Deconstructing and Restyling D3 Visualizations

Jonathan Harper
University of California, Berkeley
jharper@eecs.berkeley.edu

Maneesh Agrawala
University of California, Berkeley
maneesh@cs.berkeley.edu

ABSTRACT
The D3 JavaScript library has become a ubiquitous tool for developing visualizations on the Web. Yet, once a D3 visualization is published online its visual style is difficult to change. We present a pair of tools for deconstructing and restyling existing D3 visualizations. Our deconstruction tool analyzes a D3 visualization to extract the data, the marks and the mappings between them. Our restyling tool lets users modify the visual attributes of the marks as well as the mappings from the data to these attributes. Together our tools allow users to easily modify D3 visualizations without examining the underlying code and we show how they can be used to deconstruct and restyle a variety of D3 visualizations.

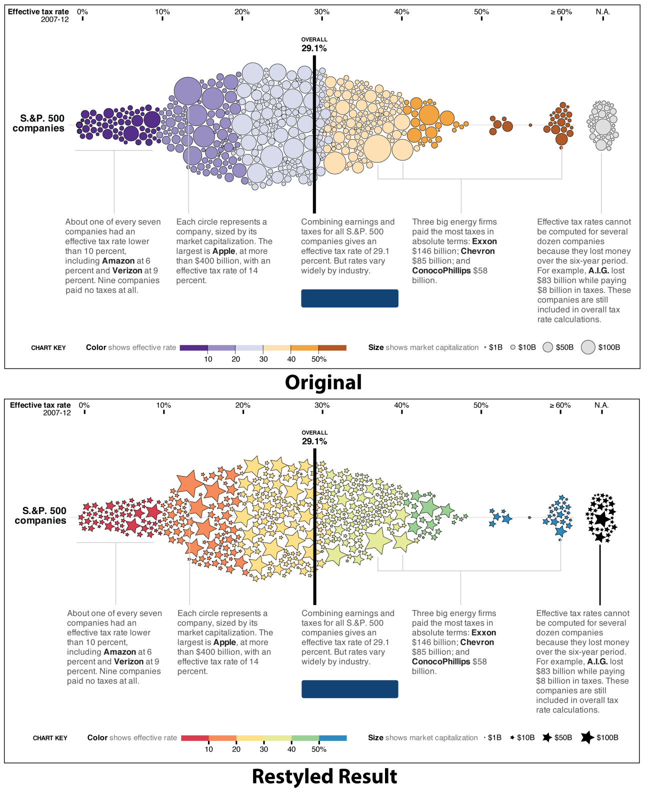
Author Keywords
Visualization, chart understanding, restyling, D3

ACM Classification Keywords
H.5.2. Information Interfaces and Presentation (e.g. HCI): User Interfaces – Graphical User Interfaces

INTRODUCTION
The D3 library [7] has emerged as one of the most popular tools for producing visualizations on the Web. News and media websites such as the New York Times [22], MTV [21] and the Boston Globe [8] regularly publish D3 visualizations that reach hundreds of thousands of viewers. Beyond these high-traffic sites, a large community of Web developers has posted thousands of example D3 visualizations online [33, 25, 2].

Because D3 is built using Web standards (e.g. HTML, SVG, JavaScript) skilled developers can modify the visual style of an existing visualization by copying and editing the code. Yet, such modification requires significant coding expertise in the Web standards and an interest in understanding how the visualization code works. So for most viewers, once a D3 visualization is published its visual style is effectively fixed.

Nevertheless, there are many reasons a viewer may want to modify the style of a visualization. For instance, a colorblind viewer may need to shift the red and green marks in a chart to colors that are easier to distinguish. A blogger might wish to embed a scatterplot in her blog, but change the color, shape and size encodings of the marks to match the overall design



About one of every seven companies had an effective tax rate lower than 10 percent, including **Amazon** at 6 percent and **Verizon** at 9 percent. Nine companies paid no taxes at all.

Each circle represents a company, sized by its market capitalization. The largest is **Apple**, at more than $400 billion, with an effective tax rate of 14 percent.

Combining earnings and taxes for all S.&P. 500 companies gives an effective tax rate of 29.1 percent. But rates vary widely by industry.

Three big energy firms paid the most taxes in absolute terms: **Exxon** $146 billion; **Chevron** $85 billion; and **ConocoPhillips** $58 billion.

Effective tax rates cannot be computed for several dozen companies because they lost money over the six-year period. For example, **A.I.G.** lost $83 billion while paying $8 billion in taxes. These companies are still included in overall tax rate calculations.

Figure 1. Our deconstruction and restyling tools allow a viewer to modify the shape and color encodings of a D3 visualization published by the New York Times. The restyled result changes the circles to stars and re-maps the tax rate data field to a new color scale.

of her webpage. A viewer might prefer to view a pie chart in the form of a bar chart so that it is easier to compare the data values. But, without understanding the underlying code and how it maps data to marks, making such stylistic changes to a D3 visualization is out of reach for most viewers.

We introduce a tool that can automatically deconstruct existing D3 visualizations, to extract the data, the marks, and the mappings between them. We also provide a graphical restyling tool that lets users modify the visual attributes of the marks as well as the mappings from the data to these attributes. Our tool warns users when such modifications break an existing mapping so that users can make sure the restyled visualization conveys all of the data shown in the original visualization. Together our tools allow users to easily restyle existing D3 visualizations without examining the underlying code (Figure 1). We demonstrate the versatility of our tools by deconstructing and restyling a number of different types of D3 visualizations including bar charts, scatterplots, donut charts, choropleth maps and line charts.

RELATED WORK

Visualizations convey information by mapping data to the visual attributes of graphical marks. Bertin first described this mapping process in his seminal work *Semiology of Graphics* [1]. Visualization design tools like Excel, Google Spreadsheets, Polaris/Tableau [28], and Lyra [26] let users directly specify how data maps to mark attributes through a graphical user interface. Programmatic toolkits and libraries such as Infovis Toolkit [14], Prefuse [16], ProtoVis [6], ggplot2 [34], D3 [7] and Vega [32], let programmers specify these mappings in code at different levels of abstraction. While these tools facilitate the process of constructing visualizations, our work addresses the inverse problem; we deconstruct an input visualization to recover its data and marks as well as the mappings relating them. We focus on D3 visualizations because they are ubiquitous and are based on open Web standards (HTML, CSS, SVG, etc.) that allow inspection.

Bitmap images of charts are even more commonly available online than D3 visualizations. Researchers have applied computer vision techniques for recognizing the chart type (e.g. bar chart, pie chart, etc.) of such images. These techniques first extract high-level shape descriptors from the image and then classify the image based on these features [18, 23] Others have developed specialized image processing methods to extract the data-encoding marks from such chart images [36, 19, 18, 17, 35]. These methods analyze the edges in the image and apply heuristics based on knowledge of the chart type to identify the bars, pie slices, etc.

Savva et al.'s ReVision [27] combines such chart type classification with mark extraction and then further analyzes the marks to semi-automatically extract the underlying data from an input chart image. However, their complete pipeline is only designed to work for bar and pie charts, and the overall accuracy of their recovered data (38-53% depending on chart type) is limited by image resolution, noise, compression artifacts and classification errors. In contrast, because we focus on deconstructing D3 visualizations, we can directly access the data bound to each mark and our recovered data is always 100% accurate. Unlike ReVision, our approach also extracts mappings between the data and mark attributes.

Our restyling tool is inspired by recent work on transforming images of visualizations to explore new visual forms [10] or aid chart reading [20]. However, because these techniques operate on bitmap images of visualizations, they are limited to warping the entire image or adding graphical overlays onto the image. Because our restyling tool operates on D3 visualizations, it can directly modify mark attributes and thereby provide far more restyling control than the earlier techniques.

D3 PRELIMINARIES

D3 is a JavaScript library for building visualizations by manipulating the Document Object Model (DOM) of a webpage. The DOM is a hierarchical representation in which each node is a tagged element such as `<body>` or `<img>` from HTML or `<rect>`, `<circle>` or `<polygon>` from Scalable Vector Graphics (SVG). Most D3 visualizations are built using SVG because it provides a complete scene graph representation for 2D vector graphics. Therefore, we focus on SVG-based D3 visualizations in this work

SVG includes several types of nodes, including a root node `<svg>` that creates a viewport for the graphics, group nodes `<g>` that allow hierarchical grouping of sub-nodes to form an SVG tree, and mark generating nodes such as `<rect>`, `<circle>`, or `<polygon>`, which produce marks on-screen. Each SVG node establishes its own local coordinate system and provides a tranformation matrix from the parent space to the local space. The mark generating nodes also include a set of visual attributes such as *position, width, height, fill-color* and *stroke-width*, that define the appearance of the mark when it is rendered. D3 developers have shown that this set of SVG primitive nodes is expressive enough to represent a wide variety of visualizations.

A key feature of D3 is that it allows developers to bind input data to SVG elements as they construct and modify a visualization. This binding lets developers specify functional relationships between the data and visual attributes of the SVG elements. Consider the bar chart example in Figure 2a-b. Lines 4-6 of the code fragment create a set of `<rect>` nodes, each bound to one entry in the `items[]` data array. Line 7 appends each `<rect>` node as a child of the root `<svg>` node to form the SVG tree (Figure 2c). Lines 8-16 apply data-dependent functions to assign the *x-position, height* and *fill-color* visual attributes of the `<rect>` nodes, while lines 17-18 set the *width* and *stroke-width* of these nodes to constants. Thus, each fully specified `<rect>` node includes a `__data__` property holding the bound data as well as a set of visual attributes (Figure 2d).

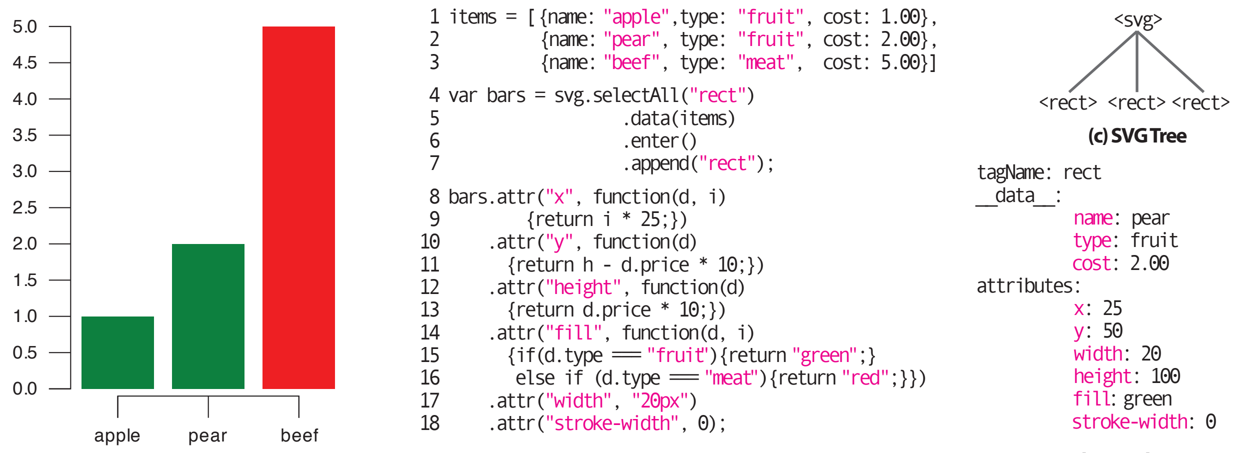
OVERVIEW

Our system includes two main tools; one for *deconstructing* a D3 visualization and one for *restyling* the visualization. The deconstruction tool analyzes the SVG representation of the visualization to recover its constituent data and marks, and to infer the mappings between them. The restyling tool uses the deconstructed information to help users change the style of a visualization while warning them when such style changes conflict with the data-encodings in the original visualization.

The data D3 binds to DOM elements is only accessible from within the JavaScript environment in which the visualization was constructed. So, in order to analyze a D3 visualization our tools require access to its environment. We built our tools as an extension for Google Chrome browser [11] because such extensions are allowed to inject code into the JavaScript environment of an existing webpage and can thereby access the bound data.

DECONSTRUCTION

To initiate deconstruction, users must right click anywhere within a D3 visualization. Our deconstruction tool listens for such click events, finds the DOM node containing the click and then walks up the DOM tree to find the root `<svg>` node of the visualization The resulting DOM sub-tree contains all of the SVG elements comprising the visualization and we analyze this sub-tree to deconstruct the visualization.



```
1 items = [{name: "apple",type: "fruit", cost: 1.00},
2          {name: "pear", type: "fruit", cost: 2.00},
3          {name: "beef", type: "meat",  cost: 5.00}]
4 var bars = svg.selectAll("rect")
5                  .data(items)
6                  .enter()
7                  .append("rect");
8 bars.attr("x", function(d, i)
9          {return i * 25;})
10    .attr("y", function(d)
11      {return h - d.price * 10;})
12    .attr("height", function(d)
13      {return d.price * 10;})
14    .attr("fill", function(d, i)
15      {if(d.type == "fruit"){return "green";}
16       else if (d.type == "meat"){return "red";}})
17    .attr("width", "20px")
18    .attr("stroke-width", 0);
```

```
tagName: rect
__data__:
          name: pear
          type: fruit
          cost: 2.00
attributes:
          x: 25
          y: 50
          width: 20
          height: 100
          fill: green
          stroke-width: 0
```

Figure 2. Example bar chart visualization (a) and D3 code fragment generating the rectangular marks representing the bars in the chart (b). The code creates three `<rect>` nodes and binds one data point to each of them (lines 4-7). It then sets the visual attributes of each using a combination of data-dependent functions and constants (lines 8-18). The code for generating the axes and labels is not shown. The resulting SVG tree (d) contains a single parent `<svg>` node and three children `<rect>` nodes (c). Each `<rect>` node includes a `__data__` property holding the data that is bound to it as well as a set of visual attributes (d).

Extracting Data and Marks

To extract the data and marks from the selected visualization, we traverse the SVG sub-tree in pre-order and enumerate all nodes that meet two conditions; (1) the node generates a graphical mark[1] and (2) the node is bound to data. These two conditions ensure that the resulting list of nodes captures all of the data-encoding marks in the visualization. In the case of our bar chart example, we enumerate a sequence of three `<rect>` nodes that form the bars of the chart.

Data extraction. Most D3 visualizations bind one data item to one mark generating SVG node. In such cases we extract the data by retrieving the `__data__` property from each node in our enumerated list. The retrieved data may consist of a simple typed value (e.g. a number, a string, etc.) or a more complex JavaScript object containing one or more data fields and a value for each field. In our bar chart example (Figure 2), each `<rect>` node is bound to a data object which contains the fields *name*, *type* and *cost*, and a value for each one. We refer to the set of data fields associated with a data object as its data *schema*. If the data consists of a simple value rather than an object, we treat its data type as the schema.

D3 visualizations can include multiple data schemas. For example, a grouped bar chart may contain one set of bars bound to a data schema for movie box office revenue, while another set of bars is bound to a schema for DVD revenue. In such cases, we group the data by schema as we extract it and generate one data table per schema. We extract 3 data tables for our bar chart example (Figure 3). The first table represents the `item[]` data bound to the `<rect>` nodes. The other two tables represent the string label data (e.g. “apple”, “pear”, “beef”) bound to the *x*-axis and the numeric label data (e.g. 0.0, 0.5, ... 5.0) bound to the *y*-axis. When D3 generates axes, it binds such label data to `<line>` nodes forming tick marks and `<text>` nodes for graphically displaying the labels. Note that Figure 3 does not show the third table.

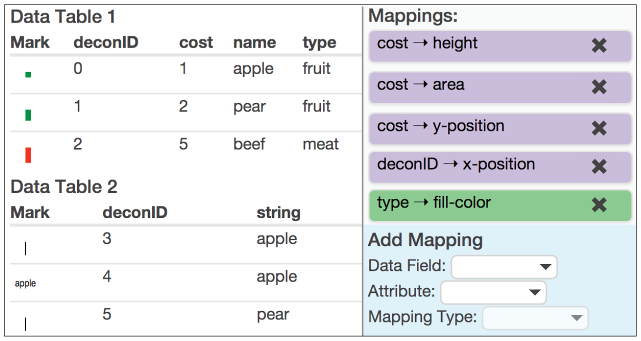
Some visualizations use the ordering in which data is bound to the elements to assign the visual attributes of the marks. For example, in line 13 our example bar chart sets the *x*-position of the bar based on the ordering index *i* in which the bar is constructed. As we extract the data we recover this ordering index by saving the pre-order tree-traversal index for each node as an additional *deconID* field in the data table. Although the pre-order traversal index may not exactly match the original data binding order *i*, there is usually a linear relationship between them. We exploit this linear relationship when we construct mappings between the data fields and visual mark attributes (see next Section on Mappings). Finally we keep a reference to the original SVG node in the visualization so that we can later modify and re-style the element.

Mark extraction. Next we extract a set of mark attributes from each SVG node in our enumerated list. These attributes fall into two categories:

- ***Appearance attributes*** determine the the visual look of the mark. Examples include *shape*, *fill-color*, *stroke-color*, *stroke-width*, *font-style* and *font-size*.
- ***Geometric attributes*** determine the spatial properties of the mark. Examples include *x-position*, *y-position*, *width*, *height*, and *area*.

The primary appearance attribute is *shape* as it defines the visual form of the mark. In the deconstruction stage we treat the tag name of the mark generating SVG node (e.g. `<rect>`, `<circle>`, etc.) as the *shape* attribute. Note that although some nodes such as `<polygon>` and `<path>` may represent many different shapes, since we only extract the tag name as the *shape* attribute our deconstruction tool can not differentiate between the different shapes that a `<polygon>` could encode (e.g. triangle, pentagon, hexagon, etc.). However, our restyling tool does allow users to map data to specific types of polygons (see Restyling Section).

[1]We consider any SVG node that produces visible marks on-screen as a mark generating node (e.g. `<rect>`, `<circle>`, `<line>`, `<polygon>`, `<path>`, `<text>`). In contrast nodes like `<svg>` and `<g>` set up and transform the view but do not produce marks.


Data Table 1

Mark	deconID	cost	name	type
🟩	0	1	apple	fruit
🟩	1	2	pear	fruit
🟥	2	5	beef	meat

Data Table 2

Mark	deconID	string
|	3	apple
apple	4	apple
|	5	pear

Mappings:
- cost → height ✖
- cost → area ✖
- cost → y-position ✖
- deconID → x-position ✖
- type → fill-color ✖

Add Mapping
Data Field:
Attribute:
Mapping Type:

Figure 3. Deconstructed data and restyling interface for the bar chart from Figure 2. Data Table 1 contains the data and marks for the bars. while Data Table 2 constains the data and marks for the *x*-axis tick marks and text labels. The Mappings Panel shows mappings extracted from the chart for Data Table 1, with linear mappings shown in purple and nominal mappings shown in green. Users can remove, change and add mappings through the Mappings and Add Mappings Panels.

All other appearance attributes directly correspond to visual attributes of the SVG node. Some of these attributes such as *fill-color*, *stroke-color*, and *stroke-width*, are defined for every mark generating node in SVG, while others such as *font-style* and *font-size* attributes are only defined for a subset of SVG nodes—e.g. the `<text>` node. As we analyze each SVG node we retrieve all of the appearance attributes that exist for it and leave the other appearance attributes as undefined.

In SVG, the representation of geometric information such as position, width, height and area can differ depending on the type of the node. For example the *position* of a `<circle>` node represents the location of its center while the *position* of a `<rect>` node represents the location of its upper left corner. However, SVG also provides access to a bounding box representation for every type of mark generating node. The geometric attributes of the bounding box are the same regardless of node type and there is usually a linear relationship between the geometric attributes of the bounding box and the corresponding geometric attributes of the underlying mark (e.g. the area/width/height of an ellipse is linearly related to the area/width/height of its bounding box). Therefore we treat the geometric attributes of the bounding box as the geometric attributes of the mark.

We use the center of the bounding box as the position of the mark and apply transformations with respect to this center point. We also transform the bounding boxes into the coordinate space of the root SVG node before extracting the geometric attributes so that the geometric information for every mark in the visualization is in the same global coordinate space. As we extract the mark attributes for each SVG node we store them with the corresponding row of the data table.

Handling line charts. While most D3 visualizations bind one data item to one mark generating SVG node, line charts typically bind an array of data to a single mark generating SVG `<path>` node which produces a polyline on screen. Conceptually each data item in the array is bound to the corresponding point in the polyline. To deconstruct such line charts, we split the data array into a sequence of individual data items and add each one to the data table. For each data item we also add a *line-id* field to the data table which stores the index of the item within the array. Similarly, we split the polyline into individual geometric points and set the *position* mark attribute of the corresponding data item to the coordinates of the point. We set the other geometric mark attributes, *width*, *height*, and *area*, to 0. Finally we set the remaining mark appearance attributes for each of these data items based on the attributes of the `<path>` node.

Extracting Mappings

Visualizations encode information by mapping data values to visual mark attributes. For example, our bar chart (Figure 2), sets the *x-position*, *height*, and *fill-color* of each bar based on functions of the underlying ordering index i, and the data fields *cost* and *type*. The first two functions (lines 8-13) are *linear mappings* between numeric data and numeric mark attributes (ordering index $i \to$ *x-position*, *cost* $\to$ *height*). The third function (lines 14-16) is a *categorical mapping* in which all data items of the same value map to the same mark attribute value (*type* $\to$ *fill-color*). Unlike a linear mapping, a categorical mapping does not require either the data field or mark attribute to be a numeric type. We recover both kinds of data-dependent mappings for each data table.

We first test for linear mappings between the numeric data fields and each numeric mark attribute by fitting a linear model using linear regression. If the resulting R^2 value for the model is equal to 1, the data fields are linearly mapped to the mark attribute and we save the line parameters (slopes and intercept) of the fitted model.

In visualizations, a single mark attribute sometimes represents a linear combination of multiple data fields. However, if any of the data fields in such a mapping are themselves linearly related, there is a simpler set of mappings each of which includes exactly one of these linearly related data fields. We recover this *parsimonious* set of linear mappings which use the fewest data fields necessary to explain the relationship between the data and a mark attribute using an iterative approach. We first check for mappings between single data fields and each mark attribute. If we find mappings for an attribute we remove it from further consideration. If we do not find a mapping for an attribute, we check for mappings between pairs of data fields and that attribute. We continue to iteratively add data fields in this manner until we find one or more mappings for the attribute or we exhaust the set of data fields in the data table. The resulting set of mappings for each attibute is its parsimonious set.

If we do not find any linear mapping for an attribute or the attribute is non-numeric, we test for categorical mappings between each data field and the attribute. Specifically, we check whether whether there is a bijective, one-to-one correspondence between the data field values and the mark attribute values by testing whether each unique data value corresponds to a unique mark attribute value.

Our approach often finds multiple mappings between different data fields and the same mark attribute. For example, if a data table includes two or more data fields that are linearly related to each other, and any of these fields is mapped to a mark attribute, our tool recovers a mapping between each of these fields and the attribute. Similarly, our tool may find multiple categorical mappings between different data fields and a single mark attribute. In both of these cases all of the extracted mappings are equally valid and can reveal unexpected correlations or structure in the data. Thus, we show all of these mappings to the user.

However our deconstruction tool may also find both a linear and a categorical mapping to the same mark attribute. Because linear mappings offer a simpler model of the relationship between a data field and a mark attribute, we automatically filter out categorical mappings in such cases. In the bar chart example, we initially recover a linear mapping *cost* $\rightarrow$ *y-position* as well as a categorical mapping *deconID* $\rightarrow$ *y-position*, but then filter out the more complex categorical mapping.

Figure 3 shows the mappings we recover for our bar chart example. Note that although our extracted data table does not recover the original ordering index *i*, it does find a linear mapping between the pre-order traversal index *deconID* and *x-position*. Clicking on a linear mapping shows two data values and their corresponding attribute values. Since linear mappings are fully specified by any two such correspondences, we show the minimum and maximum data values to reveal the full range of the correspondence. Clicking on a categorical mapping shows each unique data value and the corresponding attribute value.

Handling color attributes. Color attributes are special cases for the mapping process because they may be mapped either linearly or categorically to the underlying data in several different color spaces (e.g. RGB, HSL, LAB, etc.). For color mark attributes we test for linear mappings to each color channel independently in RGB space and in HSL space. For categorical mappings we consider the color triplet in RGB space as a single complex-typed value.

Splitting data tables. Users may sometimes wish to operate on a subset of rows in the data table. Our interface allows users to select a set of rows in a deconstructed table and generate a new table from the selection. For example, a D3 axis typically binds the same data to both the `<line>` node representing a tick mark and the `<text>` node representing a label. In order to compute mappings for these sets of nodes independently, the user can select all of the `<text>` nodes and place them in a separate data table.

RESTYLING

Our restyling tool lets users change the look of a visualization by manipulating mark attributes using three kinds of operations; (1) remove mapping, (2) change mapping, and (3) add mapping. These operations are accessible by clicking on different parts of the Mappings Panel in our interface (Figure 3). Each operation first updates the data table with new attribute values for the marks and then propagates the changes to the visualization by updating the SVG nodes corresponding to the mark. Figure 4 shows an example of how we can apply these three operations in a sequence of steps to convert our example bar chart from Figure 2 into a colored dot plot.

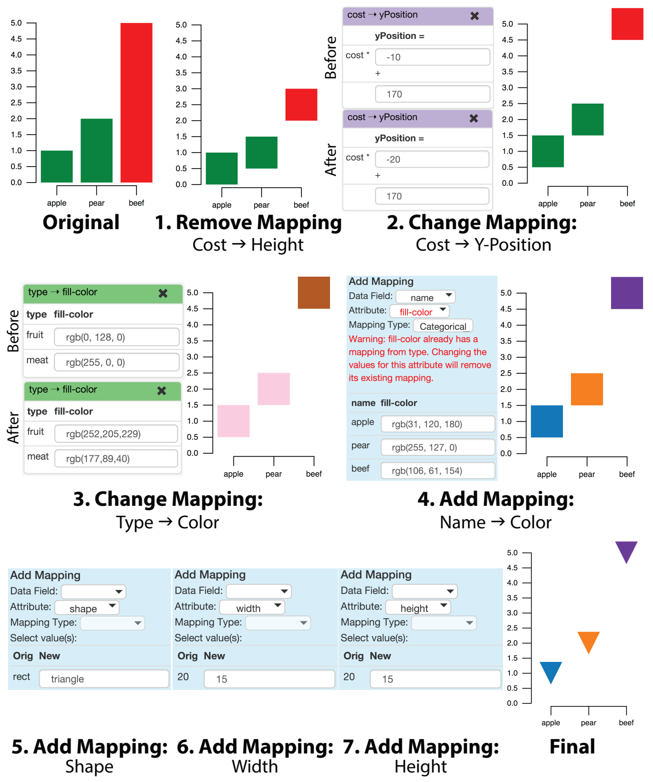


Figure 4. Restyling the example bar chart from Figure 2 using a sequence of 7 restyling operations. Step 1 removes the *cost* $\rightarrow$*height* mapping. Step 2 changes the *cost* $\rightarrow$*y-position* to raise the marks. Step 3 changes the *type* $\rightarrow$*fill-color* mapping to modify color or marks. Step 4 adds a mapping from *name* $\rightarrow$*fill-color* and to replace the mapping from step 4 despite warnings. Steps 5-7 add mappings that are not based on data values to modify the *shape, width* and *height* attributes of the marks.

Remove Mapping

Users can remove a mapping by clicking the 'X' button next to it in the Mappings Panel (Figure 3). Upon removal our tool updates the value of the mapped attribute to a constant value for every item in the data table. For linear mappings, our tool sets the constant to the attribute value of the mark with the minimum data value. For categorical mappings our tool sets it to the attribute value of the mark with the lowest *deconID*. In step 1 of our restyling example (Figure 4) we remove the *cost* $\rightarrow$ *height* mapping, and our tool scales the height of each bar to that of the bar with the lowest cost. Since our system transforms marks about the center of their bounding boxes, the two largest bars no longer touch the *x*-axis.

Change Mapping

Users can change any existing mapping by clicking on it in the Mappings Panel (Figure 3). For linear mappings users can specify new line parameters (slopes and intercept) for the mapping, which our restyling tool then applies to update the attribute values of every item in the data table. For categorical mappings users can specify the attribute value corresponding to any unique data value. Our tool then updates the attributes values for all data items that share this data value.

In step 2 of our restyling example (Figure 3) we change the linear *cost* $\rightarrow$ *y-position* mapping to re-position the rectangular marks so that their centers align with their corresponding costs on the *y*-axis. In step 3 we change the categorical *type* → *fill-color* mapping so that ‘fruit’ maps to ‘pink’ and ‘meat’ maps to ‘brown’.

Add Mapping

Users can add a new mapping by selecting three mapping properties – one or more data fields, a mark attribute and a mapping type – via drop-down menus in the Add Mapping Panel (Figure 3). The data field menu includes each of the extracted fields in the data schema as well as an option to choose ‘none’. The mark attribute menu includes all of the extracted appearance and geometric attributes. The mapping type menu includes linear and categorical types as well as an option to choose ’none’. As a user selects from these menus our interface further constrains the available options to ensure that the resulting mapping is valid. Specifically, it checks that linear mappings always map numeric data fields to a numeric mark attribute. For instance, if the user selects linear as the mapping type, our interface grays out and disables selection for all non-numeric data fields and mark attributes. Conversely, if the user selects a non-numeric data field or mark attribute, our interface disables the linear mapping type.

A key feature of the interface is that it warns users if they try to add a mapping to an attribute that already encodes a different data field. In such cases it is likely that the new mapping conflicts with the existing mapping and would change the information conveyed by the visualization. We warn users of such conflicts in two ways. First, in the mark attributes drop-down menu we highlight in red all attributes that are already part of an existing mapping. Second, if the user selects one of these red attributes we display a text warning explaining that adding the new mapping will invalidate the existing mapping. If the user proceeds to add the new mapping despite the warnings, our restyling tool removes the existing mapping before creating the new one.

After selecting the three mapping properties, users can specify how data values map to attribute values using the same interface as for the change mapping operation. Finally the newly created mapping is added to the list of mappings in the Mappings Panel.

In step 4 of our restyling example (Figure 4) we decide to re-color the marks using the *name* data field instead of *type*. To make this change we add a categorical *name* → *fill-color* mapping. But because the *type* → *fill-color* mapping already maps a data field to the *fill-color* attribute, our interface highlights *fill-color* in red in the drop-down menu and displays warning text as soon as we select it from the menu. Despite the warnings we continue to build the new mapping by specifying that ’apple’ maps to ’blue’, ’pear’ maps to ’orange’ and ’beef’ maps to ’purple’. Our tool then removes *type* → *fill-color* and adds *name* → *fill-color* to the Mappings Panel.

The data-encoding marks in a visualization often include some attributes that do not encode data but significantly effect the look of the visualization. In our restyling example, the *shape* attribute of the marks does not encode data. Users can modify such attributes values by selecting the attribute and setting the data field to ’none’ in the Add Mapping Panel. Our interface then displays the set of unique attribute values that appear in the visualization and users can interactively specify new values for them. In step 5 of our restyling example (Figure 4) we change the shape of the marks from ’rect’ to ’triangle’. Similarly in steps 6 and 7 we change the widths and heights of the marks from 20 to 15.

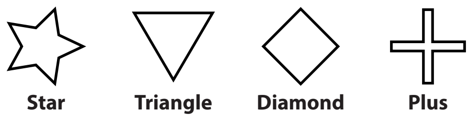


Figure 5. Symbolic shapes supported by our restyling tool.

Updating Marks in the Visualization

As users perform restyling operations our tool updates mark attributes in the data table and then updates corresponding SVG nodes in the DOM of the original visualization. For each updated mark the restyling tool creates a new SVG node with its type specified by the *shape* attribute. Although the *shape* attribute is usually an SVG node type, our restyling tool also supports a set of symbolic shapes (star, diamond, triangle, plus) that are all represented as SVG `<polygon>` nodes with pre-defined geometry (Figure 5). If the *shape* attribute is one of these symbolic shapes our tool converts it into the corresponding pre-defined `<polygon>` node.

Our restyling tool also transforms all geometric attributes from the global coordinate space into the local space of the original SVG node representing mark. It then converts these bounding box based geometric attributes into attributes appropriate for the node type (e.g. *position* is upper-left corner for `<rect>` nodes, but center for `<circle>` nodes). Our tool then sets the geometric and appearance attributes of the new SVG node and binds data as well as event handlers from the original node to the new node. Finally, our tool replaces the original SVG node with the newly created node.

RESULTS

As shown in Figures 1, 6 and 7 we have used our system to deconstruct and restyle a number of D3 visualizations from a variety of sources. These examples incorporate modifications of many different mark attributes including *fill-color* (Figures 1, 6a–c, 6e, 6f, 7g–i), *stroke-color* (Figure 6b, 6e, 6f, 7g), *stroke-width* (Figures 6e, 6f), *font-style* (Figure 6a, 6b, 6f, 7g), *shape* (Figures 1, 6a, 6c, 6d, 6f, 7g, 7i), *position* (Figures 6a, 6d, 7g-i), as well as *width* and *height* (Figures 6a, 6b, 6d, 6f, 7g-i). All of our restyled examples represent the same data as the original visualization, but some examples re-map data fields to different mark attributes (Figures 6c, 6e–f, 7g, 7i), while other examples change the parameters of the existing mappings (Figures 1, 6a, 6b,6d–f, 7g, 7h). Many examples also modify mark attributes that do not encode data but affect the look of the visualization (Figures 1, 6a, 6b, 6e, 6f, 7g, 7i.

New York Times Chart (Figure 1). In the original visualization U.S. companies are represented by circles that show each company’s effective tax rate using the *fill-color* and *x-position* attributes. Each company’s market capitalization is mapped to the *area* attribute. In our restyled result we change the *fill-color* attribute to color marks using a red to blue diverging color scale [9]. We also change each mark’s *shape*

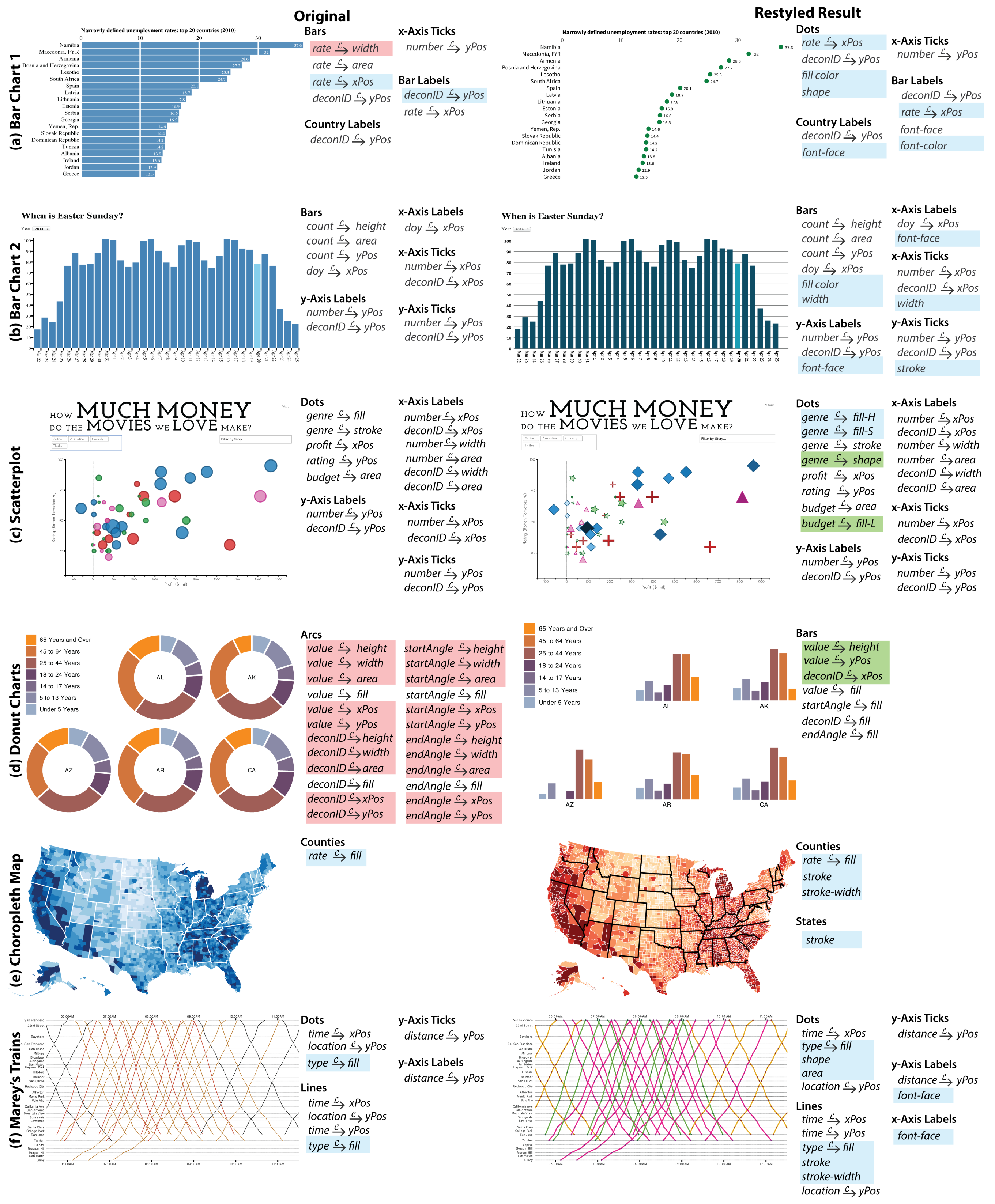


Figure 6. A variety of D3 visualizations collected from the Web, before (left column) and after (right column) deconstruction and restyling with our tools. Deconstructed mappings are shown to the right of the original visualization, and the mappings after restyling are shown to the right of the restyled result along with any unmapped attributes that were changed in the restyling. We use the $\xrightarrow{L}$ and $\xrightarrow{C}$ notation to indicate linear and categorical mappings respectively. Mappings removed and added during restyling are highlighted in red and and green respectively. Changed mappings and attributes are highlighted in blue.

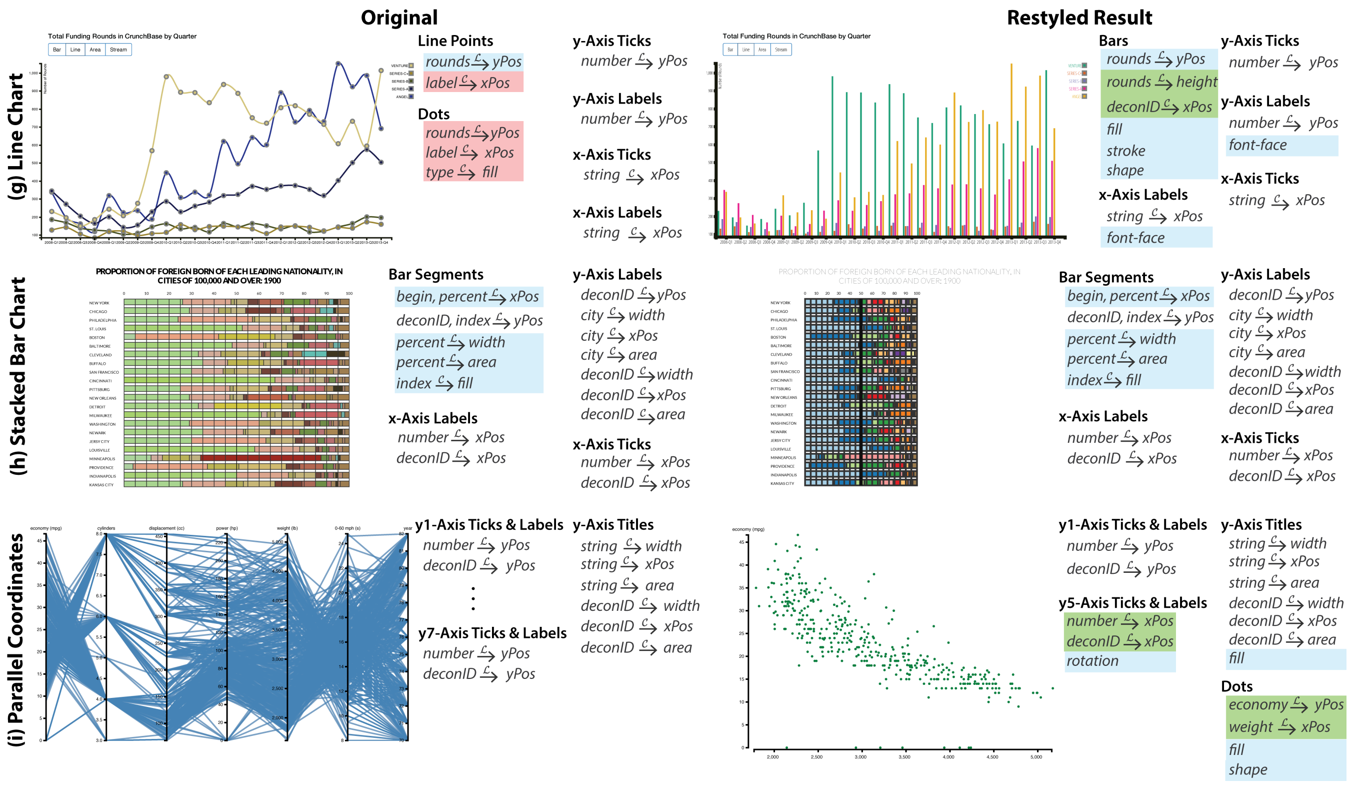


Figure 7. A variety of D3 visualizations collected from the Web, before (left column) and after (right column) deconstruction and restyling with our tools. Deconstructed mappings are shown to the right of the original visualization, and the mappings after restyling are shown to the right of the restyled result along with any unmapped attributes that were changed in the restyling. We use the $\xrightarrow{L}$ and $\xrightarrow{C}$ notation to indicate linear and categorical mappings respectively. Mappings removed and added during restyling are highlighted in red and green respectively. Changed mappings and attributes are highlighted in blue.

attribute to represent the companies using stars but we leave the *market capitalization* $\rightarrow$ *area* mapping unchanged. Original visualization from the New York Times [29].

Bar Chart 1 (Figure 6a). The original bar chart shows the 20 countries with the highest unemployment rates sorted by unemployment rate along the *y*-axis. We restyle the chart into a dot plot using the approach outlined in Figure 4. Original visualization by Leon du Toit [13].

Bar Chart 2 (Figure 6b). The original chart shows the dates Easter Sunday falls on with frequency of each day represented by bar height. The chart highlights one bar in light blue representing the date for Easter on the year selected in a drop-down menu. We restyle the bar chart to resemble the visual design of charts published by the Economist by changing the values of several unmapped mark attributes. Original visualization by Chris Pudney [24].

Scatterplot (Figure 6c). The original scatterplot depicts the Rotten Tomatoes data including the *rating, profit, genre,* and *budget* for popular movies using several different mark attributes. We restyle the chart by adding a redundant categorical mapping *genre* $\rightarrow$ *shape* to visually differentiate the genres. We also add a redundant linear mapping *budget* $\rightarrow$ *fill-color-L* so that the lightness channel of each mark's color depicts the movie budget while the original mapping of *genre* to the hue and saturation of the *fill-color* is unchanged. Original visualization by Jim Vallandingham [30].

Donut Charts (Figure 6d). Each donut chart in the original visualization represents the percentage of the population of a U.S. state by age group. We restyle each donut chart into a bar chart by removing the mappings to the attributes of the arcs and then changing the *shape* attribute of each arc to a rectangle. Finally we create new mappings for the *height*, *x-position* and *y-position* of each bar. Original visualization by Michael Bostock [4].

Choropleth Map (Figure 6e). The original choropleth map shows U.S. unemployment by county using the fill-color of each map region. We restyle the coloring of the map by changing the *rate* $\rightarrow$ *fill-color* mapping for counties to a multi-hue yellow to red scale [9]. We also update the *stroke-color* and *stroke-width* attributes for map regions to emphasize the borders between states and counties. Original is a modified version of a visualization by Michael Bostock [3].

Marey's Trains (Figure 6f). The original visualization is a recreation of Marey's train schedule chart. Each line represents a train and the original visualization maps the *time*, *location* and *train type* (regular, limited service, baby bullet) data fields for each train to the *x-position*, *y-position* and *fill-color* attributes respectively. We restyle the visualization, changing the mappings to *fill-color* and as well as the values of several unmapped attributes in order to emphasize the different types of trains as well as their stops times and locations. Original visualization by Michael Bostock [5].

Line Chart (Figure 7g) The original line chart shows the number of tech company funding rounds by quarter, with each line representing a different type of investment. We change the line chart into a grouped bar chart by first changing the points deconstructed from each line into rectangle shapes. We then add mappings from the *funding rounds* data field to the *height* and *y-position* attributes of the bars. We arrange the bars into groups using a *deconID* $\rightarrow$ *x-position* mapping. We change the mapping *investment type* $\rightarrow$ *fill-color* to change the coloring of the bars. Finally, we modify the unmapped *font-face* attribute. Original visualization by Steven Hall [15].

Stacked Bar Chart (Figure 7h). The original visualization shows the proportion of immigrants by nationality in large U.S. cities around 1900. We restyle the visualization to a narrower aspect ratio by modifying the *begin, percent* $\rightarrow$ *x-position* and *percent* $\rightarrow$ *width* mappings. The original bar colors are not colorblind safe, so we also change the *nationality* $\rightarrow$ *fill-color* mapping to a qualitative color scale that is perceptually distinguishable for protanopes – red-green colorblind individuals [9]. Original visualization by Jim Vallandingham [31].

Parallel Coordinates (Figure 7i). The original parallel coordinates chart depicts seven properties (*economy*, number of *cylinders*, *displacement*, *power*, *weight*, *acceleration*, and *year* of release) of different kinds of cars. We restyle the chart into a scatterplot that shows the relationship between two of these properties – *economy* and *weight*. Specifically we change the *shape* attribute of each line mark to a circular dot. We then add *economy* $\rightarrow$ *y-position* and *weight* $\rightarrow$ *x-position* mappings for each dot. Finally, we remove all axes except the first and fifth which represent weight and economy. We then generate a new *x*-axis by changing the *rotation* attribute of the weight axis (fifth *y*-axis) marks. Original visualization by Jason Davies [12].

Our deconstruction and restyling tools allow rapid iteration on the visual design of a D3 visualization without having to understand its implementation. While our tools do assume that users understand that charts are comprised of mappings between data and mark attributes, users do not need experience with D3 or JavaScript to use our tools effectively. In practice we have found that even experienced D3 developers in our lab prefer to use our tools to explore the design space of a visualization to avoid the overhead of modifying code.

Limitations

While our tools can successfully deconstruct and restyle many different types of D3 visualizations, they have some limitations. Our deconstruction tool currently focuses on extracting linear and categorical mappings because they are most common. However, visualizations can include more complex functional mappings. For example, some D3 visualizations map numeric data fields to colors using a piecewise linear mapping that interpolates between pairs of colors in a sequential color scale [9]. Similarly, logarithmic, exponential and polynomial transforms are sometimes used to map data to positions of marks (e.g. log scale in a scatterplot). Our tool cannot deconstruct such non-linear functional mappings.

Our tools parameterize the geometric attributes of marks using their bounding boxes. While this approach provides control over the position, size and area of marks independent of their shape, it prevents our tools from manipulating the vertices and angles of shapes. In donut charts such as the one in Figure 6d, our deconstruction tool cannot recover the mapping from data to arc angle. However, our tool does recover the data bound to each mark and we can therefore create a new mapping from the data to rectangular marks to convert the donut chart into a bar chart.

Our restyling tool is also unable to add new SVG nodes to the visualization, which limits the kinds of visual redesigns it can produce. For example, in Figure 6d our tool cannot add text labels to the bar charts indicating the value of each bar, because no such `<text>` node existed in the original visualization. With the ability to add new SVG nodes bound to an existing data schema, our restyling tool could generate text labels displaying the percentage represented by each bar.

After restyling a visualization using our tools, the interaction and animation effects of the original may no longer be functional in the restyled version. Although we attach event listeners and data from the original SVG nodes in the to new nodes in the restyled visualization, these listeners often contain D3 code that depends on specific attributes values of the original node (e.g. shape, width or height). As we modify these attributes the interaction and animation code may no longer function correctly.

CONCLUSION AND FUTURE WORK

We have presented a pair of tools for deconstructing and restyling D3 visualizations. Our tools empower viewers to modify the visual look of existing D3 visualizations without having to understand or examine the underlying code. We believe there are several open directions for future work.

Handling a wider range of functional mappings. While linear and categorical mappings between data and mark attributes are most common, some visualizations include more complex mappings (e.g. logarithmic, polynomial, piecewise linear). It may be possible to automatically discover such mappings using data mining techniques. Similarly it should be possible to extend our restyling tool to let users define complex mappings between the data and mark attributes.

Visualization style transfer. Style transfer is the problem of applying the visual style of an exemplar visualization to a target visualization. The challenge is to ensure that the target visualization conveys the same data it conveyed before the style transfer. By analyzing the deconstructed mappings in the original exemplar and target visualizations it may be possible to perform the style transfer while preserving informational content of the original target.

Acknowledgments

This work was partially supported by the NSF Graduate Research Fellowship DGE 1106400.

REFERENCES

1. Bertin, J. *Semiology of graphics: Diagrams, networks, maps*. University of Wisconsin press, 1983.

2. Bostock, M. D3 Gallery. `https://github.com/mbostock/d3/wiki/Gallery`. Retrieved April 2014.

3. Bostock, M. Choropleth. `http://bl.ocks.org/mbostock/4060606/`, Nov. 2012. Retrieved April 2014.

4. Bostock, M. Donut Multiples. `http://bl.ocks.org/mbostock/3888852/`, Oct. 2012. Retrieved April 2014.

5. Bostock, M. Marey's Trains. `http://bl.ocks.org/mbostock/5544008/`, May 2013. Retrieved April 2014.

6. Bostock, M., and Heer, J. Protovis: A graphical toolkit for visualization. *IEEE TVCG 15*, 6 (2009), 1121–1128.

7. Bostock, M., Ogievetsky, V., and Heer, J. D^3 data-driven documents. *IEEE TVCG 17*, 12 (2011), 2301–2309.

8. Boston Globe. 'Women' a central theme in Menino's speech. `http://www.bostonglobe.com/2013/01/30/mai/yWQCjhK7lyBaqgqFrakr1M/story.html`, May 2013. Retrieved April 2014.

9. Brewer, C. ColorBrewer: Color Advice for Maps. `http://colorbrewer2.org/`. Retrieved April 2014.

10. Brosz, J., Nacenta, M. A., Pusch, R., Carpendale, S., and Hurter, C. Transmogrification: Causal manipulation of visualizations. In *Proc. of UIST*, ACM (2013), 97–106.

11. Chromium. `https://developer.chrome.com/extensions/getstarted`. Retrieved April 2014.

12. Davies, J. Parallel Coordinates. `http://bl.ocks.org/jasondavies/1341281`, Nov. 2011. Retrieved July 2014.

13. du Toit, L. Unemployment ranked with horizontal bars. `http://bl.ocks.org/leondutoit/6436923/`, Sept. 2013. Retrieved April 2014.

14. Fekete, J. The infovis toolkit. In *INFOVIS* (2004), 167–174.

15. Hall, S. Multi Chart — delimited. `http://projects.delimited.io/experiments/multi-series/multi-chart.html`. Retrieved April 2014.

16. Heer, J., Card, S. K., and Landay, J. A. Prefuse: A toolkit for interactive information visualization. In *Proc. of SIGCHI* (2005), 421–430.

17. Huang, W., Liu, R., and Tan, C. L. Extraction of vectorized graphical information from scientific chart images. In *Proc. of ICDAR*, vol. 1, IEEE (2007), 521–525.

18. Huang, W., and Tan, C. L. A system for understanding imaged infographics and its applications. In *Proc. of DocEng*, ACM (2007), 9–18.

19. Huang, W., Tan, C. L., and Leow, W. K. Model-based chart image recognition. In *Graphics Recognition*. Springer, 2004, 87–99.

20. Kong, N., and Agrawala, M. Graphical overlays: Using layered elements to aid chart reading. *IEEE TVCG 18*, 12 (2012), 2631–2638.

21. MTV. `http://vma-twittertracker.mtv.com/live/`. Retrieved April 2014.

22. New York Times. Across U.S. Companies, Tax Rates Vary Greatly. `http://www.nytimes.com/interactive/2013/05/25/sunday-review/corporate-taxes.html`, May 2013. Retrieved April 2014.

23. Prasad, V. S. N., Siddiquie, B., Golbeck, J., and Davis, L. Classifying computer generated charts. In *Proc. of CBMI*, IEEE (2007), 85–92.

24. Pudney, C. Easter Sunday. `http://bl.ocks.org/cpudney/2248382/`, Mar. 2012. Retrieved April 2014.

25. Ros, I. `https://bl.ocksplorer.org`. Retrieved April 2014.

26. Satyanarayan, A., and Heer, J. Lyra: An interactive visualization design environment. In *EuroVis* (2014), To appear.

27. Savva, M., Kong, N., Chhajta, A., Fei-Fei, L., Agrawala, M., and Heer, J. Revision: A utomated classification, analysis and redesign of chart images. In *Proc. of UIST* (2011), 393–402.

28. Stolte, C., Tang, D., and Hanrahan, P. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE TVCG 8*, 1 (2002), 52–65.

29. Times, T. N. Y. Across u.s. companies, tax varies greatly. `http://www.nytimes.com/interactive/2013/05/25/sunday-review/corporate-taxes.html/`, May 2013. Retrieved April 2014.

30. Vallandingham, J. How Much Money Do The Movies We Love Make? `http://vallandingham.me/vis/movie/`. Retrieved April 2014.

31. Vallandingham, J. Nationality by City. `http://vallandingham.me/vis/nationality_by_city.html`. Retrieved April 2014.

32. Vega. `http://trifacta.github.io/vega/`. Retrieved April 2014.

33. Viau, C. The Big List of D3.js Examples. `http://christopheviau.com/d3list/`. Retrieved April 2014.

34. Wickham, H. *ggplot2: elegant graphics for data analysis*. Springer, 2009.

35. Yang, L., Huang, W., and Tan, C. L. Semi-automatic ground truth generation for chart image recognition. In *Document Analysis Systems VII*. Springer, 2006, 324–335.

36. Zhou, Y. P., and Tan, C. L. Hough technique for bar charts detection and recognition in document images. In *Proc. of ICIP*, vol. 2, IEEE (2000), 605–608.