Protovis: A Graphical Toolkit for Visualization

Michael Bostock and Jeffrey Heer

Abstract—Despite myriad tools for visualizing data, there remains a gap between the notational efficiency of high-level visualization systems and the expressiveness and accessibility of low-level graphical systems. Powerful visualization systems may be inflexible or impose abstractions foreign to visual thinking, while graphical systems such as rendering APIs and vector-based drawing programs are tedious for complex work. We argue that an easy-to-use graphical system tailored for visualization is needed. In response, we contribute Protovis, an extensible toolkit for constructing visualizations by composing simple graphical primitives. In Protovis, designers specify visualizations as a hierarchy of marks with visual properties defined as functions of data. This representation achieves a level of *expressiveness* comparable to low-level graphics systems, while improving *efficiency*—the effort required to specify a visualization—and *accessibility*—the effort required to learn and modify the representation. We substantiate this claim through a diverse collection of examples and comparative analysis with popular visualization tools.

Index Terms—Information visualization, user interfaces, toolkits, 2D graphics.

1 Introduction

A popular approach to visualization is to import data into charting software, specify a desired chart type, and then tweak visual parameters as needed to produce the final graphic. Modern charting software may support a dozen or more chart types, such as pie and line, while supporting numerous customizable visual parameters, such as color and font. As noted by Wilkinson [37], this approach is especially popular in user interfaces, where often a chart can be produced in a few clicks.

Chart typologies are successful because they are quick and easy to use, but suffer simultaneously because they are highly restrictive. Many visualizations cannot be made simply because they are not one of the supported types. In addition, despite customization, designers may be unable to control the precise graphical output. A typological grammar limits the space of possible visualizations, as compared to what is possible in more general graphical systems.

As a result, designers may resort to vector-based drawing programs to realize their intent [31]. This is unfortunate; while such programs offer the utmost flexibility, they are not tailored for visualization. Drawing vector graphics by hand is time-consuming and error-prone, and even with the ability to import or generate simple graphics from data, the process often cannot support interaction and live data.

High-level chart types and low-level vector drawing represent two extremes, but in practice designers choose between many different systems, considering *expressiveness* ("Can I build it?"), *efficiency* ("How long will it take?") and *accessibility* ("Do I know how?"). The choice of tool affects the resulting work, as it biases designers towards visualizations that are easier to produce in the given tool. As Maslow [23] famously quipped, "I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail."

The interfaces of these tools, as well as their underlying models, vary substantially. Heer & Agrawala [15] noted the difficulty in identifying common design patterns within existing visualization tools, and consequently the high cost for users to learn and evaluate unfamiliar systems. A tool that is both expressive and efficient, if difficult to learn, may be inaccessible to users and of diminished practical value.

Despite the diversity of existing tools, surveyed in Section 2, we argue that there is still a gap between low-level graphical systems and high-level visualization systems. Many direct manipulation graphical systems are easy to learn but tedious for complex work, while powerful visualization systems can be intimidating to novices or inflexible.

Moreover, the abstractions used by visualization systems may be foreign to designers. While vector graphics editors allow designers to think concretely in terms of graphical marks, most expressive visualization tools make use of abstract specifications of data processing and visual encoding operators. Such systems require that designers translate their intended visual design into toolkit abstractions, often hindering accessibility.

In response, we contribute Protovis, an embedded domain-specific language [19] for constructing visualizations by composing simple graphical marks such as bars, lines and labels. In Protovis, designers specify visualizations as a hierarchy of marks with visual properties defined as functions of data. Inheritance of properties across composed marks—similar to cascading of style sheets used in web design—enables concise visualization definitions with a large expressive range and a minimum of intervening abstractions. Protovis is implemented in JavaScript, with rendering support for HTML 5 canvas, SVG, and Flash.

To evaluate Protovis, we built example applications demonstrating the toolkit's expressiveness and notational efficiency. We use these examples as points of comparison with Processing and Flare, two popular visualization tools. To assess accessibility, we present a comparative analysis using the Cognitive Dimensions of Notation framework [13] and share feedback from designers using Protovis.

2 Related Work

For the purpose of comparison we divide tools used to visualize data into two categories: visualization systems based on high-level abstractions tailored to data visualization, and graphical systems using only low-level graphical primitives. This dichotomy is not strict; indeed, Protovis is intended to fall somewhere in-between.

2.1 Graphical Systems

Formal visualization systems are not required to construct visualizations; with enough effort any graphical tool can be used. Vector-based drawing programs such as Adobe Illustrator are popular, especially for print. One benefit of these systems is the close cognitive mapping between the representations employed by the tool and the desired result: designers directly manipulate graphical marks to create and customize their visualization. This improves accessibility and reduces the gulf of execution [25]—the gap between designer's goals and the actions needed to attain them.

For total control, as well as to support interaction and live data, any number of low-level rendering APIs are available, such as OpenGL, Java2D, and Processing [26]. Processing was designed to be accessible to new users and non-programmers, to "teach fundamentals of

• *The authors are with the Computer Science Department of Stanford University, Stanford, CA 94305.*
E-mail: {mbostock, jheer}@cs.stanford.edu.

Manuscript received 31 March 2009; accepted 27 July 2009; posted online 11 October 2009; mailed on 5 October 2009.
For information on obtaining reprints of this article, please send email to: tvcg@computer.org .

computer programming within a visual context." Although these libraries are general purpose, they typically support only imperative methods for rendering graphical primitives such as ellipses and polygons. Higher-level tools such as Flash and Piccolo [1] further provide a scene graph abstraction to simplify tasks such as interaction and animation. Still, without any visualization abstractions, the construction of even simple charts is tedious.

2.2 Visualization Systems

Visualization systems are tools designed for the explicit purpose of data visualization, employing abstractions and mathematical models suited to this task. Such tools also commonly support data management, layout algorithms, interaction, and animation.

2.2.1 Consumer Software

By far the most widely-used visualization tools are those integrated into consumer software, often spreadsheet applications, such as Microsoft Excel and Google Spreadsheets. Although features vary, the underlying representation in these applications is uniform: a chart typology. A user simply selects the cells of data to visualize and then picks the desired chart type. The user may further customize the chart configuration to make minor adjustments to the chart's appearance. Some research systems, including IBM's Many-Eyes [33], fit this model.

Despite the shortcomings noted by Wilkinson [37] and Tufte [31] (not to mention questionable default chart configurations [11]), these tools must be recognized for their success in user adoption: they are easy to use, provided the user's needs are immediately satisfied by the built-in types. Creating a chart involves only a few quick actions; the process is more selection than creation.

The main drawback of this approach is that it requires a small, closed system. If the desired chart type is not supported, or the desired visual parameter is not exposed in the interface, no recourse is available to the user and either the visualization design must be compromised or another tool adopted. Given the high cost of switching tools, and the iterative nature of visualization design [6], frequent compromise is likely.

2.2.2 Analytical and Exploratory Tools

A number of tools have originated in the visualization research community, establishing theoretical underpinnings and providing richer options for visual data exploration. Tableau and its predecessor Polaris [29] integrate data manipulation with visualization, automatically deriving database queries from the visual specification. Wilkinson's Grammar of Graphics [37] is an elegant language for specifying visualizations as statistical graphs, "shunning chart typology" and offering greater flexibility. These systems both benefit from metadata, for example choosing appropriate default visual encodings for ordinal versus quantitative fields [3, 22].

Despite the expressive power of these tools relative to chart typologies, control over graphical output is still limited, making them less compelling for presentation and often unsuited for the design of novel, customized visualizations. These are closed systems; it may not be possible for the designer to customize all visual aspects if desired or introduce new forms of visual encoding. In addition, the complexity of the underlying model may be a barrier to entry for new users, due to a steep learning curve. Although high-level abstractions allow concise specifications,[1] they may appear magical if the user does not fully understand how the specification translates to the resulting visualization. This lack of understanding worsens the gulf of execution, as the necessary actions to correct the specification may be unclear.

2.2.3 Programming Toolkits

Programming toolkits are popular for presenting live data or allowing user interaction. Many support only a limited number of chart types, such as the Google Chart API, JFreeChart and PlotKit. Such toolkits offer similar trade-offs to the simple facilities built into consumer software, as discussed previously.

More expressive visualization toolkits include the InfoVis Toolkit [10], Improvise [35], and the Prefuse and Flare toolkits [17, 12]. (See [15] for a more thorough list of existing toolkits.) Each toolkit provides an integrated data management framework coupled with visualization and interaction components. The InfoVis Toolkit and Improvise provide a collection of visualization "widgets" that encapsulate visualizations into monolithic units. Such systems can be extended by creating new components from scratch or subclassing existing components. As a result the model of these toolkits is quite similar to the aforementioned chart typologies, and inherit many of the corresponding limitations: customized visualizations may require significant software engineering.

In contrast, visualizations in Prefuse and Flare are defined over a collection of parameterized visual objects associated with data. Following the data state model of Chi et al. [7], designers determine the properties of these visual objects (e.g., position, shape, color) by specifying a series of configurable operators that perform common actions such as layout and color encoding. By composing a visualization from fine-grained operators, Prefuse and Flare allow the construction of custom visualizations. Developers can further extend the system by defining new operators and visual primitives. To use these tools effectively developers must become steeped in the workings of the toolkit, including the library of provided operators and the stack of abstractions (e.g., axes, scales, visual objects, and renderers). While these tools can simplify many hard visualization tasks, they may also make easy tasks unnecessarily complex.

The web offers many possibilities for system architectures, ranging from thin clients running natively in the browser [9, 20] to "fat servers" where most computation occurs on the server, and hybrid approaches in-between [2, 38]. Our approach is partly agnostic to system architecture: while our initial implementation uses a thin client approach, the declarative specification is portable to other rendering engines (e.g., Java 2D, Flash), and data transformations and parts of the display could potentially be performed on the server. Our initial focus is only to improve the language designers use to specify visualizations; however, future research could determine whether this declarative approach allows optimization of the visualization pipeline, for example through lazy evaluation of visual properties [24] with large datasets.

3 Design

In designing Protovis, our goal was not to replace existing systems, but rather to support a missing point in the design space of visualization tools. Both charting software and analytical tools such as Tableau are successful in practice, but their expressiveness is limited. We wanted to provide tools that enable more low-level control of the design. Unlike existing toolkits primarily suited to software engineers, we created Protovis to make interactive visualization more accessible to web and interaction designers. Thus we attempted to minimize the number of intervening abstractions, allowing designers to focus on the composition of data-representative graphic elements using a notation that is concise and easy to learn.

Protovis uses a simple grammar of graphical primitives to compose visualizations. These primitives are called **marks**, and include familiar elements such as bars, lines and labels. Although a bar *mark* may be used to construct a bar *chart*, marks know nothing about charts; it is only through their specification and composition that charts are produced. These building blocks permit many combinatorial possibilities.

Marks are associated with **data**: a mark is generated once per associated datum, mapping the datum to visual **properties** such as position and color. Thus, a single mark specification represents a *set* of visual elements that share the same data and visual encoding. The type of mark defines the names of properties and their meaning. A property may be static, ignoring the associated datum and returning a constant; or, it may be dynamic, derived from the associated datum or index. Such dynamic encodings can be specified succinctly using anonymous functions, as shown in Figures 2 & 4. Special properties called **event**

[1] In ggplot2, an implementation of Wilkinson's Grammar in R, an example grouped bar chart is specified as `ggplot(diamonds, aes(x=clarity, fill=cut)) + geom_bar(position="dodge")` [36].

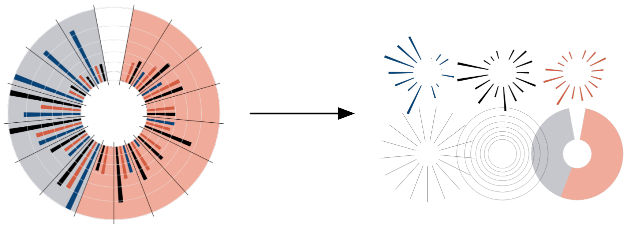
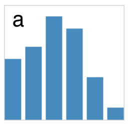


Fig. 1. Decomposing a visualization into marks.

```
new pv.Panel().canvas("fig3a")
  .add(pv.Bar)
    .data([1, 1.2, 1.7, 1.5, .7, .2])
    .bottom(0).width(20)
    .height(function(d) d * 80)
    .left(function() this.index * 25)
  .root.render();
```



```
new pv.Panel().canvas("fig3b")
  .data([[1, 1.2, 1.7, 1.5, .7],
         [.5, 1, .8, 1.1, 1.3],
         [.2, .5, .8, .9, 1]])
  .add(pv.Area)
    .data(function(d) d)
    .fillStyle(pv.Colors.category19.parent)
    .bottom(function() let (c = this.cousin())
        c ? (c.bottom + c.height) : 0)
    .height(function(d) d * 40)
    .left(function() this.index * 35)
  .root.render();
```

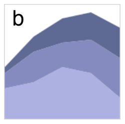


Fig. 2. Specifying two simple charts. (a) Bar. (b) Stacked area.

handlers can be registered to add interactivity.

Protovis uses **inheritance** to simplify the specification of related marks: a new mark can be derived from an existing mark, inheriting its properties. The new mark can then override properties to specify new behavior, potentially in terms of the old behavior. In this way, the old mark serves as the *prototype* for the new mark. Prototypal inheritance is familiar to web developers, as it is a feature of the JavaScript language and Cascading Style Sheets. It is also similar to Prefuse's *Cascaded Table* design pattern [15]. Most mark types share the same basic properties for consistency and to facilitate inheritance.

Marks may have associated **anchors**, which are named positions inside or nearby. Anchors can be used to position related marks, such as labels for grid lines. Likewise, **panels** can be used to offset positions and to replicate marks in small multiple displays [30]. An example of these features working together is shown in Figure 4. The specification of marks, properties and panels is detailed in Subsections 3.1-3.3.

Data can be imported using JavaScript Object Notation [21], which allows integration with many existing web services. Data can be atomic (e.g., numbers) or hierarchical (e.g., nested arrays, objects). **Data transformations** are performed using JavaScript language features and additional methods provided by Protovis, described in Subsection 3.4. Alternatively, transformations can be pushed to the server.

Finally, Protovis includes additional features to simplify the specification of marks, such as date formatting, scale transformations, gradients and color palettes. These features are not discussed, but documentation is online at `protovis.org`.

3.1 Marks

The first task when using Protovis is to decompose the desired visualization into its constituent marks, as in Figure 1. A bar chart might consist of one or more bars, a rule, and a label; a pie chart might include only a wedge. In this way, we divide the problem of constructing a complex visualization into a series of smaller, easier problems.

Once the initial set of marks is decided—it is always possible to add or remove marks later—the remaining task is to define the properties of each mark. Of course, this is the crux of visualization, so it is critical that the built-in mark types and properties are intuitive and useful.

3.1.1 Properties

Properties control how marks are rendered by mapping a given datum to a named rendering parameter. For example, the bar in Figure 2 is defined in terms of its left margin, bottom margin, width and height.

As marks are generated, Protovis internally builds a **scene graph** recording the computed property values for each. The scene graph serves two purposes. First, it facilitates the definition of related marks through *property chaining*, where one mark's properties are defined in terms of another's. The property value is retrieved from the scene graph rather than recomputed. It is also needed if property functions depend on transient state, such as the current time or random variables. Second, the scene graph supports interaction and animation.

Property functions can access the scene graph through the `this` object, as well as globals or bound variables outside the function. Protovis uses a family tree convention for scene graph navigation: property functions can refer to the `parent` panel that encloses the current mark, the previous `sibling`, or the `cousin` in the previous instantiation of the parent panel, as in Figure 2. A zero-based `index` attribute can be used to compute a regularly-spaced left position, sufficing as a trivial scale transformation.

Interactivity can be added by registering **event handlers** on marks. Handlers respond to mouse or keyboard input events and mutate the scene graph or underlying data to update the display. For example, a handler might recolor a mark red in response to a mouse click event: `event("click",function()this.fillStyle("red"))`.

All marks share two fundamental properties, `data` and `visible`. The data property is typically a static array, though in the case of nested panels, a function can be used to dereference hierarchical data, as in small multiple displays. Visibility determines whether or not a mark is rendered; if false, the other properties are not evaluated.

3.1.2 Box Model

Many graphical systems use a Cartesian coordinate system where the location of graphical elements is specified with a two-dimensional vector $\langle x, y \rangle$. Protovis takes a slightly different approach, adapting the CSS box model [8]. This is consistent with the layout of visual elements in web pages, and allows the specification of right-facing or top-facing charts (akin to bidirectional text), simply by changing left to right or bottom to top.

Points are specified with two orthogonal properties; using left and top is equivalent to placing the origin in the top-left corner of the viewport. If redundant properties are specified, left takes priority over right and top takes priority over bottom. Boxes (axis-aligned rectangles) can be specified using four orthogonal properties, such as left, top, width and height, as shown in Figure 3.

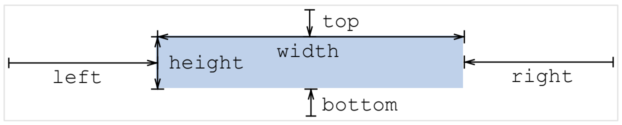


Fig. 3. Specifying position and dimension using the box model.

3.2 Panels

Panels allow repeated or nested structures, commonly used in small multiple displays [30] where a small visualization is tiled to facilitate comparison across one or more dimensions. Other types of visualizations may benefit from repeated and possibly overlapping structure as well, such as the stacked area chart in Figure 2. Panels can also offset the position of marks to provide padding from surrounding content.

All Protovis displays have at least one panel; this is the root panel to which marks are rendered. The box model properties (four margins, width and height) are used to offset the positions of contained marks. The data property determines the panel count: a panel is generated

```
    var panel = new pv.Panel()
      .width(160).height(160)
      .bottom(10).left(10).right(30);

  ┌ panel.add(pv.Area)
  │   .data([1, 1.2, 1.7, 1.5, .7, .5, .2])
a │   .bottom(0)
  │   .height(function(d) d * 80)
  │   .left(function() this.index * 25)
  └   .fillStyle("lightblue")
b ┌   .anchor("top").add(pv.Line)
  └     .strokeStyle("black")
c └     .add(pv.Dot);

d ┌ panel.add(pv.Rule)
  └   .bottom(0)
  ┌   .add(pv.Rule)
  │     .data(pv.range(.5, 2, .5))
e │     .bottom(function(d) d * 80)
  └     .strokeStyle("white")
f └     .anchor("right").add(pv.Label);

    panel.render();
```

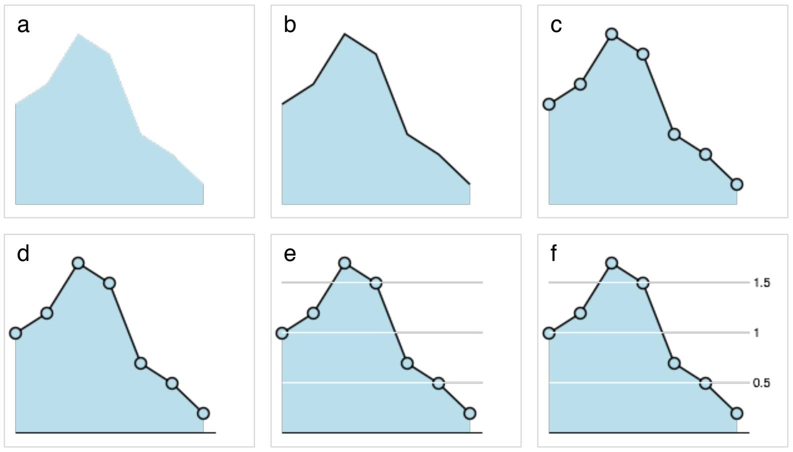
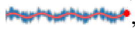


Fig. 4. Dissection of an area chart specification in Protovis. The first three lines set the dimensions and margins of the root panel. (a) Next, an area mark visually encodes the array of numbers with height. (b) A black line is added to the area's top anchor for emphasis. (c) A dot is derived from the line to indicate samples; note that it inherits the fill color from the area. (d) A horizontal rule, added to the panel, serves as the x-axis. (e) A second rule implements white grid lines. (f) Finally, a label is added to the rule's right anchor to show reference values.

once per associated datum. When nested panels are used, property functions can declare additional arguments to access the data associated with enclosing panels.

Panels can be rendered inline, facilitating the creation of sparklines [32] such as , , or . This allows designers to reuse browser layout features, such as text flow and tables; designers can also overlay HTML elements such as rich text and images.

3.3 Built-in Mark Types

We now describe the initial set of graphical primitives provided by Protovis. Protovis is extensible, and we expect to introduce new mark types in the future. At the same time, it is desirable to keep the set of supported marks small, so as to avoid overwhelming users with a cornucopia of similar options.

3.3.1 Shapes

The **Line** mark type represents a series of connected line segments, or *polyline*, that can be stroked with a configurable color and thickness. By default, a distinct stroke color is allocated from a stock color palette. Each articulation point in the line corresponds to a datum; for n points, n - 1 connected line segments are drawn. The point is positioned using the box model. Arbitrary paths are also possible, allowing radar plots and other custom visualizations.

Just as a line represents a polyline, the **Area** mark type represents a *polygon*. However, an area is not an arbitrary polygon; vertices are paired either horizontally or vertically into parallel *spans*, and each span corresponds to an associated datum. Either the width or the height must be specified, but not both; this determines whether the area is horizontally-oriented or vertically-oriented. Like lines, areas can be stroked and filled with arbitrary colors.

The **Bar** mark type is an axis-aligned rectangle that can be stroked and filled. Bars are used for many chart types, including bar charts, histograms and Gantt charts. Bars can also be used as decorations, for example to draw a frame border around a panel.

A **Dot** is simply a sized glyph (e.g., ×, ▽, ◇, □, ○) centered at a given point that can also be stroked and filled. The `size` property is proportional to the area of the rendered glyph to encourage meaningful visual encodings. Dots can visually encode up to eight dimensions of data, though this may be unwise due to integrality [34].

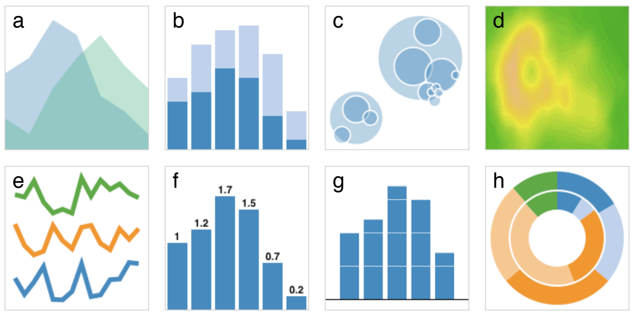


Fig. 5. Examples of built-in mark types. (a-h) Area; Bar; Dot; Image; Line; Label and Bar; Rule and Bar; Wedge.

No visualization system would be complete without the ability to render the ubiquitous pie chart, so a **Wedge** mark type is provided. Specified in terms of start and end angle, inner and outer radius, wedges can be used to construct donut charts and polar bar charts as well. If the `angle` property is used, the end angle is implied by adding this value to start angle. By default, the start angle is the previously-generated wedge's end angle. This design allows explicit control over the wedge placement if desired, while offering convenient defaults for the construction of radial graphs.

The **Rule** mark type is provided to render horizontal and vertical rules that are frequently needed for axes and grid lines. For example, specifying only the bottom margin draws horizontal rules, while specifying only the left draws vertical rules. Rules can also be used as thin bars. The visual style is controlled in the same manner as lines.

The **Link** mark type enables construction of node-link diagrams by specifying connections between mark instances. The `nodes` property determines a mark collection to use as nodes. Nodes are then indexed according to values returned by the `nodeKey` property. By default, a link's prototype mark provides the nodes (i.e., links "inherit" their nodes) and each node's zero-based mark index serves as its key. For each datum, the `sourceKey` and `targetKey` properties specify the link end points in terms of node keys. The specification can accomodate multiple graph and tree data structures, including separate node and edge tables, single tables with parent keys, pointer-based tree

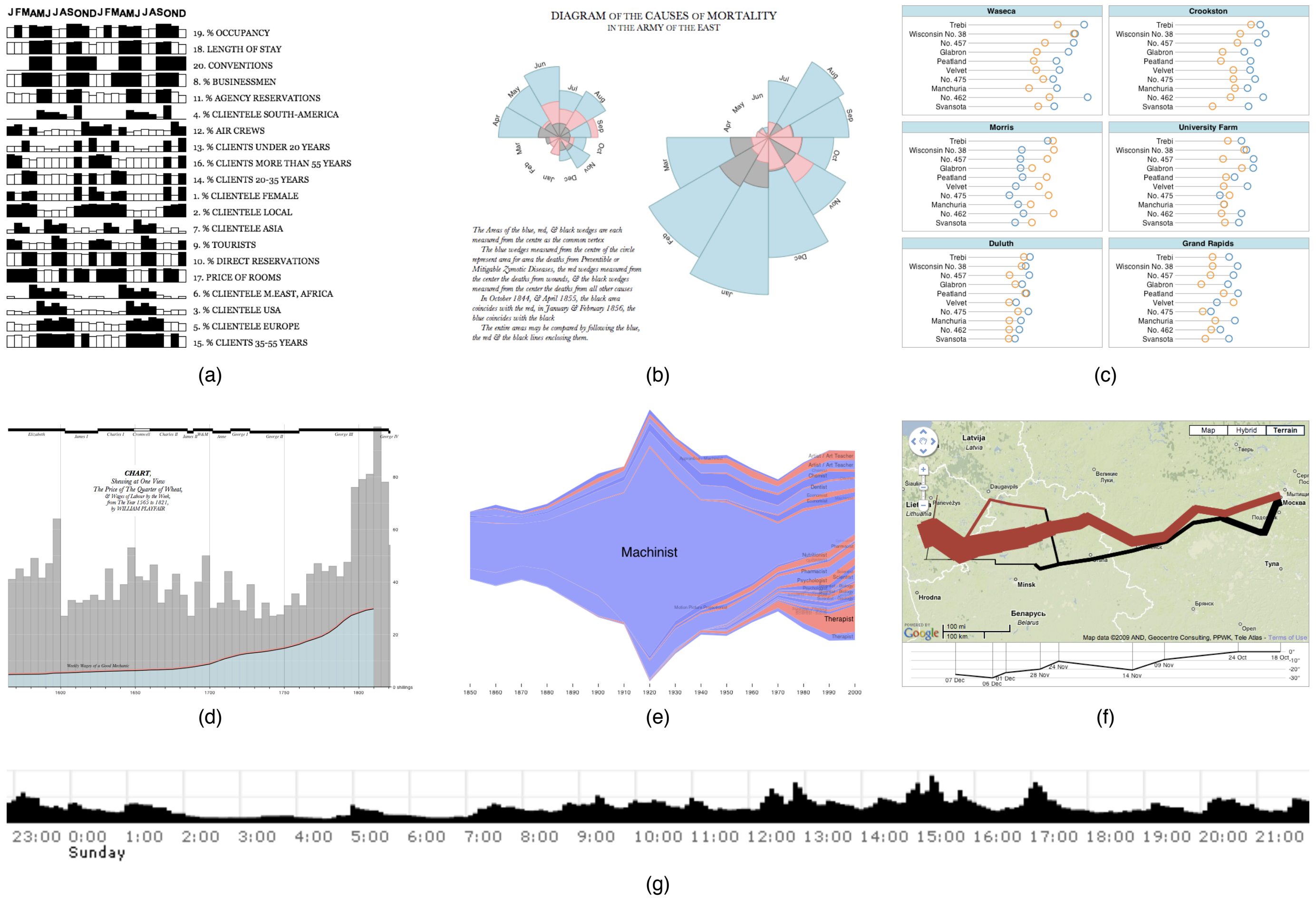


Fig. 6. Example applications. (a) Jacques Bertin's analysis of hotel patterns. (b) Florence Nightingale's chart of deaths in the Crimean War. (c) A trellis display of barley yields. (d) William Playfair's chart comparing the price of wheat and wages. (e) The Job Voyager. (f) Charles Minard's flow map of Napoleon's march to Moscow, as a Google Maps "mash-up". (g) Live Twitter updates containing the word "oakland".

structures, and adjacency lists. Currently links are drawn as straight lines styled identically to line marks, with optional arrows specified by the `sourceEnd` and `targetEnd` properties. The optional anchor properties `sourceAnchor` and `targetAnchor` afford more precise positioning of end points with respect to nodes.

3.3.2 Text and Images

The **Label** mark type allows textual annotation, such as labeling points in a scatterplot, axis ticks and legends. A number of text placement properties are available in addition to box model positioning: rotation angle, horizontal and vertical alignment. The text itself is set with the `text` property, whose default value is the identity function.

Finally, the **Image** mark type supports static images and the generation of dynamic images such as heat maps. For the latter, the color of each pixel is specified using a function. Figure 5(d) shows an example image that visually encodes elevation using a color gradient.

3.4 Data Transformation

Data is rarely in the exact format needed to produce the desired visualization. Much of the work involved in producing a visualization is finding data, munging it into a consistent format, deriving additional fields, and refolding it to match the structure of the visualization specification. Furthermore, knowledge gained from preliminary visualizations can motivate new visualizations, possibly requiring changes to the data format or new data from different sources [6]; making this process easy encourages users to make the right visualization, rather than compromising the design to suit the arbitrary format of the data.

A frequently-needed data transformation is to group relational data into a tree; this is accomplished using the **nest** operator [29, 37]. Given a key function, which returns the key value for a given datum, the nest operator groups data with the same key value. Multiple key functions can be specified to produce nested groups.

The nest operator also provides sorting and rollup functionality. Elements in the tree can be sorted by keys or values, using either default lexicographic ordering or a custom comparator function. The `rollup` method returns a map with entries for each key, whose values are the results of applying a function to each group of elements. For example, rollup can be used to compute the median value for each group, which can then be used to sort groups (as in Figure 6(c)).

Protovis includes a number of summary statistics, such as count, sum, max, min, mean, median, and quantile. We also provide several methods for arrays, such as number range generation, scaling, permutation, and cross and blend operators [37].

4 Evaluation

As we developed Protovis, we sought to meet our design goals of creating an *expressive*, *efficient*, and *accessible* visualization tool. Here we present findings from early-stage evaluations. To evaluate expressiveness and efficiency, we built a variety of example applications. To evaluate accessibility, we analyzed Protovis using the Cognitive Dimensions of Notation framework [13] and solicited feedback from designers. Where applicable, we compare Protovis with Processing and Flare, popular tools that serve as exemplars of low-level graphics programming and high-level visualization systems, respectively.

Table 1. Load time and memory usage on example charts (n=10).

Chart	Time (ms)		Memory (MB)	
	μ	σ	μ	σ
blank page	*19*	*1*	*3.94*	*0.03*
crimea-rose	38	4	0.25	0.04
barley-trellis	52	3	0.37	0.04
wheat-playfair	55	8	0.26	0.04
hotels-bertin	61	5	0.62	0.04
job-voyager	499	10	0.73	0.05
line-10	32	3	0.14	0.08
line-100	34	1	0.20	0.05
line-1,000	55	2	0.19	0.04
line-10,000	271	10	0.40	0.05
line-100,000	2,630	22	2.01	0.04
dot-10	32	2	0.12	0.04
dot-100	36	2	0.18	0.06
dot-1,000	94	2	1.08	0.05
dot-10,000	692	8	10.1	0.06
dot-100,000	7,370	66	102	1.83

4.1 Applications and Performance

Using Protovis, we were able to quickly and concisely specify a diverse set of visualizations. A subset of these examples is shown in Figure 6, including classic visualizations originally drawn by hand juxtaposed with more modern examples, such as a "mash-up" with online mapping tools and interactive stacked graphs. We indeed found that Protovis facilitated design and implementation: Bertin's permutation matrix of hotel visits required only 38 lines of JavaScript, Nightingale's *coxcomb* 55, and Becker et al.'s *trellis display* (complete with main-effects ordering) only 45.

To evaluate system performance, we used profiling tools to measure the load time and memory usage of several example charts. The results are shown in Table 1. The data was collected using SVG in Safari 4 on Mac OS X 10.5.7, with a 2.8 GHz Intel Core 2 Duo processor and 4 GB of RAM. The numbers for each chart are shown relative to the baseline time and memory for a blank page. Also included are standard line and scatterplots for datasets ranging from 10 to 100,000 points. Although the thin-client SVG renderer is usable with up to 10,000 data points, larger datasets likely require a different system architecture—though not necessarily a different specification language.

We now present a pair of in-depth comparisons: a simple pie chart specification and the Job Voyager, an interactive stream graph [5].

4.1.1 Pie Charts

The specifications for a simple pie chart ( ) in Protovis, Processing and Flare are shown in Figure 7. Although the pie chart is a trivial instantiation of a standard chart type (neither custom nor interactive), differences between the three systems are apparent.

Processing uses an imperative specification where most instructions either render a shape immediately or have a side-effect on the drawing environment. For instance, `noStroke` disables stroking of paths, so that calls to `arc` are only filled. Like Protovis, Processing uses simple graphical primitives; each pie wedge is specified literally in terms of radii and angles. However, the exact mapping of data to visual properties is arguably less clear. While the impact on this simple example is minor, problems of accessibility accrue as complexity increases.

The Flare specification employs two visualization abstractions that simplify chart construction. The first is `PieLayout`, which translates the univariate data into the dimensions of each pie slice. The second is `ColorEncoder`, which translates the same data into a suitable quantitative color encoding with a light-to-dark blue gradient. The higher-level abstractions of Flare arguably result in less effort, thanks to the automatic layout and color encoding. However, the user must understand these abstractions in order to use them ("How do I reference the data values? What are 'nodes'?"), and they may not be suitable for custom visualizations.

a

```
new pv.Panel().width(150).height(150)
  .add(pv.Wedge)
    .left(75).bottom(75)
    .outerRadius(70)
    .angle(function(d) d * 2 * Math.PI)
    .data(pv.normalize([1, 1.2, 1.7, 1.5, .7]))
  .root.render();
```

b

```
size(150, 150);
background(255);
smooth();
noStroke();
int diameter = 140;
float[] data = {1, 1.2, 1.7, 1.5, .7};
float lastAng = 0, sum = 0;
for (int i = 0; i < data.length; i++) {
  sum += data[i];
}
for (int i = 0; i < data.length; i++) {
  fill(data[i] * 120);
  float ang = data[i] / sum * 2 * PI;
  arc(width / 2, height / 2, diameter,
      diameter, lastAng, lastAng + ang);
  lastAng += ang;
}
```

c

```
vis = new Visualization(
    Data.fromArray([{value: 1}, ...]));
vis.bounds = new Rectangle(0, 0, 150, 150);
vis.operators.add(new PieLayout(
    "data.value", 0.9));
vis.operators.add(new ColorEncoder(
    "data.value", "nodes", "fillColor"));
vis.update();
addChild(vis);
```

Fig. 7. Specifying a simple pie chart. (a-c) Protovis; Processing; Flare.

4.1.2 Job Voyager

The **Job Voyager** by Heer et al. [18] shows U.S. census data for occupation and gender over 150 years. The dataset has over 6,000 data points. The display is interactive, allowing the user to filter the visible occupations according to a search query. Figure 6(e) shows jobs with the suffix "-ist". The data is visualized as a stacked area chart using a *ThemeRiver* layout [5, 14]. A single area mark is enclosed by a panel that is repeated for each job-sex pair. This is a refinement of the technique shown in Figure 2; the area's cousin determines the bottom position. The nest operator groups the relational data by both sex and occupation, and then sorts alphabetically by occupation.

The Protovis version is notably more concise than the Flare original: with comments and import statements removed, the Flare version remains over twice as long. However, the Flare version uses pre-built operators for performing area and label layout, whereas the Protovis version directly specifies the stacking and labeling rules; a more balanced comparison including the size of Flare operators increases the code size ratio from 2:1 to 5:1. Of course, to developers familiar with Flare, these toolkit abstractions may make the specification easier to implement due to encapsulation. Furthermore, they facilitate niceties such as animated transitions between states. We plan to add similar support to future versions of Protovis, though such abstractions would be optional, and not required learning for all users.

4.2 Cognitive Dimensions of Notation

Though counting lines of code provides some insight into the effort required to specify a visualization, it alone is hardly a satisfactory measure. As an initial assessment of Protovis' accessibility, we performed an analysis using the Cognitive Dimensions of Notation (CDN) framework [13], an inspection method for evaluating the effectiveness of notational systems such as programming languages and visual interfaces.

CDN provides a collection of **cognitive dimensions**: useful heuristics for evaluating a notation system and the environment in which it is manipulated. Due to space limitations, we briefly discuss only a relevant subset of the 14 cognitive dimensions; we draw comparisons to other tools where appropriate.

Closeness of mapping (closeness of representation to domain) · In Protovis, the notation and problem domain are closely linked: visualizations are defined directly in terms of graphical marks, though one has to map from textual specification to visual output. Furthermore, Protovis uses existing web conventions such as the CSS box model to leverage existing knowledge. In contrast, rendering APIs such as Processing provide imperative rendering commands that also correspond to graphic marks, but lack both direct specification of marks in terms of properties and a scene graph that allows object-level manipulation of marks. Flare instead requires conceptualizing the visualization in terms of encoding operators, which in turn have a number of effects (and side-effects) upon visual properties.

Hidden dependencies (important links between entities are not visible) · Protovis provides a few "behind-the-scenes" facilities such as `parent`, `sibling`, and `cousin` scene graph accessors, and inheritance may at times result in unintended property settings. Flare uses a host of abstractions including operators, axes, scale bindings, renderers, and animators, few of which are visible in the visualization specification. As a result, designers often report that it is frustrating to determine the provenance of visual properties ("Is the axis layout overwriting my manual scale range settings?").

Role-expressiveness (the purpose of a component is readily inferred) and *Visibility* (ability to view components easily) · Protovis specifications directly denote the mark types and their visual properties, enabling identification of each visual mapping. However, as discussed above, inheritance hampers this visibility. In Flare, the operators are typically identified by their name and constructor parameters; in many cases documentation or code-inspection is required to understand the effects of running the operator.

Consistency (similar semantics are expressed in similar syntactic forms) · Protovis reuses the semantics of mark properties as much as possible to facilitate consistency and improve learning. Some inconsistencies do occur: for example Marks do not inherit Panel properties. In Flare, most operators are idiosyncratic but are instantiated and composed in a consistent fashion. However, the relation between visual properties and the actual appearance is indirect: Flare uses pluggable renderers that can use or ignore visual object properties.

Viscosity (resistance to change) · Protovis allows rapid iteration by changing property settings, and the visibility and consistency of properties facilitates rapid switching between mark types (except when switching between Cartesian and radial marks). Flare allows quick changes to operator settings and composition, but there are little consistency guarantees of properties across operators. Processing requires direct modification of imperative code, and so viscosity is largely determined by the developer's own engineering skill.

Hard mental operations (high demand on cognitive resources) · By directly surfacing visual property values, Protovis may incur more mental effort regarding the mathematics of visual encoding, e.g., multiplying index and data values rather than simply invoking an axis encoding as in Flare. Protovis provides scale transforms to aid this process, at the cost of learning an additional, through relatively straightforward, abstraction. In Flare, the most common taxing operations are search tasks due to lack of visibility or hidden dependencies: how to configure an operator, override a buried default value, or access an intermediate abstraction.

Diffuseness (verbosity of language) · Section 4.1 discusses tool verbosity. We have found Protovis to be concise in practice, though this may be complicated by the incorporation of additional libraries or interaction with the browser. Flare descriptions are also concise so long as an operator for the desired task is already defined.

Abstraction (types and availability of abstraction mechanism) · Protovis does not yet provide toolkit mechanisms for abstraction, such as template definitions. However, akin to Processing, programming language abstractions such as functions and classes can be used. Flare enables abstraction via operator and interactor classes, though these require significant toolkit knowledge to author.

In summary, we have found that Protovis rates favorably in terms of closeness of mapping, hidden dependencies, visibility, consistency, viscosity, and diffuseness—lending credence to our claim of toolkit accessibility. The analysis also points to some areas for potential innovation, e.g., promoting visibility across inheritance relations.

4.3 Designer Feedback

We are also actively seeking feedback from visualization designers and tool builders to inform the iterative design of Protovis. Initial feedback has been quite positive; respondents have particularly praised the direct and concise nature of Protovis specifications. For example, one toolkit author wrote, *"You've captured the basics in a concise and elegant manner,"* while another designer noted that Protovis *"gives us a lot of flexibility with a shallow learning curve. Great things might come from this."* Others surprised us by quickly responding with demos they had built, including the first "mash-up" of Protovis with mapping libraries and a bar chart of live posting activity on Twitter; see Figure 6(f,g). Mozilla Messaging is working to include Protovis with Thunderbird 3, calling it *"delightful"* and *"useful to add-on authors."*

Respondents also pointed out important areas for improvement. Most feedback noted the need for higher-level support for scale transforms and axis labeling; we subsequently added scale facilities for computing ranges, setting mark properties, and generating tick and label values. We have maintained an "opt-in" design philosophy for such higher-level abstractions: we support various statistical and charting features in a way that complements our mark-based specifications, ensuring that higher-level abstractions can be easily added, but also easily removed or stubbed-out. Other requests included richer support for interaction and animation, which we are now implementing.

5 Discussion and Future Work

The previous sections give some indication of the effort required to construct "real-world" custom visualizations in our system, as well as the system's capabilities and performance. While it is true that other systems are more concise, few simultaneously provide the same degree of flexibility and control. And, because our system uses only simple shapes with explicit visual encodings, its behavior is transparent: there is not a lot of "magic" behind the scenes. We posit that the directness of the specification makes the system more accessible to new users.

5.1 Evaluation

Of course, our hypotheses require further inquiry. While we have applied the Cognitive Dimensions of Notation framework in an initial usability evaluation, both user studies and observed real-world usage will further evaluate our claims. There are many questions to ask:

- Is the system easy to learn for new users? How quickly can novice users learn to develop nontrivial, original works?
- Is the system accessible to non-programmers? Are the features modeled after web page layout useful, or is the traditional mathematical graph approach more effective?
- Is the system suitable for complex visualizations? Are existing features sufficiently expressive, or are new graphical primitives and abstractions necessary?
- What impact does the system have on the creative process? Will users produce better visualizations with a more expressive system, or are errors, missing features and meaningless encodings more likely?

The last of these questions are perhaps the most important, since they are relevant not just to producers of visualizations, but to consumers. By taking a bottom-up approach, every visual element in Protovis is specified explicitly in contrast to top-down approaches where visual elements are inherited from a template. As a result, our system favors minimalist designs. This may discourage chartjunk and raise the data-ink ratio [30], but at the same time it risks the omission of helpful marks that are cumbersome to specify.

In the same vein, by requiring that the user specify the visualization explicitly and by making all visual properties equally dynamic, it may be more likely that an inappropriate visual encoding is used. Compare this to a system such as Polaris or Mackinlay's APT [22], which intelligently suggest meaningful designs based on metadata. The corollary is that high-level visualization systems may inadvertently make the wrong automatic decision, and not allow the user to compensate. If we can encourage users to produce the right visualization by example and with appropriate low-level building blocks, we may help users construct effective visualizations without artificially restricting the design space. Furthermore, automatic presentation tools may be adapted to generate Protovis specifications, combining the benefits of high-level tools with low-level design control.

Going forward, we plan to refine and extend the design of Protovis. These plans include the design of a development environment to assist visualization creation (a strategy that has proven effective for tools such as Processing), potentially including a visual editor. We also intend to conduct a formal evaluation of Protovis development. We consider such work to be largely formative; we believe the true measure of the toolkit's value lies in the creation and dissemination of successful visualization by others. We will accordingly continue to monitor and respond to the needs and insights of Protovis users.

5.2 An Ecology of Visualization

One of the most exciting features of our system is that visualizations become open source: since specifications are concise and are not compiled, but instead interpreted at runtime by the web browser, users can easily view the source code and data behind any visualization. In addition to learning by example, visualizations are constructed from modular primitives which make it easier for designers to incorporate discovered techniques into their own work through copy-and-paste [4].

Open source also facilitates some degree of collaboration [16], since users can more easily create derivative works to show different views or fix mistakes. For example, in our recreation of Burtin's antibiotic effectiveness chart (Figure 1) [27], we discovered a missing grid line for the minimum inhibitory concentration of 0.01, as well as an exaggeration of some values for Penicillin.

Systems such as sense.us [18] and Many Eyes [33] have helped demonstrate the value of collaborative *sharing* of data, visualizations, and insights gained from an analysis. The Google Visualization API, meanwhile, lets developers package chart templates as "gadgets" for reuse by others. However, one component of the visualization pipeline that is not well addressed by these systems is the collaborative *construction* of novel, interactive visualizations. Just as HTML enables "everyday programmers" [28] to author web pages, by balancing expressiveness, efficiency, and accessibility, we hope that Protovis will help foster a greater diversity of customized web-based visualizations.

Acknowledgments

The authors thank Tom Carden and Michal Migurski of Stamen Design, David Ascher and Andrew Sutherland of Mozilla, and Nelson Minar, Hadley Wickham, and Daniel Gonçalvez for their feedback.

References

[1] B. B. Bederson, J. Grosjean, and J. Meyer. Toolkit design for interactive structured graphics. *IEEE Trans. on Software Engineering*, 30(8):535–546, 2004.

[2] M. Bender, R. Klein, A. Disch, and A. Ebert. A functional framework for web-based information visualization systems. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):8–23, 2000.

[3] J. Bertin. *Semiology of graphics*. University of Wisconsin Press, Madison, WI, 1983.

[4] J. Brandt, P. J. Guo, J. Lewenstein, and S. R. Klemmer. Opportunistic programming: how rapid ideation and prototyping occur in practice. In *WEUSE '08: Proc. 4th Workshop on End-User Software Engineering*, pages 1–5, New York, NY, 2008. ACM.

[5] L. Byron and M. Wattenberg. Stacked graphs – geometry & aesthetics. *IEEE Trans. Vis. and Comp. Graphics*, 14(6):1245–1252, 2008.

[6] S. K. Card, J. D. Mackinlay, and B. Shneiderman, editors. *Readings in information visualization: using vision to think*. Morgan Kaufmann, San Francisco, CA, 1999.

[7] E. H. Chi and J. T. Riedl. An operator interaction framework for visualization systems. In *Proc. IEEE InfoVis*, pages 63–70, 1998.

[8] CSS Box Model. `http://w3.org/TR/CSS2/box.html`, March 2009.

[9] S. G. Eick, M. A. Eick, J. Fugitt, B. Horst, M. Khailo, and R. A. Lankenau. Thin client visualization. In *VAST '07: Proceedings of the 2007 IEEE Symposium on Visual Analytics Science and Technology*, pages 51–58, Washington, DC, USA, 2007. IEEE Computer Society.

[10] J.-D. Fekete. The InfoVis Toolkit. In *Proc. IEEE InfoVis*, pages 167–174, 2004.

[11] S. Few. *Show Me the Numbers: Designing Tables and graphs to Enlighten*. Analytics Press, Berkeley, CA, 2004.

[12] Flare. `http://flare.prefuse.org`, March 2009.

[13] T. R. G. Green. Cognitive dimensions of notations. In *Proceedings of the fifth conference of the British Computer Society, Human-Computer Interaction Specialist Group on People and computers V*, pages 443–460, New York, NY, USA, 1989. Cambridge University Press.

[14] S. Havre, E. Hetzler, P. Whitney, and L. Nowell. ThemeRiver: Visualizing thematic changes in large document collections. *IEEE Trans. Vis. and Comp. Graphics*, 8(1):9–20, 2002.

[15] J. Heer and M. Agrawala. Software design patterns for information visualization. *IEEE Trans. Vis. and Comp. Graphics*, 12(5):853–860, 2006.

[16] J. Heer and M. Agrawala. Design considerations for collaborative visual analytics. *Information Visualization*, 7(1):49–62, 2008.

[17] J. Heer, S. K. Card, and J. A. Landay. prefuse: a toolkit for interactive information visualization. In *Proc. ACM CHI*, pages 421–430, 2005.

[18] J. Heer, F. B. Viégas, and M. Wattenberg. Voyager and voyeurs: Supporting asynchronous collaborative information visualization. In *Proc. ACM CHI*, pages 1029–1038, 2007.

[19] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28:196, December 1996.

[20] D. W. Johnson and T. J. Jankun-Kelly. A scalability study of web-native information visualization. In *GI '08: Proceedings of graphics interface 2008*, pages 163–168, Toronto, Ont., Canada, Canada, 2008. Canadian Information Processing Society.

[21] JSON. `http://json.org`, March 2009.

[22] J. Mackinlay. Automating the design of graphical presentations of relational information. *ACM Trans. Graph.*, 5(2):110–141, 1986.

[23] A. Maslow. *The Pyschology of Science: A Reconnaissance*. Harper & Row, Madison, WI, 1966.

[24] P. J. Moran and C. Henze. Large field visualization with demand-driven calculation. In *VIS '99: Proc. Visualization '99*, pages 27–33, 1999.

[25] D. A. Norman. *The Psychology of Everyday Things*. Basic Books, New York, NY, 1988.

[26] Processing. `http://processing.org`, March 2009.

[27] R. R. Remington and R. Fripp. *Design and Science: The Life and Work of Will Burtin*. Ashgate, 2007.

[28] M. B. Rosson, J. Ballin, and H. Nash. Everyday programming: Challenges and opportunities for informal web development. In *VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, pages 123–130, Washington, DC, USA, 2004. IEEE Computer Society.

[29] C. Stolte, D. Tang, and P. Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Trans. Vis. and Comp. Graphics*, 8:52–65, 2002.

[30] E. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, CT, 1992.

[31] E. Tufte. Ask E.T.: Graphing Software. `http://www.edwardtufte.com/bboard/q-and-a-fetch-msg?msg_id=00000p`, April 2001.

[32] E. Tufte. *Beautiful Evidence*. Graphics Press, Cheshire, CT, 2006.

[33] F. B. Viégas, M. Wattenberg, F. van Ham, J. Kriss, and M. McKeon. Many Eyes: a site for visualization at internet scale. *IEEE Trans. Vis. and Comp. Graphics*, 13(6):1121–1128, 2007.

[34] C. Ware. *Information visualization: perception for design*. Morgan Kaufmann, San Francisco, CA, 2004.

[35] C. E. Weaver. Building highly-coordinated visualizations in Improvise. In *Proc. IEEE InfoVis*, pages 159–166, 2004.

[36] H. Wickham. ggplot2. `http://had.co.nz/ggplot2/`.

[37] L. Wilkinson. *The Grammar of Graphics (Statistics and Computing)*. Springer-Verlag New York, Inc., Secaucus, NJ, 2005.

[38] J. Wood, K. Brodlie, J. Seo, D. Duke, and J. Walton. A web services architecture for visualization. *eScience, IEEE International Conference on*, 0:1–7, 2008.