a jump when the user crosses a threshold and the chart resizes. To avoid bouncing back and forth between the two states around the threshold, once the resize occurs, the process stops and the user must re-grab the handle in order to perform another resize. While this extra step adds friction to resizing individual charts, this is much easier for users to control and is in-line with the discrete nature of the operation.

Row and Column Resize

The next level up in the layout resize hierarchy is continuous resizing of a whole row or column. This resize method leverages the gridding scheme and allows users to resize one dimension of all the charts in a particular row or column at once. In my prototype of this resize method, I created resize lines between all the adjacent rows and columns in the layout that became visible on hover. A user could click and drag a vertical line to adjust the adjacent column widths and similarly for a horizontal line to change adjacent row heights (Figure 4-10). Unlike the single charts, there are no constraints on individual row or column sizes and thus a user can freely move the lines to any desired size. For this implementation, I chose to keep the total size of the layout constant and thus increasing the size of one column would decrease the size of



(a)



(b)

Figure 4-10: User drags dotted line to resize columns.

the adjacent one. An alternative option would be for the size of all other columns to remain constant and let the overall layout size expand and contract, but since resizing of the entire layout is a different operation, I chose to keep this operation just to row and column sizes.

Whole Layout Resize

Finally, the top level is a continuous resize of the entire layout. Here, again, the overall size of the layout is not constrained, so users can adjust to any desired size. In my prototype, I implemented handles, similar to what already exists for individual groups in Lyra, that can be dragged to the desired size. Unlike for individual groups, resize changes to the whole layout must be distributed among constituent groups so that they all change proportionally the same amount. See Figure 4-11.

With these resize concepts – individual discrete, row/column continuous, and whole layout continuous – users could create a much wider breadth of gridded layouts. Now, users would not be constrained to groups of only one unit (a single row and column) in size or equal sized columns and rows. Additionally, resizing of the entire layout also allows users to change the overall size and the aspect ratio of all the charts

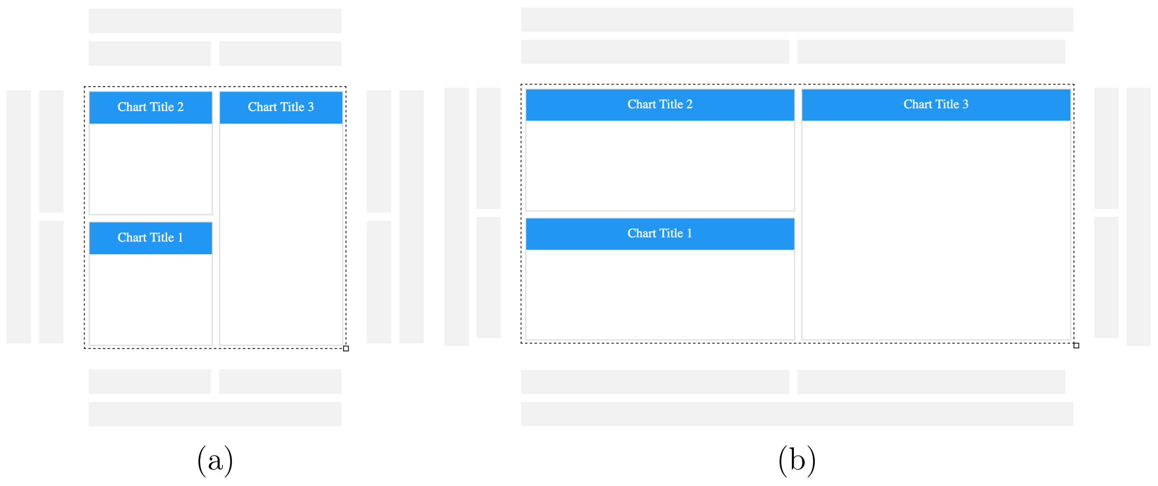


Figure 4-11: User drags handle out to the right to increase the width of the entire layout.

(e.g. make them longer or taller). A gallery of some example layouts that could be created with this final prototype can be seen in Figure 4-12.

4.3.3 Faceting

One of the motivating ideas for my thesis work is easier creation of small-multiple visualizations. Small-multiple plots are created by faceting a dataset by a particular field that is either ordinal or binned and then creating multiple charts with the same scales and axes that each include a different subset of the faceted dataset. I considered several methods of creating such plots, including more manual methods that leveraged my layout creation process and automated methods leveraging Vega layout specification. Due to the data-driven nature of this concept, I could not prototype with the basic web development tools (vanilla JS, HTML, and CSS) that I had for the previous prototypes and instead relied on interaction sketches.

The first method I imagined for faceting leveraged my dropzones and layout creation mechanism. With this approach, I wanted to focus on what a direct manipulation approach to facetting would look like. With this in mind, I considered being able to directly pull sets of data from an existing group to break the group into multiple charts with different segments of the data. Consider an area chart with 3 different

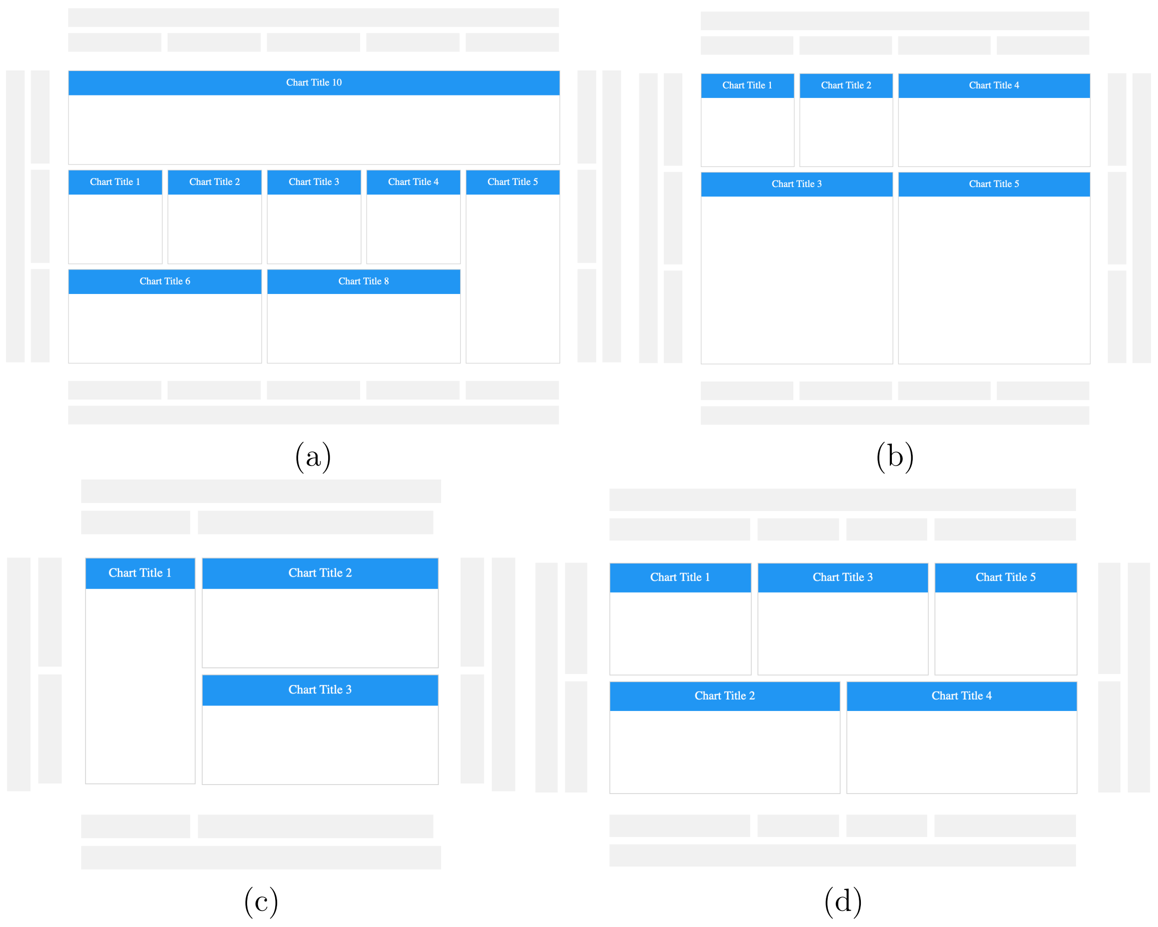


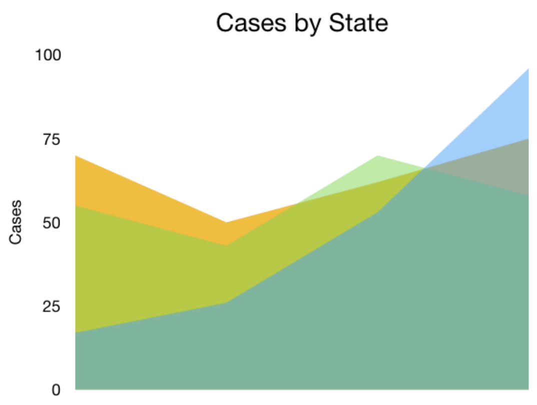
Figure 4-12: Example layouts that can be created with my layout prototype and resize capabilities: (a) large layout with 10 groups, (b) individual discrete resize, (c) column resize, and (d) 5 groups staggered.

series, colored blue, orange, and green, for example (see Figure 4-13). When a user clicks and drags on a particular color series, say the blue one, the dropzones for layout creation would appear and allow the user to drag and drop the data series. Like with the layout creation, a new group would be created next to the original chart on the side where the series was dropped. Then the new chart would be populated with only the dragged data, and this data would be removed from the original chart.

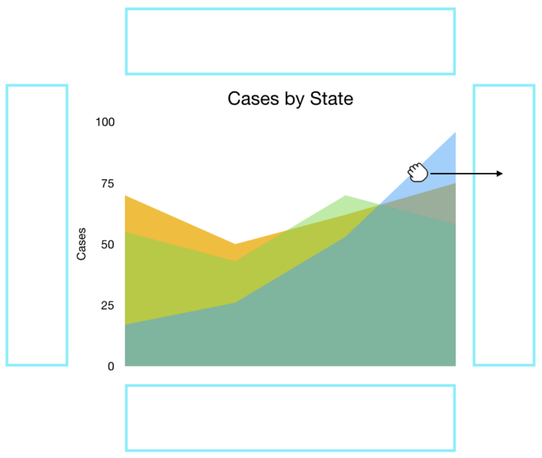
There were several concerns I considered with this approach. The first centered on consistency and affordances. There are few other instances in Lyra in which users can directly edit or manipulate visual elements (e.g. axes, titles) on the canvas so it is unclear whether users would be able to discover this functionality of being able to click and drag a series of marks in a visualization. Additionally, for particularly busy data, it may be difficult for users to select the desired series. The second concern with this method was usability. This process does not scale well for more than a few bins or categories and would require a very repetitive and tedious process for the user to successively pull out each individual field value in a large set. Consider state-by-state election data (as in Figure 1-1(b)), for example, in which a user would have to pull out the series for each of the 50 states one at a time.

An alternative method I sketched included a more data-driven process for creating small multiples. Instead of series being dragged one at a time, a user could instead drag a data field onto a dropzone to facet the dataset on that field. The system would then automatically duplicate the current group into small multiples using that data facet. See Figure 4-14.

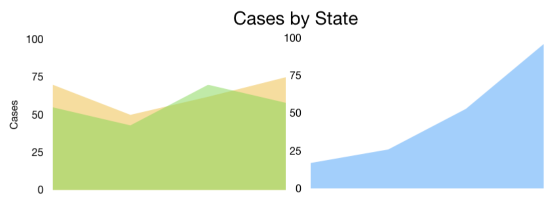
This method of dragging and dropping data fields onto dropzones is a process that already exists within Lyra and thus represents a familiar interaction for users. Additionally, this method is automated and does not require tedious dragging of individual series to create the small multiples. The main drawback with this method is that the layout of the plots is less flexible as users cannot specify the location of each individual facet. Functionality could be added to let users choose whether they want the plots arranged in a row or column and more granularity could be specified in an inspector.



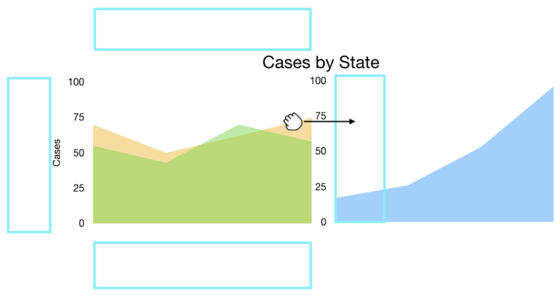
(a)



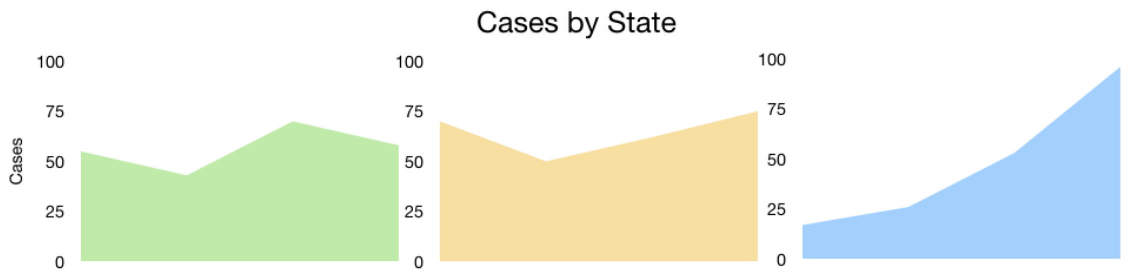
(b)



(c)

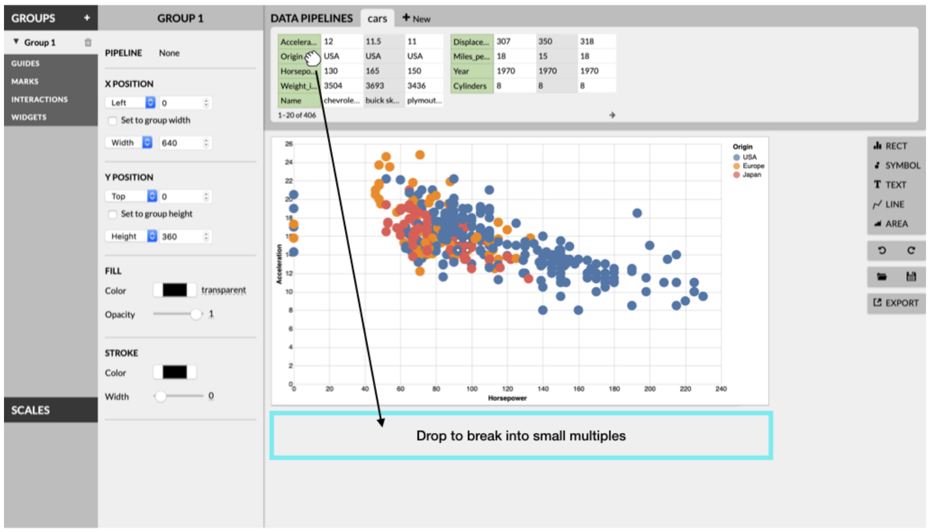


(d)

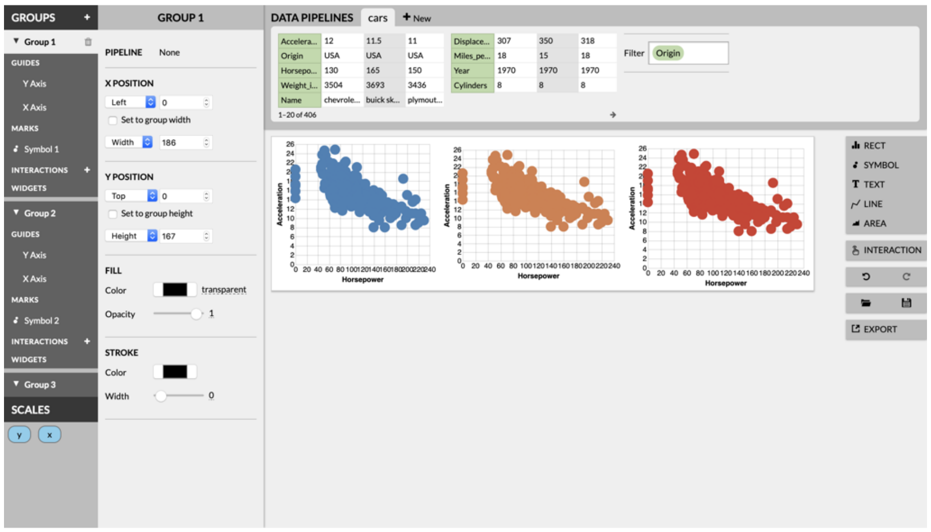


(e)

Figure 4-13: Shows proposed process for creating a small multiple visualization from a chart of 3 series.



(a)



(b)

Figure 4-14: User drags data field onto dropzone to create a facet of the active group.

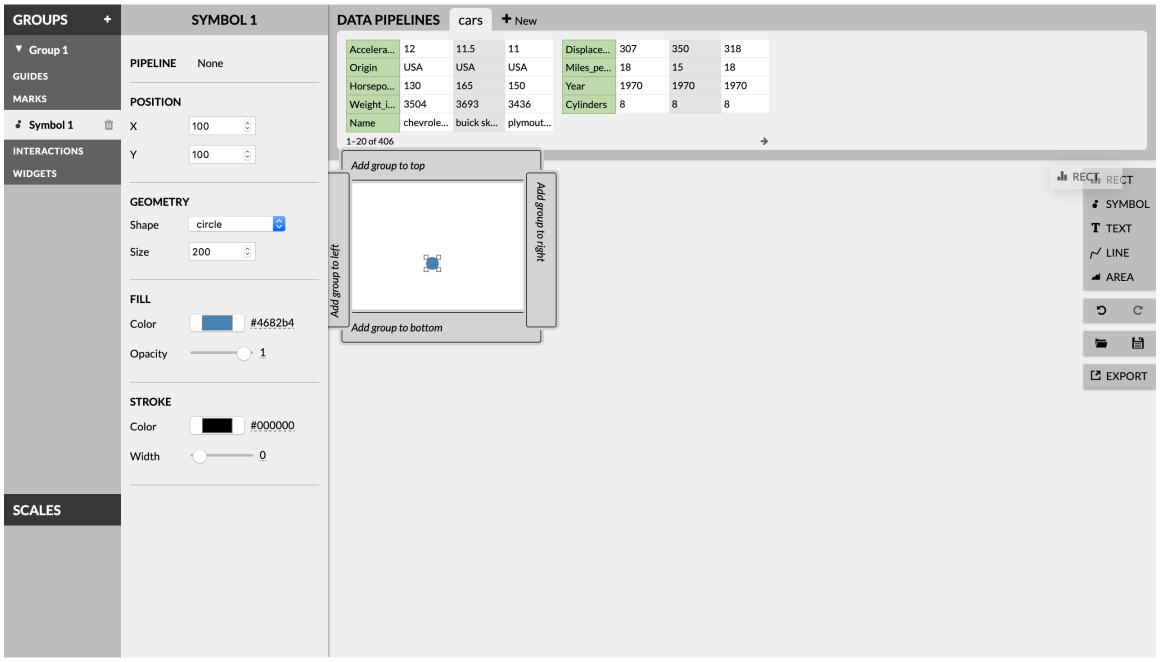
4.4 Final Implementation

After extensive prototyping, I transitioned to implementing the strongest aspects—in terms of usability and expressiveness—of the prototypes in the Lyra codebase. While most of the code from the prototypes was not reusable, I carried over many of the learnings, interaction methods, and base concepts, notably the gridded layout creation scheme, resizing mechanisms, and faceting process.

4.4.1 Layout and Group Creation

The strongest concept from my prototyping phase was the process for creating gridded layouts. In the final implementation for layout, I leveraged existing workflows within Lyra for this new concept. Previously, to create a new group, a user must first click a plus symbol next to the "Groups" header in the left sidebar and then click a mark from the right sidebar to add it to the group. Once a mark is added, the user can then click and drag data fields to create the desired chart. Since adding marks to a group is the first step towards creating a chart before the actual data is bound, I figured for the new mechanism these marks could be directly dragged onto new group dropzones to initialize a group with a particular mark type already added. In the final implementation, a user can click and drag a mark, which makes dropzones appear for all the rows and columns on each side of the current layout (Figure 4-15) and for any empty grid cells (Figure 4-16). The user can then drag that mark to one of those dropzones to create a new group in that position, initialized with the dropped mark type. Dropping in a dropzone around the edges of the visualization will create a new column or row, depending on the side, while dropping in an empty cell will create a new group without increasing the number of rows or columns in the visualization. From there, the process for binding data to marks remains unchanged. Thus, in the span of a couple clicks, the user can easily create new groups, bind data, and create a layout with the desired number and positioning of charts.

The main change from the prototype was that marks are dragged and dropped instead of entire groups. In the prototype, I imagined that an entire chart could be



PIPELINE None

POSITION

X	100
Y	100

GEOMETRY

Shape	circle
Size	200

FILL

Color	#4682b4
Opacity	1

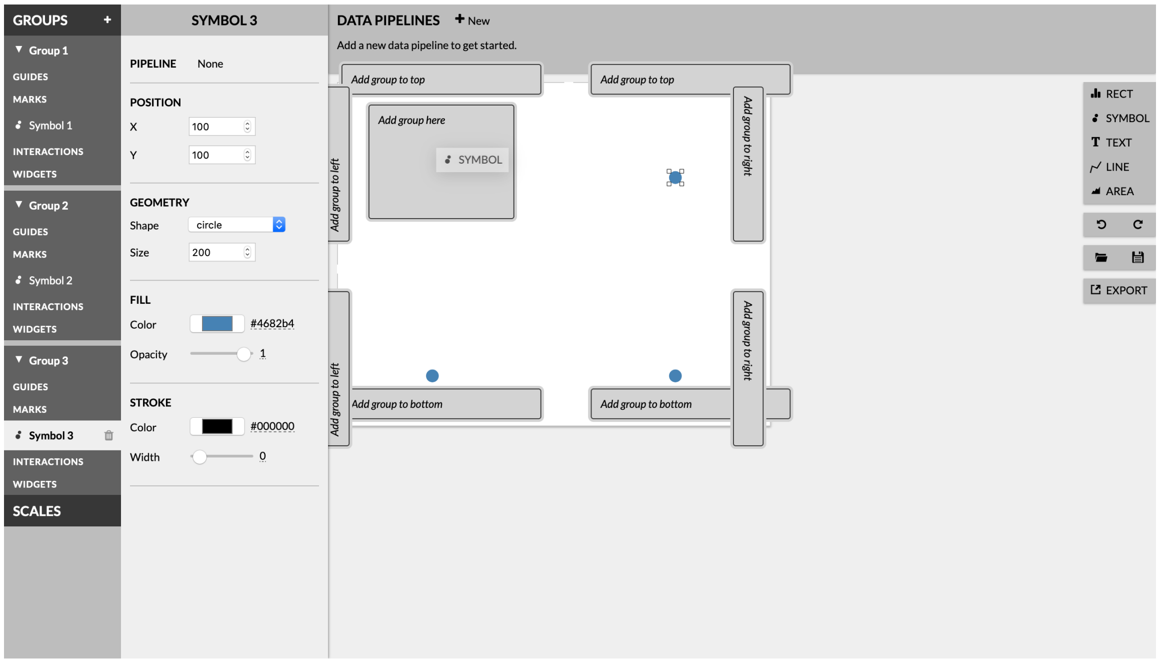
STROKE

Color	#000000
Width	0

Accelera...	12	11.5	11	Displace...	307	350	318
Origin	USA	USA	USA	Miles_pe...	18	15	18
Horsepo...	130	165	150	Year	1970	1970	1970
Weight_i...	3504	3693	3436	Cylinders	8	8	8
Name	chevrole...	buick sk...	plymout...				

1–20 of 406

Figure 4-15: Dropzones appear surrounding layout when a user begins dragging a mark.



Add a new data pipeline to get started.

PIPELINE None

POSITION

X	100
Y	100

GEOMETRY

Shape	circle
Size	200

FILL

Color	#4682b4
Opacity	1

STROKE

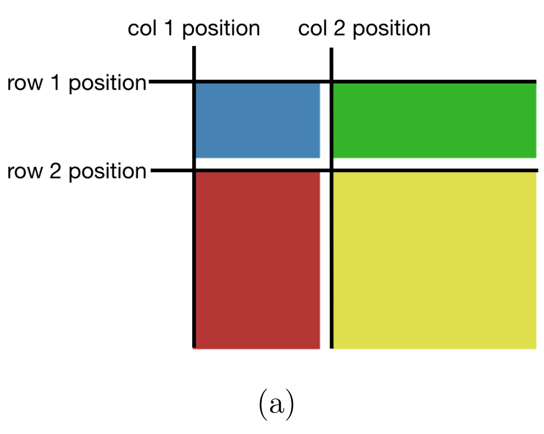
Color	#000000
Width	0

Figure 4-16: Placeholder dropzones appear in empty cells when a user begins dragging a mark.

duplicated on drop as that would make it easier to create small multiples and just edit the data source. When designing a MV with several distinct charts, however, this adds friction as the user would have to first remove all the marks and data fields before creating the new chart. When transitioning back to the Lyra codebase, however, the option of directly dragging marks became apparent. This concept of dragging marks onto the canvas had already been tried in Lyra, but it didn't work previously as the active group was really the only available drop target. Thus, the previous version of Lyra required users to click on the marks to make them appear on the canvas, instead of dragging them. In the new implementation of Lyra, however, adding dropzones that appear when a user starts dragging a mark makes it clear to users what they can do with that mark.

The Lyra implementation of the gridded layout concept largely relies on creating a network of signals used to determine the size and positioning for each row and column in the layout. In Vega, signals act as reactive variables that can update and set values for various properties throughout the specification. Lyra uses signals liberally, setting a signal for almost every supported property defined by the Vega specification, and then converting them to their values when exported to a Vega specification. The new layout signals work much in the same way. They define a system of column/row sizes and positions based on one another so the whole network reacts to resizing or positioning changes of upstream columns or rows (columns closer to the left and rows closer to the top). Upon export, however, a user most likely does not need the resulting Vega specification to have reactive layout components, and thus, these signals are used to set the position and size values for each of the groups in the layout.

To get a better understanding of the signal implementation, I will walk through a small scale example (see Figure 4-17). Each column and row of a layout are defined by a size and position. The size refers to the width for a column and the height of a row, while position is the x position of the start of a column and the y value of the top of a row. All of the columns and rows can be sized independently. The position for each row and column after the first ones, however, can be determined by the size and position of the previous one. In the example, `col_1_size` is set to 50 and `col_2_size`



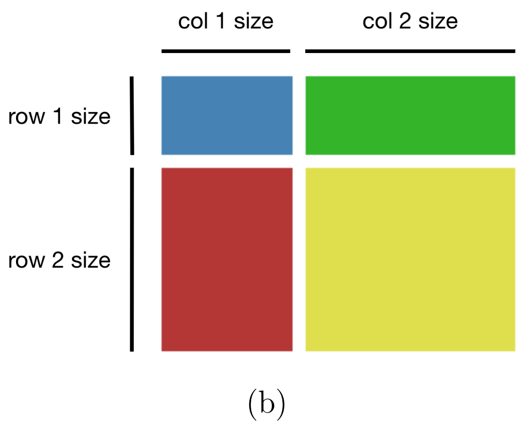


Figure 4-17: Reference example showing layout signal (a) position and (b) size.

to 80. The value for the position of the first column, `col_1_position`, is set to 0. Now instead of setting the value for the position of the second column to 50, the update property of the signal is set to `col_1_position + col_1_size + padding`, which means that if the size or position of the first column is changed, the values for the second column will automatically update as well. These updates can be chained together such that the position of a third column would be based on the size and position of the second, and so on, and the same logic applies to the positioning of rows.

This implementation for signals helps simplify the process for resizing. As changes are made to the size or position of a row or column in the layout, the changes automatically propagate and update the sizing and positioning of all the impacted groups in the layout.

4.4.2 Resize

The resizing features that made it into the final implementation do not fully cover the methods developed in the prototypes, but instead serve as a proof-of-concept of the layout signals and their ability to handle resize changes. As mentioned before, continuous resize for an individual chart already exists within Lyra (via handles), but a mechanism for resizing entire rows or columns was also added for more layout-specific resizing.

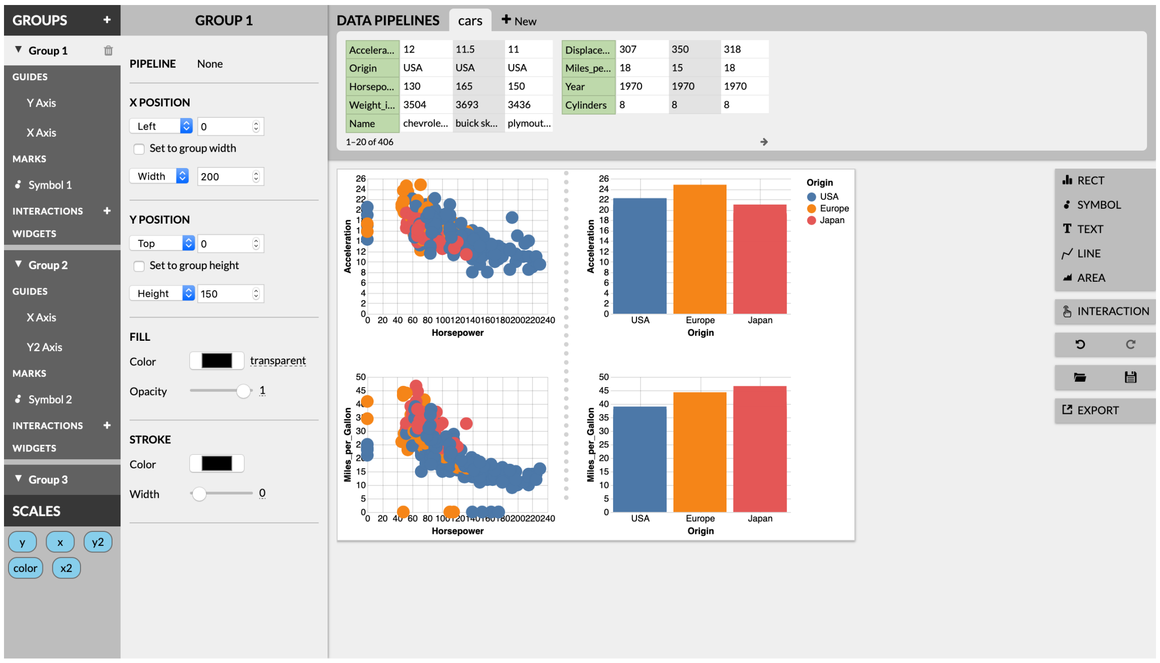
The resize handles for an entire row or column appear as a horizontal or vertical dotted line between two rows or columns, respectively (Figure 4-18). These lines are only visible on hover so that they do not take away from the overall visualization. These lines can be dragged perpendicular to their orientation to increase or decrease the size of the adjacent rows or columns.

The implementation logic for these resize lines is based on the new layout signals described in the layout section. When a line is dragged, the signal value for the size of the corresponding row or column is updated depending on the direction and magnitude of the drag. For example, dragging a vertical line to the right of column 1 to the right 20 px would increase the size of column 1 by 20 px, while dragging to the left would decrease the size of the column. The signals for the size of all the groups in a row or column are linked to the signal for the layout row or column size (e.g. the width of a group in column 1 row 2 would be set to the size of column 1, while the height would be set to the size of row 2), and thus, update when the corresponding layout signal is changed. Additionally, since the signals for the group positions are also dependent on signals for the row and column sizes, all of the downstream groups reposition themselves to account for the changing column or row size.

4.4.3 Faceting

Faceting was included in the final implementation to make creating small multiples easier. The process for creating a faceted layout looks similar to the prototyped interaction. When a data field is clicked and dragged, users are presented with new dropzones that allow for faceting groups in a row or column (Figure 4-19), with the dataset partitioned by the values of the dropped data field. Upon dropping the field in one of those dropzones, the facet is applied to the current group and the layout is set to a row or column for the faceted groups, depending on which dropzone was selected. Since the facet is applied to a specific group and faceted groups are only created at runtime, any changes made after-the-fact to one of the groups (e.g. adding data, changing the fill color, etc) propagate to all other groups in the facet.

One of the key implementation decisions for faceting was how to apply a layout.



Accelera...	12	11.5	11	Displace...	307	350	318
Origin	USA	USA	USA	Miles_pe...	18	15	18
Horsepo...	130	165	150	Year	1970	1970	1970
Weight_i...	3504	3693	3436	Cylinders	8	8	8
Name	chevrole...	buick sk...	plymout...				

1–20 of 406

PIPELINE None

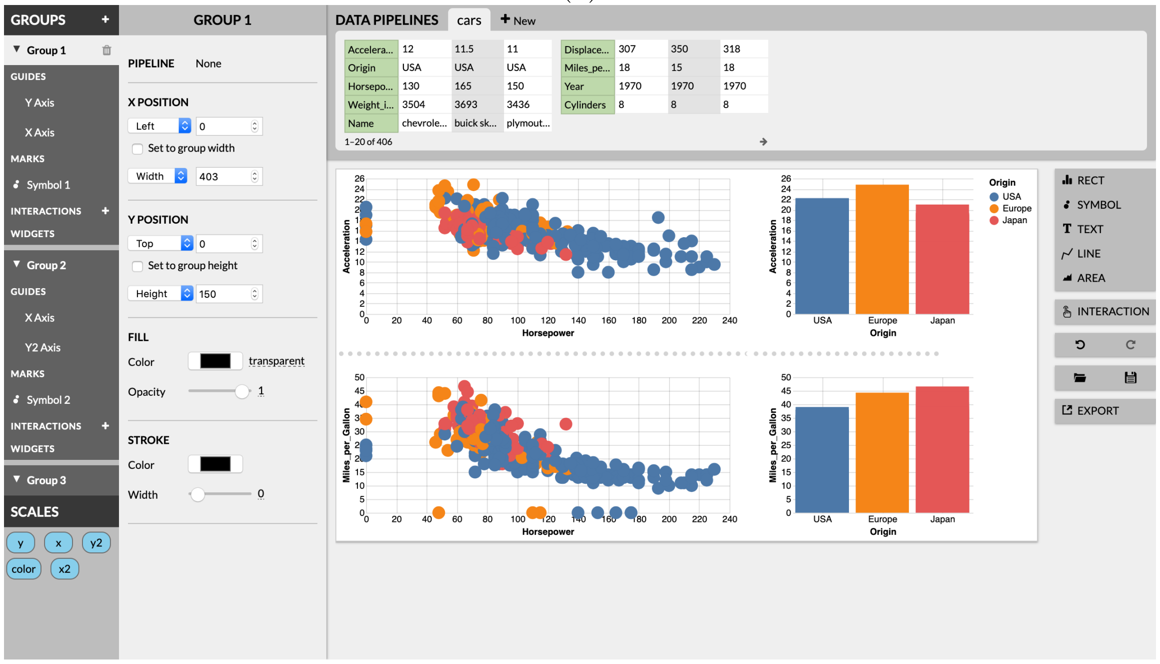
X POSITION
Left 0
Set to group width
Width 200

Y POSITION
Top 0
Set to group height
Height 150

FILL
Color transparent
Opacity 1

STROKE
Color
Width 0

(a)



Accelera...	12	11.5	11	Displace...	307	350	318
Origin	USA	USA	USA	Miles_pe...	18	15	18
Horsepo...	130	165	150	Year	1970	1970	1970
Weight_i...	3504	3693	3436	Cylinders	8	8	8
Name	chevrole...	buick sk...	plymout...				

1–20 of 406

PIPELINE None

X POSITION
Left 0
Set to group width
Width 403

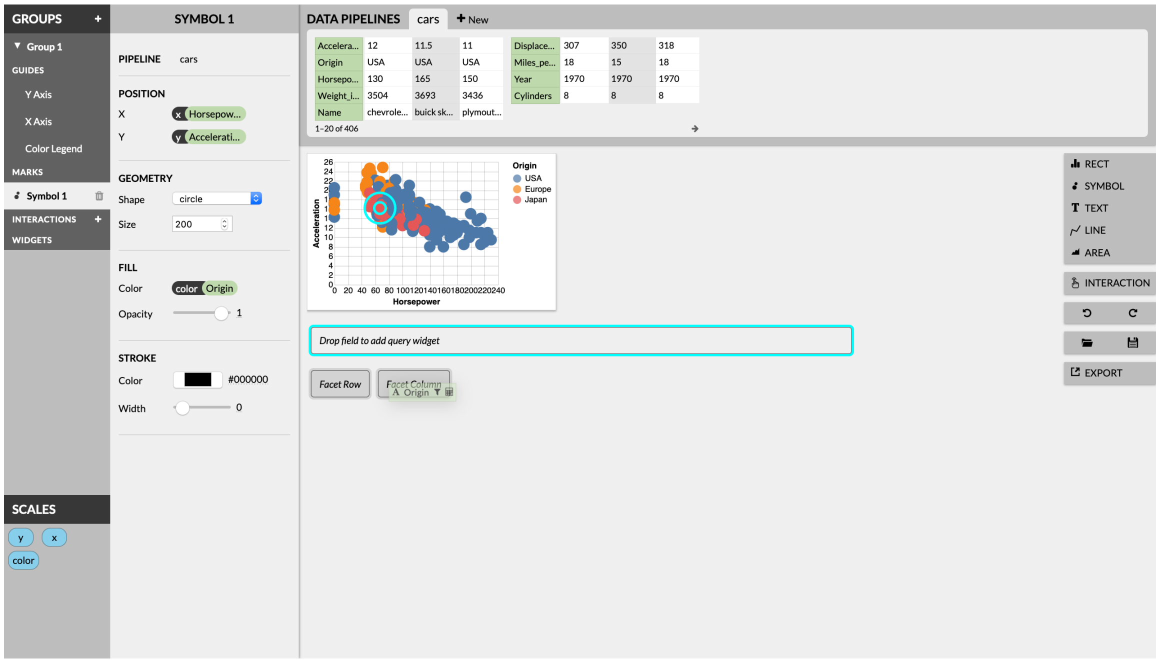
Y POSITION
Top 0
Set to group height
Height 150

FILL
Color transparent
Opacity 1

STROKE
Color
Width 0

(b)

Figure 4-18: On hover, the resize line between (a) columns or (b) rows becomes visible.



PIPELINE cars

POSITION
X: x Horsepow...
Y: y Accelerati...

GEOMETRY
Shape: circle
Size: 200

FILL
Color: color Origin
Opacity: 1

STROKE
Color: #000000
Width: 0

Accelera...	12	11.5	11	Displace...	307	350	318
Origin	USA	USA	USA	Miles_pe...	18	15	18
Horsepo...	130	165	150	Year	1970	1970	1970
Weight_i...	3504	3693	3436	Cylinders	8	8	8
Name	chevrole...	buick sk...	plymout...				

1–20 of 406

Figure 4-19: Dropzones for faceting appear below the canvas when a user clicks and drags a data field.

Unlike the previous methods for creating layout, faceting is inherently data-driven. In the Vega, a facet can be applied to a group, and the number of faceted groups are determined and created at run-time. This presents a challenge to the layout implementation above, in which distinct signals are created for the positioning and sizing of each group in a layout. Since faceted groups are only created at run-time, creating signals ahead of time for each group created by the facet is not feasible. Thus, I instead turned to the Vega layout specification, in which I utilized the *columns* property to specify the number of columns in the layout. When left undefined, *columns* defaults to the number of groups in the visualization, and thus the layout appears as a row. When *columns* is set to 1, on the other hand, all the groups are arranged in a single column. Any number in between will "wrap" the groups in the visualization such that the number of columns is equal to the value of the property. The resultant visualization can have any number of rows necessary to satisfy the constraint on the number of columns. Since I only provide dropzones for faceted groups to be arranged in either a row or column, the layout *columns* value is either set to the default or set to 1, respectively, in my current implementation. This implementation makes it possible for faceted groups to be positioned at run-time.

Chapter 5

Evaluation

I evaluate the new additions to Lyra by walking through some use cases to show the new processes and showing an example gallery to highlight the range of multiple-view visualizations that can be created with the new functionality. The examples shown in this section are primarily recreations of MV visualizations from the Vega-Lite example gallery with the aim of highlighting different layouts and mark types, and showing the new features, like faceting and row/column resizing.

First, I will walk through the general layout, resize, and facet processes. Figure 5-1 shows the process of adding a group to create a layout. Starting from an initial empty group, a user first adds a symbol mark, and then drags a rectangle mark to the right most dropzone. This results in a second group placed to the right of the first, initialized with a rectangle mark.

In the cases where there are multiple rows and columns and not all cells of the layout are filled in with groups, placeholder dropzones also appear as options for dropping a new mark, as seen in Figure 5-2. Dropping a mark in a placeholder dropzone results in adding a group to that empty spot in the layout.

Figure 5-3 shows the process for resizing a column and row of a layout. Starting from four equal-sized groups in a two-by-two grid created with the process described above, a user first hovers over the region between the columns to see the resize line appear. A user then clicks and drags the resize line to the right to increase the size of the first column. Next, the same process can be repeated with the horizontal line

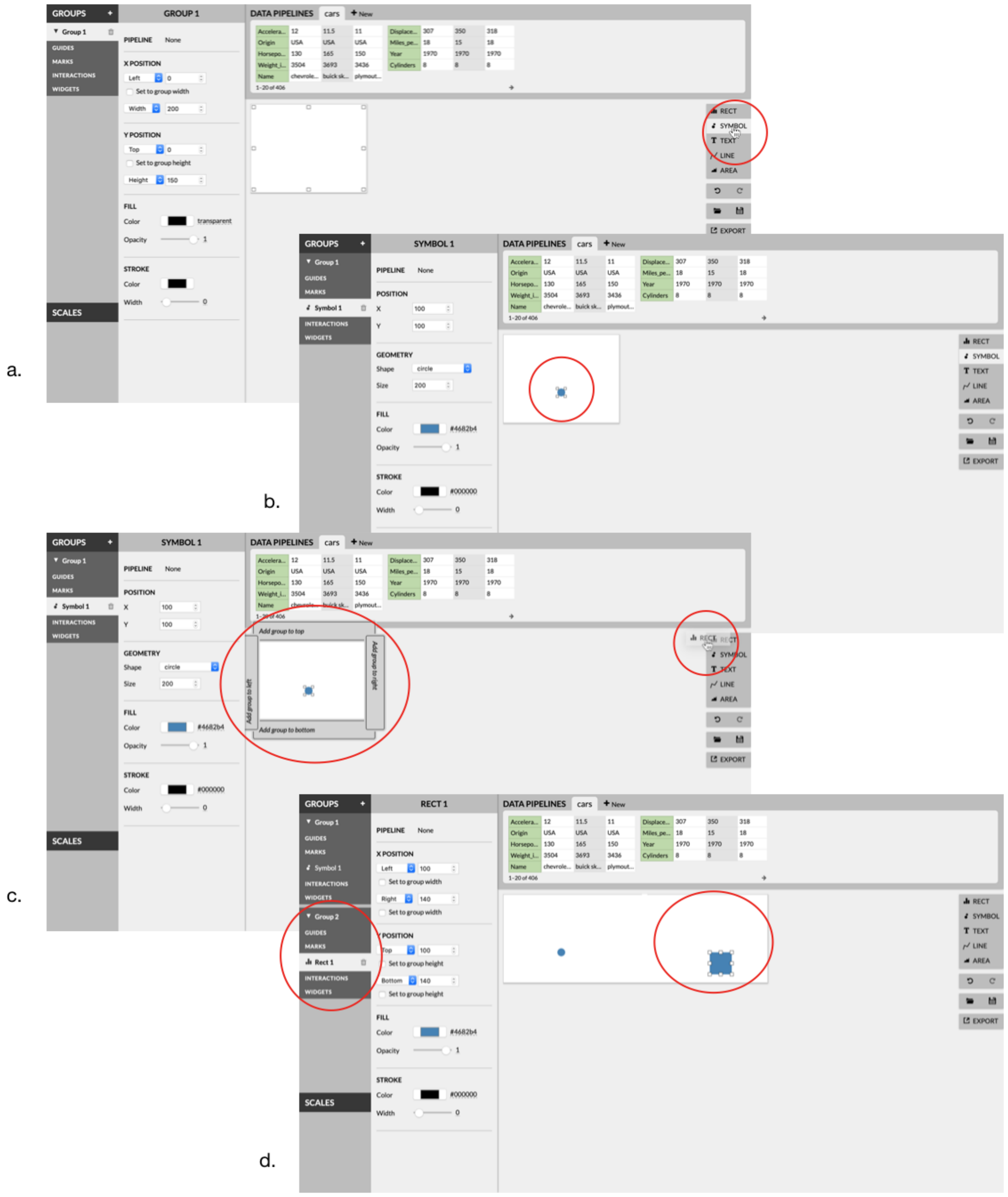
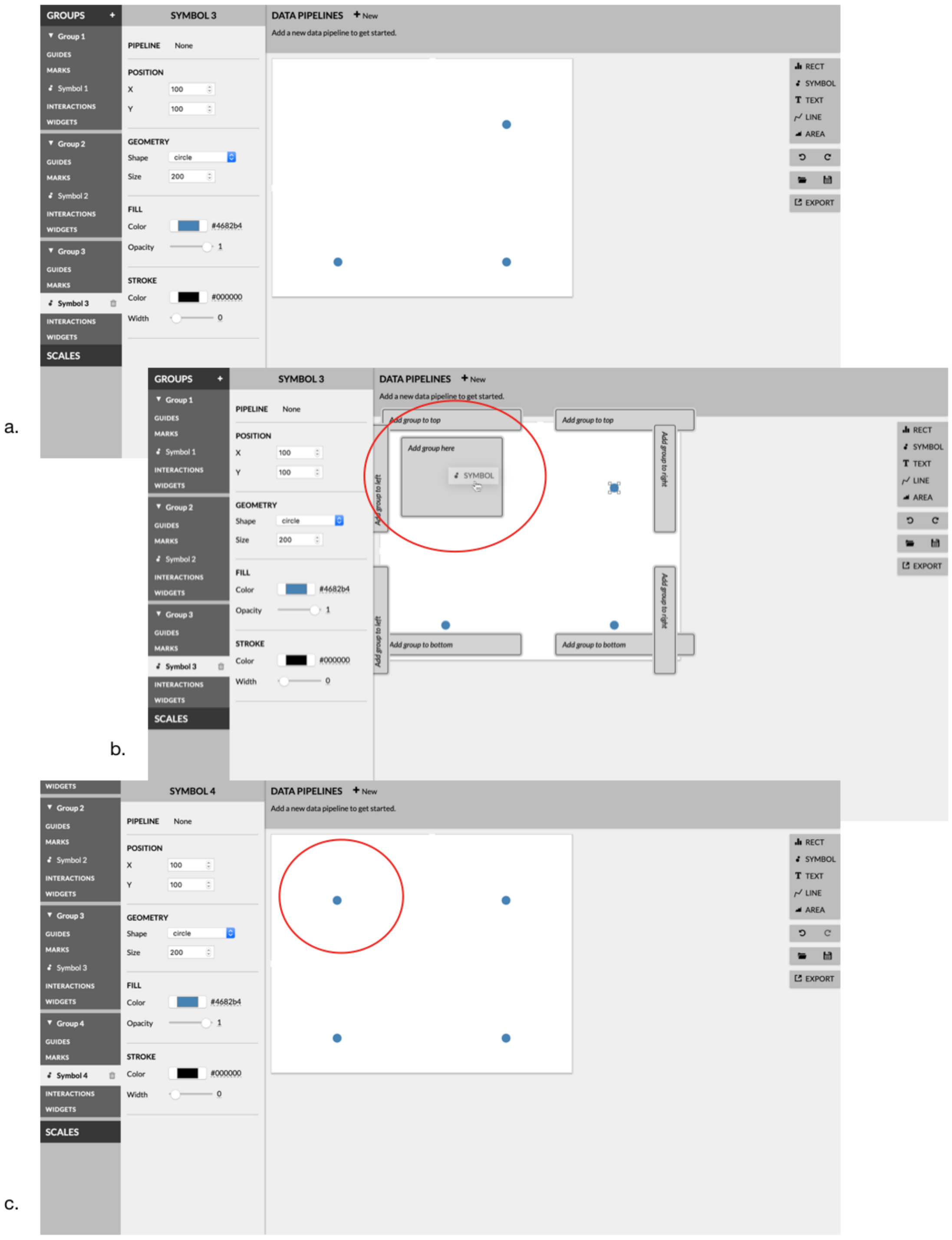


Figure 5-1: Step by step process for creating a layout with the dropzones starting from an empty canvas. Red circles added for clarity.



PIPELINE None

POSITION
X 100
Y 100

GEOMETRY
Shape circle
Size 200

FILL
Color #4682b4
Opacity 1

STROKE
Color #000000
Width 0

Figure 5-2: Step by step process for dropping a mark in a placeholder dropzone and initializing a new group in that position. Red circles added for clarity.

between the two rows. This time the line is dragged downward to increase the size of the first row. These resizing capabilities can be used to enlarge detailed charts or add emphasis to particular charts by making them relatively larger than others. After the resizing shown in Figure 5-3, the top left group is larger than all the others, making it the most prominent, while the bottom right one is the smallest. Additionally, by enlarging the first column, the x-axis detail on both scatter plots is more easily readable.

Finally, Figure 5-4 walks through the process of applying a facet. Starting from a single group, the user clicks and drags the "Origin" field, causing several dropzones to appear. The user then drags to the dropzone labeled "Facet Column," which creates small multiples faceted on the origin field arranged in a single column. Since there are only 3 values for the origin field in this dataset—USA, Japan, and Europe—exactly 3 groups are created by the facet operation. The origin field is an ordinal data type, which is typical for small multiples, but faceting can also be performed with quantitative data and the facet operation will bin the values.

Now having demonstrated the basic processes for creating MV visualizations with the new methods, I will show a range of examples. Figure 5-5 shows a large layout with nine scatter plots arranged in a three-by-three matrix. In this example, the nine views each have distinct x and y axes, differentiating it from a small multiples plot. Each chart plots two of only three data fields from a cars dataset and the matrix represents every permutation of those data fields. The origin field is represented as a color scale and a single legend for all the charts is in the top right corner. This layout maintains all the defaults so no resizing was needed.

Figure 5-6 has a range of different MVs with two views and several different graphical marks, including symbol, bar, and area. Figure 5-6(a) contains two views of Seattle weather data. The larger top view shows time series data of maximum temperature on the y-axis, precipitation encoded as size, and weather as color. The smaller bottom view shows a histogram of weather conditions. Figure 5-6(b) has an overview-and-detail style visualization with two area plots of the same stock data. A smaller view shows the whole time series, while an enlarged top view highlights

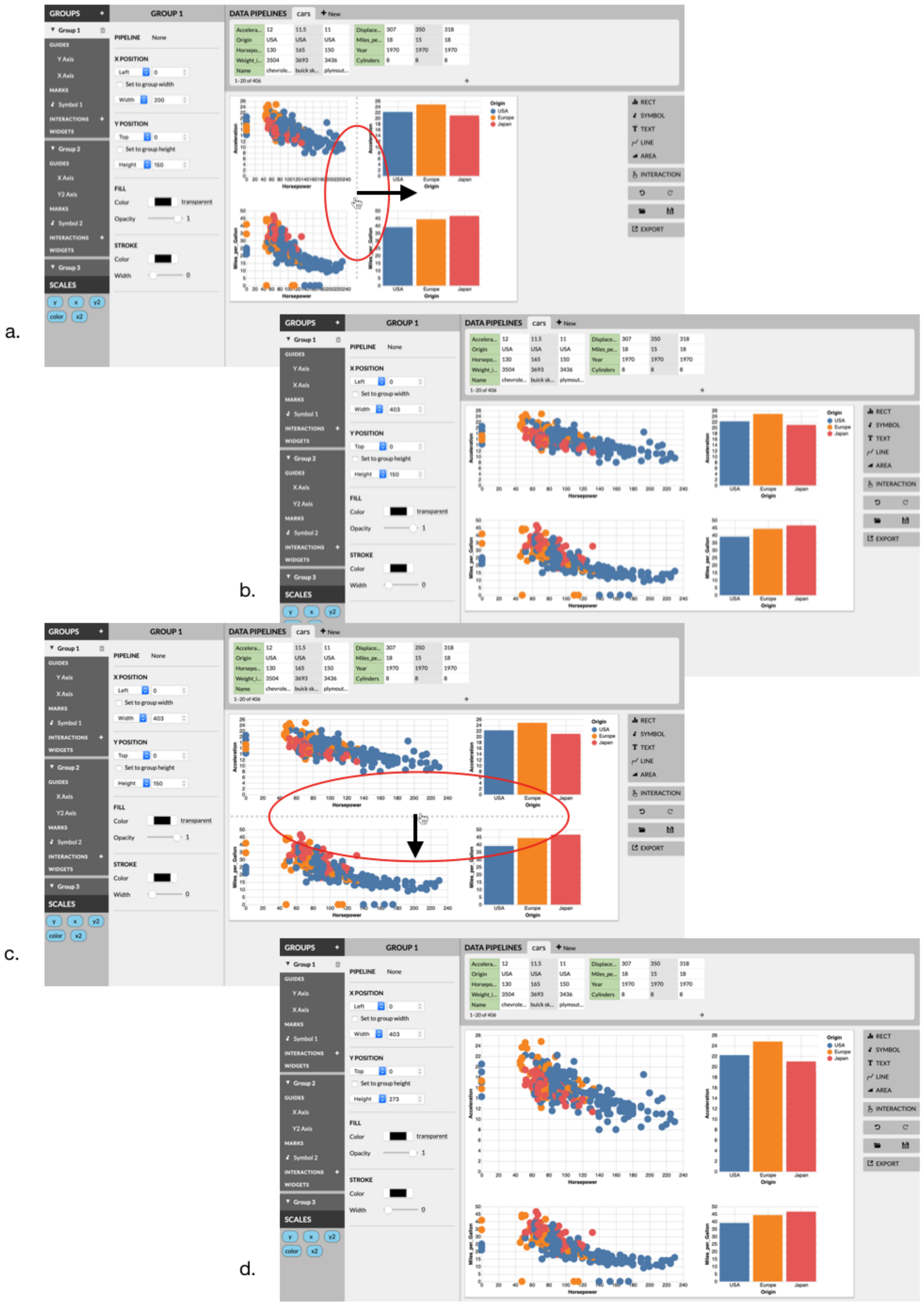


Figure 5-3: (a) Starting from a 2x2 grid of groups, (b) a user drags a column resize line to the right and (c) a row resize line down to (d) adjust the column and row sizes of the layout. Red circles and arrows added for clarity.

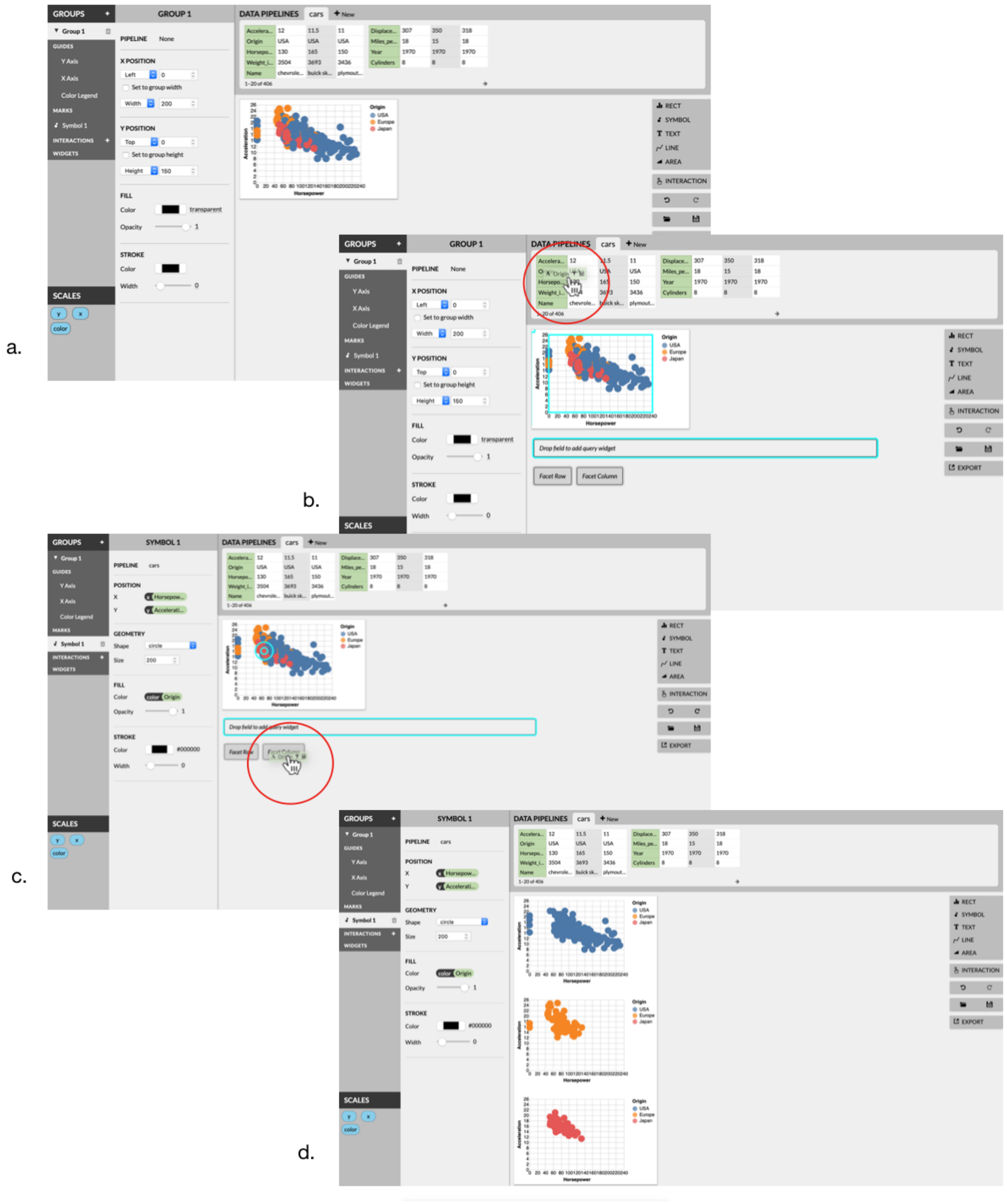


Figure 5-4: Process of applying a column facet to a group. Red circles added for clarity.

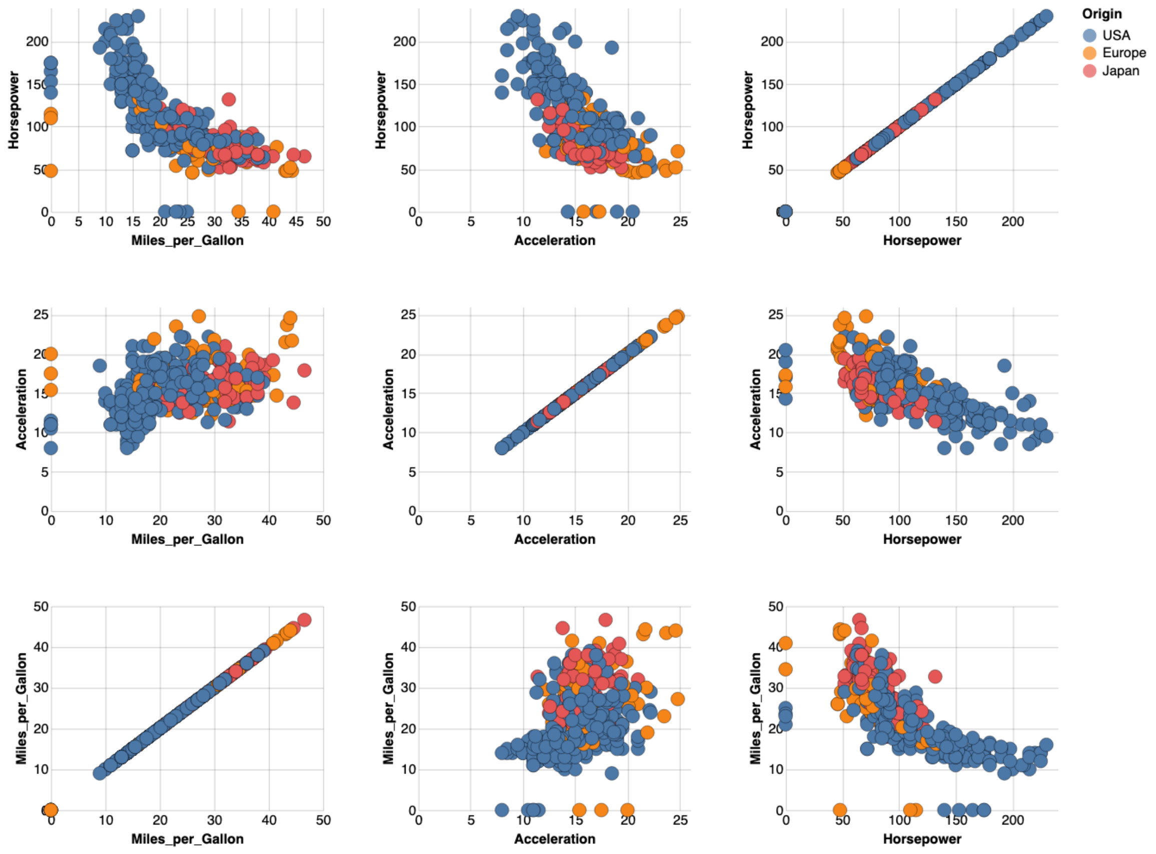


Figure 5-5: MV with a 3x3 grid of groups.

detail for just the selected range. Finally, Figure 5-6(c) and Figure 5-6(d) show a linking interaction between the bar chart and scatter plot views of car data. Linking interactions are a common use case for MV visualizations in which interacting with one of the views (brushing or clicking to select part of the data) causes the other view to update based on the selected data.

Figure 5-7 shows an unusual layout with 5 views. Here, the groups are laid out with one in the first row, three in the second, and 1 in the third. This case highlights an advantage of my gridded layout implementation over the Vega layout specification. The Vega layout specification strives to completely fill each row to the desired number of columns before wrapping around to the next row. My implementation affords additional expressiveness as users have more control over where they place groups within the layout.

Figure 5-8 compares two facet layouts for the same set of three groups. In Figure 5-8(a), the faceted groups are laid out in a column, while Figure 5-8(b) shows them arranged in a row. These are currently the only layout options available to users when faceting. Figure 5-9 facets plots of life expectancy versus fertility by "cluster" (continent) in a row. This plot indicates a shortcoming of this implementation. As the number of groups in the facet increases, it becomes increasingly difficult to view them all in a normal viewing space. Here, the exported visualization is shrunk to fit the width of this page, and reading the small text on the charts is nearly impossible at this scale.

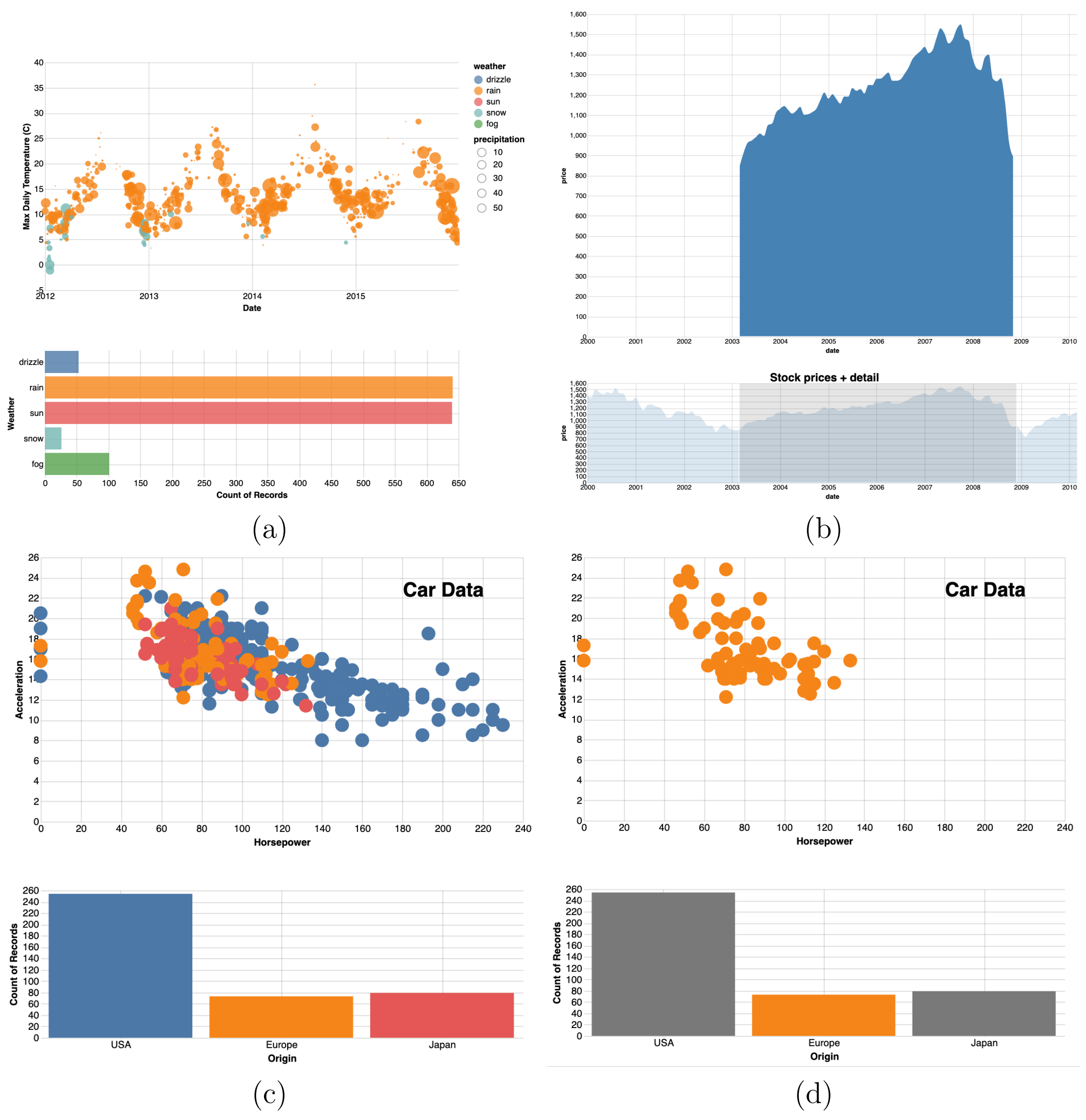


Figure 5-6: Examples of 2-view MVs. (a) Shows bubble and histogram charts for Seattle weather data. (b) Shows overview-and-detail plots for stock data with a brushing interaction on the bottom, overview plot filtering data on the top, detail chart. (c and d) Show another linking example in which clicking on bars on the histogram filters data on the scatterplot.

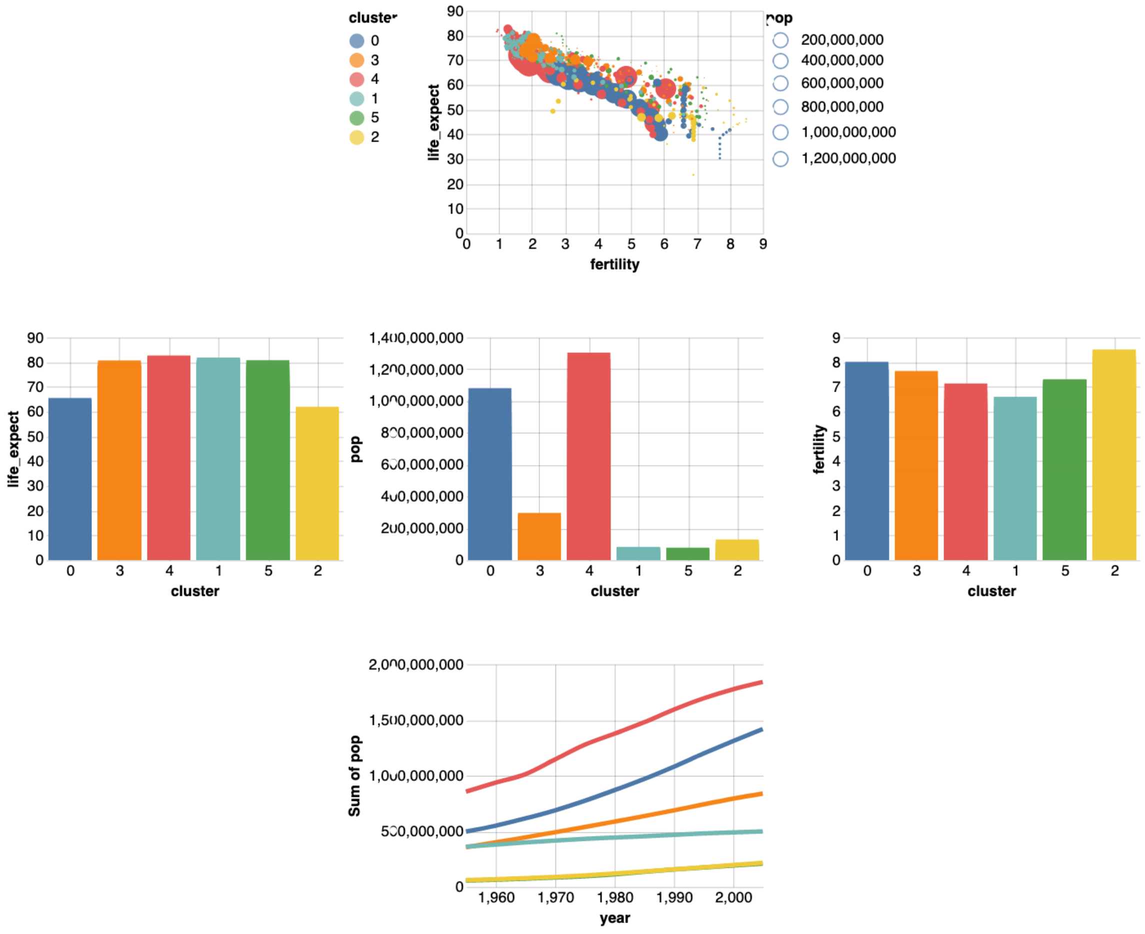


Figure 5-7: MV with 5 groups in non-standard arrangement.

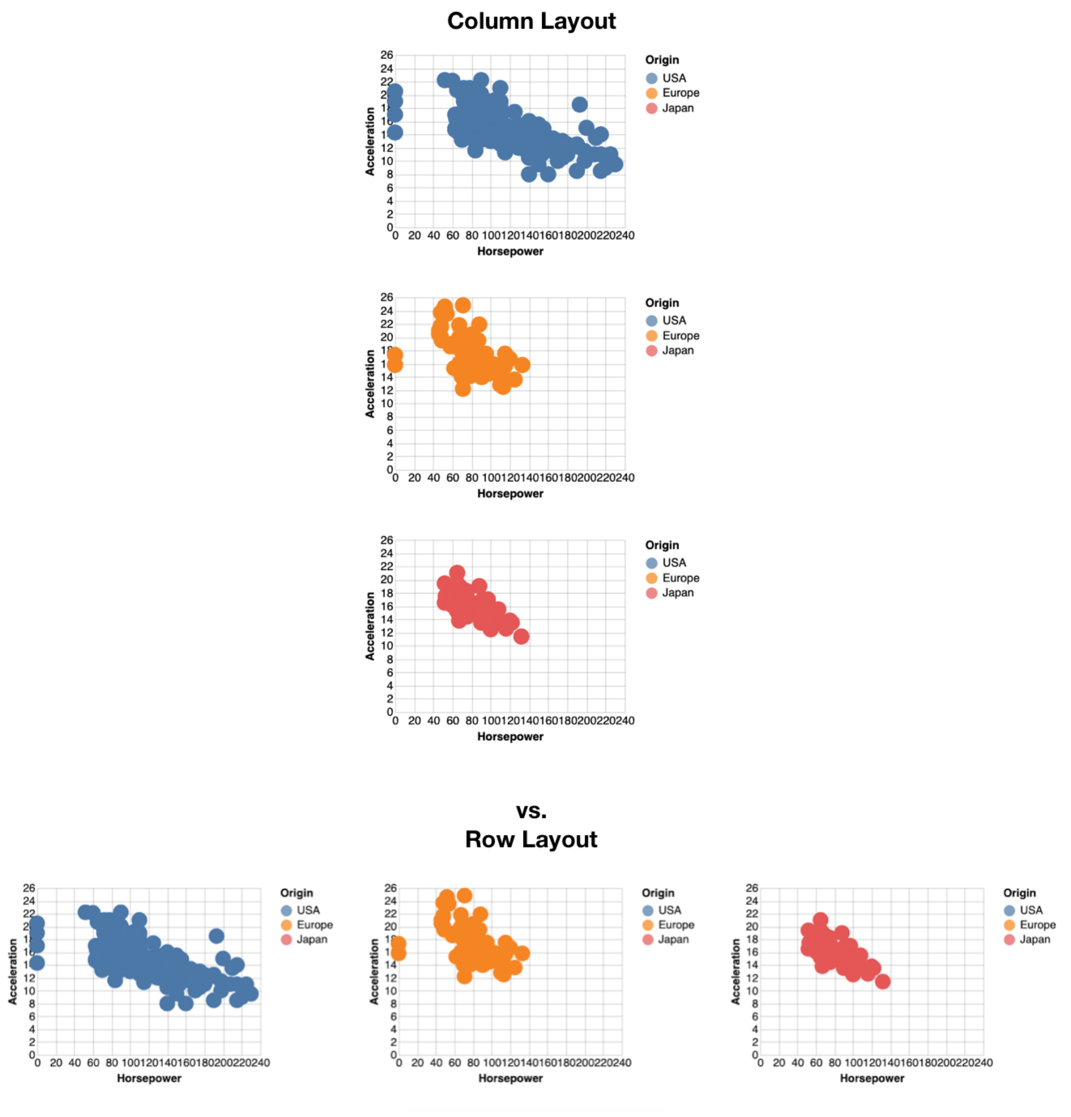


Figure 5-8: Shows same group and facet but wtih (top) row and (bottom) column layout.

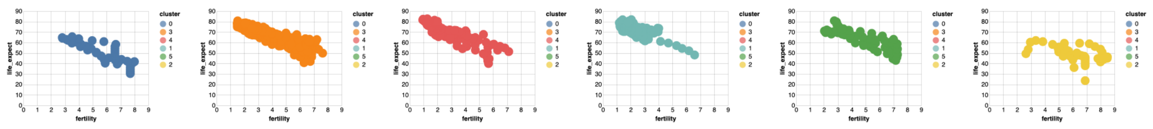


Figure 5-9: Row layout applied to a 6-group facet.

Chapter 6

Discussion

The methods I implemented for creating gridded layouts and automatic faceting make creating multiple-view and small-multiple visualizations easier for users. Compared to the previous methods requiring button clicks to add groups and manual positioning of charts, the process of dragging and dropping graphical marks onto positional dropzones to add groups and create a layout for a MV visualization is more in-line with the principles of direct manipulation. These processes reduced *articulatory distance* as adding a group to particular spot requires dragging a mark to a dropzone in that position [1]. Unlike other visualization software tools, Lyra now has processes specifically targeting creation of multiple-view visualizations. Similarly, faceting is much easier for users. This operation requires a drag-and-drop of a data field onto a dropzone and multiple groups are automatically created and positioned based on the facet definition. Faceted groups were previously not supported in Lyra so this feature both adds expressiveness of the possible visualizations that can be created with Lyra and is automated to require less work of users.

Throughout the prototyping process, numerous trade-offs were assessed before arriving at the final implementation. At a high level, the final implementation for layout is less flexible than the snap-to-grid functionality but provides easier mechanisms for users to create and adjust larger, more complex layouts. Similarly, the current implementation does not allow users to quickly explore various layouts for a given number of groups as was possible with predefined layout options, but users are not constrained to only the layouts provided. The current implementation was judged to be sufficiently expressive to cover most gridded layout designs and easy to use.

6.1 Limitations

While greatly improving the process for creating MV visualizations, there are still several limitations with the current implementation regarding resizing elements and layout compatibility. For one, there are no current mechanisms for creating groups in a layout that are greater than 1 grid unit in size. In the prototypes, multiple dropzones could be selected to initialize a larger group and the individual discrete resize feature allowed for increasing the size of the group by an entire row or column. These features are somewhat redundant and the second mechanism is strictly more expressive. Consequently, adding just the resize mechanism would greatly increase the expressiveness of the implementation.

The other notable limitation of the current implementation is that the facet and general layouts are not compatible. The Vega layout is a top-level property, meaning that it overrides any other values for ordering or arranging groups in the specification. Since the general layout implementation uses signals to directly set the x and y values for groups, these are overridden when a Vega layout specification is defined. Thus, with the current implementation, users can either have a general layout with any arbitrary number and positioning of groups or they can create small multiples with a Vega layout specified, but both cannot coexist in a single visualization at this point. Future work for developing a mechanism for laying out faceted groups in a way that does not conflict with the general layout case is discussed in the next section.

Chapter 7

Future Work

Future work for my implementation of multiple view visualizations centers on expanding the expressiveness of the current implementation. This includes implementing further resize features, more compatible facet and general layout implementations, and a user study for feedback.

There were several concepts from the layout resizing prototypes that did not make it into the final implementation but would greatly expand the expressiveness of the current implementation. One such concept would be the individual discrete resize, which would allow for charts larger than one unit in size. This feature would require new handles for users to drag that are distinct from the current handles for continuous resize to visually indicate a distinct behavior for this element. Additionally, whole layout resize is another concept worth carrying over as it allows for rescaling the entire layout, while keeping the relative sizes of the charts the same. This feature could reuse the existing handles since it is a continuous process but there would need to be a new method of selecting the entire layout instead of individual groups within it. Combined, these resize concepts would allow for a greater variety of possible visualizations that could be created with the system.

Further work could also be done to provide additional affordances for the facet layout. As it stands, the automatic layout applied with a facet can either be a row or column. Once a layout is applied, the user has limited flexibility to make changes to that layout. With the current implementation, however, more control over the

Vega layout specification for the facet could be exposed to users. For example, users could be allowed to choose the number of columns so that a multiple row and column gridded layout could be created. Other properties could allow the user to specify spacing or padding between charts.

Another area of future work would be to merge the two layout concepts or at least make them compatible. More work could be done to explore alternative methods of fitting faceted groups into the general layout concept for sizing and positioning. Alternatively, faceted layouts could be implemented as a subset of the general layout, allowing for multiple facets or faceted and non-faceted groups to co-existing in a single visualization. Prior work from Xi Chen et al. indicates it is common to have small-multiples embedded within a larger visualization layout and their hierarchical concept developed for handling small multiples supports the idea of treating small multiples as a group and a single cell in a larger layout. Thus, this will need to be an ongoing area of work to make these types of MV visualizations buildable in Lyra.

Finally, a user study would help to understand the intuitiveness of the system. In a user study, the new methods could be compared to existing processes for creating MV visualizations in Lyra and other visualization tools. A user study would also help assess how successful the new MV processes are for making such visualizations easier to create and provide important feedback for future development for these methods.

Bibliography

[1] Alan F. Blackwell and Thomas R.G. Green. *Notational Systems – the Cognitive Dimensions of Notations framework*, pages 103–134. HCI Models, Theories, and Frameworks: Toward a Multidisciplinary Science. Morgan Kaufmann, San Francisco, 2003.

[2] X. Chen, W. Zeng, Y. Lin, H. M. Al-maneea, J. Roberts, and R. Chang. Composition and configuration patterns in multiple-view visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 2020.

[3] Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. Direct manipulation interfaces. *Human–Computer Interaction*, 1(4):311–338, 1985.

[4] Zhicheng Liu, John Thompson, Alan Wilson, Mira Dontcheva, James Delorey, Sam Grigg, Bernard Kerr, and John Stasko. Data illustrator: Augmenting vector design tools with lazy data binding for expressive visualization authoring. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, page 1–13, New York, NY, USA, 2018. Association for Computing Machinery.

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

[6] B. Saket, S. Huron, C. Perin, and A. Endert. Investigating direct manipulation of graphical encodings as a method for user interaction. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):482–491, 2020.

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

[8] Arvind Satyanarayan and Jeffrey Heer. Lyra: An interactive visualization design environment. *Computer Graphics Forum*, 33(3):351–360, 2014.

[9] Vega: A visualization grammar, 2014. https://vega.github.io/vega/.

[10] Marks, 2014. https://vega.github.io/vega/docs/marks/#from.

[11] Group mark, 2014. https://vega.github.io/vega/docs/marks/group/.

[12] Layout, 2014. https://vega.github.io/vega/docs/layout/.

[13] Signals, 2014. https://vega.github.io/vega/docs/signals/.

[14] Where new cases are higher and staying high, 2020. https://www.nytimes.com/interactive/2020/us/coronavirus-us-cases.html?action=click&module=Top%20Stories&pgtype=Homepage.

[15] Randy Yeip, Stuart A. Thompson, and Will Welch. A field guide to red and blue america, 2016. http://graphics.wsj.com/elections/2016/field-guide-red-blue-america/.

[16] J. Zong, D. Barnwal, R. Neogy, and A. Satyanarayan. Lyra 2: Designing interactive visualizations by demonstration. *IEEE Transactions on Visualization and Computer Graphics*, 2020.