$\mathbb{D}^3$: Data-Driven Documents

Michael Bostock, Vadim Ogievetsky and Jeffrey Heer

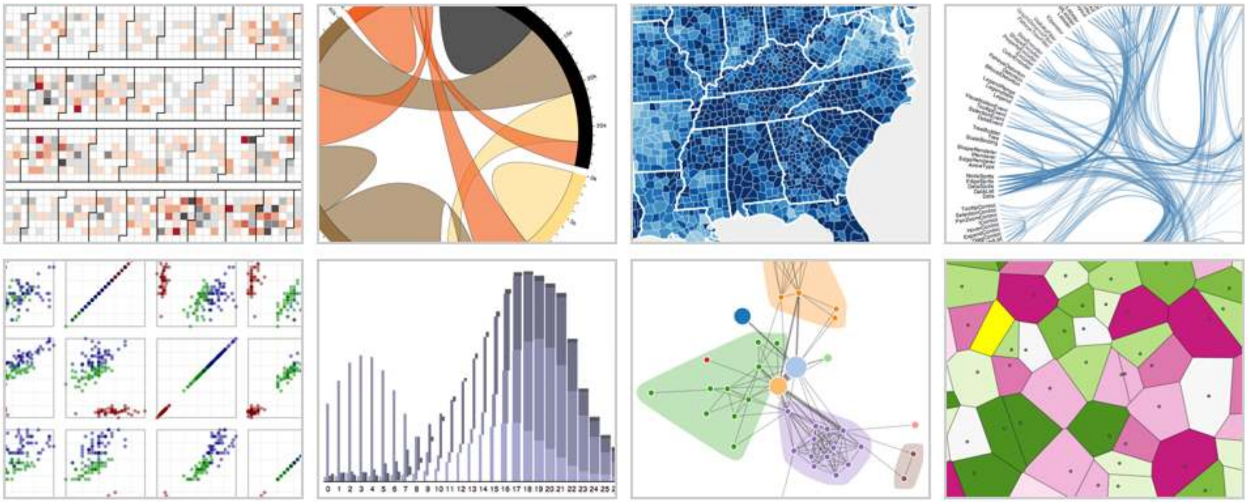

Fig. 1. Interactive visualizations built with D3, running inside Google Chrome. From left to right: calendar view, chord diagram, choropleth map, hierarchical edge bundling, scatterplot matrix, grouped & stacked bars, force-directed graph clusters, Voronoi tessellation.

Abstract—Data-Driven Documents (D3) is a novel representation-transparent approach to visualization for the web. Rather than hide the underlying scenegraph within a toolkit-specific abstraction, D3 enables direct inspection and manipulation of a native representation: the standard *document object model* (DOM). With D3, designers selectively bind input data to arbitrary document elements, applying dynamic transforms to both generate and modify content. We show how representational transparency improves expressiveness and better integrates with developer tools than prior approaches, while offering comparable notational efficiency and retaining powerful declarative components. Immediate evaluation of operators further simplifies debugging and allows iterative development. Additionally, we demonstrate how D3 transforms naturally enable animation and interaction with dramatic performance improvements over intermediate representations.

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

1 Introduction

When building visualizations, designers often employ multiple tools simultaneously. This is particularly true on the web, where interactive visualizations combine varied technologies: HTML for page content, CSS for aesthetics, JavaScript for interaction, SVG for vector graphics, and so on. One of the great successes of the web as a platform is the (mostly) seamless cooperation of such technologies, enabled by a shared representation of the page called the *document object model* (DOM). The DOM exposes the hierarchical structure of page content, such as paragraph and table elements, allowing reference and manipulation. In addition to programming interfaces, modern browsers include powerful graphical tools for developers that display the element tree, reveal inherited style values, and debug interactive scripts.

Unfortunately, this blissful interoperability is typically lost with visualization toolkits due to encapsulation of the DOM with more specialized forms. Rather than empowering direct manipulation of the existing model, such toolkits [2, 9, 18] supplant it with custom scenegraph abstractions. This approach may provide substantial gains in *efficiency*—reducing the effort required to specify a visualization—but it incurs a high opportunity cost: it ignores developers' knowledge of standards, and the tools and resources that augment these standards.

The resulting cost to *accessibility*—the difficulty of learning the representation—may trump efficiency gains, at least for new users. Scarcity of documentation and ineffectual debugging exacerbate the problem, impeding users from gaining deeper understanding of toolkit abstractions and limiting the toolkit's potential. Systems with intermediate scenegraph abstractions and delayed property evaluation can be particularly difficult to debug: internal structures are exposed only when errors arise, often at unexpected times.

Furthermore, intermediate representations may diminish *expressiveness*—the diversity of possible visualizations—and introduce substantial runtime overhead. Certain tasks that could be offloaded to a more suitable tool, such as specifying fonts via CSS, may be stymied by encapsulation. Similarly, while graphical features such as clipping may be supported by the underlying representations, they may not be exposed by the toolkit. Even if extensibility is available as a means for greater expression, it requires in-depth knowledge of toolkit internals and poses a substantial barrier to the average user.

Our awareness of these issues comes in part from thousands of user observations over the two years since releasing Protovis [2], despite our attempt to balance expressiveness, efficiency and accessibility. We now refine these three goals with specific objectives:

Compatibility. Tools do not exist in isolation, but within an ecosystem of related components. Technology reuse utilizes prior knowledge and reference materials, improving accessibility. Offloading a subset of tasks to specialized tools can improve efficiency, avoiding the generality and complexity of a monolithic approach. And, full access to

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

Manuscript received 31 March 2011; accepted 1 August 2011; posted online 23 October 2011; mailed on 14 October 2011.
For information on obtaining reprints of this article, please send email to: tvcg@computer.org.

Fig. 2. Treemaps made with D3. The visualizations are implemented in pure HTML & CSS, improving browser compatibility. A stable layout algorithm enables animated transitions for changing cell values (from left to right) without disorienting rearrangements.

the native representation removes limits on expressiveness.

Debugging. Trial and error is a fundamental part of development and the learning process; accessible tools must be designed to support debugging when the inevitable occurs. Better tools facilitate poking and prodding to explore the side-effects of operations interactively. While encapsulation of control flow and representation often improves efficiency, it may also lead to an "impedance mismatch" if internal state is exposed, violating the user's mental model.

Performance. Visualizations can be greatly enhanced by interaction and animation [15]. However, high-level abstractions may limit a developer's ability to execute fast incremental scene changes if the system lacks sufficient information (such as a dependency graph) to avoid redundant computation. Focusing on transformation rather than representation shifts this responsibility to the developer, improving performance while enabling animation and interaction.

To address these concerns, we contribute Data-Driven Documents (D3), an embedded domain-specific language [16] for transforming the document object model based on data. With D3, designers selectively bind input data to arbitrary document elements, applying dynamic transforms to both generate and modify content; the document *is* the scenegraph. This is a generalization of Protovis, and through declarative helper modules built on top of these transforms, we can achieve specifications with comparable notational efficiency. And yet, D3's standardized representation improves expressiveness and accessibility, while transforms offer dramatic performance gains and enable animated transitions.

We argue these claims by comparing D3 to existing web-based methods for visualization, considering how language design achieves our objectives; we also describe several applications to convey representative usage. Through performance benchmarks, we demonstrate that D3 is at least twice as fast as Protovis. Lastly, we share anecdotes that suggest D3's potential for dynamic visualization.

2 Related Work

D3 is not a traditional visualization framework. Rather than introduce a novel graphical grammar, D3 solves a different, smaller problem: efficient manipulation of documents based on data. Thus D3's core contribution is a visualization "kernel" rather than a framework, and its closest analogues are other document transformers such as jQuery, CSS and XSLT. As the document model directly specifies graphical primitives, D3 also bears a resemblance to low-level graphics libraries such as Processing and Raphaël. For high-level capability, D3 includes a collection of helper modules that sit on top of the selection-based kernel; these modules are heavily influenced by prior visualization systems, including Protovis.

2.1 Document Transformers

Although browsers have built-in APIs for manipulating the DOM, these interfaces are verbose and cumbersome, likely due to standards bodies' emphasis on unambiguous designs that can be implemented consistently by vendors and survive future revision. As a result, JavaScript libraries [19, 23, 29, 33] that enable more convenient manipulation are hugely popular. Of these, jQuery, is so successful it is often considered synonymous with JavaScript among novices.

These libraries share the concept of a *selection*: identify a set of elements using simple predicates, then apply a series of operators that mutate the selected elements. The universality of this concept is no coincidence; the idea originates from Cascading Style Sheets [21] (CSS): a declarative language for applying aesthetics (e.g., fonts and colors) to elements. JavaScript-based selections provide flexibility on top of CSS, as styles can be computed dynamically in response to user events or changing data.

a
```
var ps = document.getElementsByTagName("p");
for (var i = 0; i < ps.length; i++) {
  var p = ps.item(i);
  p.style.setProperty("color", "white", null);
}
```

b
```
p { color: white; }
```

c
```
$("p").css("color", "white");
```

d
```
d3.selectAll("p").style("color", "white");
```

Fig. 3. A simple document transformation that colors paragraphs white. (a) W3C DOM API; (b) CSS; (c) jQuery; (d) D3.

For data visualization, document transformers must handle the creation and deletion of elements, not just the styling of existing nodes. This is impossible with CSS, and tedious with jQuery as it lacks a mechanism for adding or removing elements to match a dataset; data must be bound to nodes individually (if at all), rather than through a high-level data join (see §3.2). This makes jQuery incapable of *data-driven* transformations, and thus ill-suited for dynamic visualizations involving complex transitions.

Extensible Stylesheet Language Transformations [41] (XSLT) is another declarative approach to document transformation. Source data is encoded as XML, then transformed into HTML using an XSLT stylesheet consisting of template rules. Each rule pattern-matches the source data, directing the corresponding structure of the output document through recursive application. XSLT's approach is elegant, but only for simple transformations: without high-level visual abstractions, nor the flexibility of imperative programming, XSLT is cumbersome for any math-heavy visualization task (e.g., interpolation, geographic projection or statistical methods).

2.2 Graphics Libraries

Dealing directly with graphical marks provides a close cognitive mapping between the toolkit representation and the desired result, reducing the gulf of execution [24] for designers and improving accessibility. Yet, as previously discussed [2], low-level graphics libraries such as

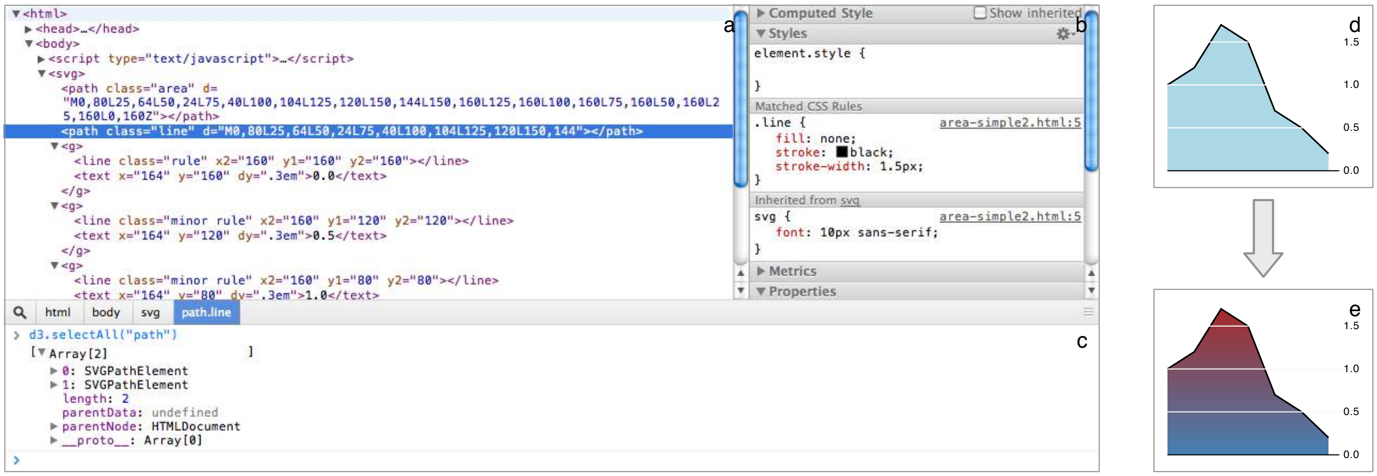

Fig. 4. Google Chrome's developer tools. The top regions inspect (a) the document hierarchy and (b) inherited style properties; underneath, (c) the console allows interactive evaluation of JavaScript. (d) The current document: an area chart. (e) The area chart modified through the console.

Processing [28] and Raphaël [30] are tedious for complex visualization tasks as they lack convenient abstractions.

Furthermore, many graphics libraries do not provide a scenegraph that can be inspected for debugging. For example, Processing uses immediate mode rendering, and Raphaël encapsulates SVG and Microsoft's proprietary Vector Markup Language (VML). Toolkit-specific scenegraph abstractions diminish compatibility and expressiveness: elements cannot be styled using external stylesheets, and graphical effects such as dashed strokes and composite filters may be unusable even if supported natively.

Minor variations in graphical abstractions also present a hurdle to new users. Consider drawing a wheel. In Processing, the `ellipse` operator draws a circle, which takes four arguments: x and y of the center, width and height. Raphaël provides a `circle` operator that takes three arguments, preferring radius. Protovis defines `Dot` and `Wedge` mark types, either of which can render circles, as well as the `Line` type with polar interpolation. Each abstraction differs slightly from the standard SVG `circle` element. Standards further benefit from a network effect: the more people that use a technology, the more demand for documentation, and thus the greater supply. Despite the efforts of developers to document their work, there are far more reference and training materials for standards than for custom libraries.

2.3 Information Visualization Systems

Researchers have developed a variety of toolkits for facilitating visualization design. One class of framework [8, 38] provides a hierarchy of visualization components. New visualizations are introduced either by authoring new components or subclassing existing ones. A second class of framework [9, 14] explicitly instantiates the InfoVis Reference Model [5, 12] using a set of composable operators for data management, visual encoding, and interaction. Though new combinations of operators can enable customized views, we have observed in practice that most novel visualizations require programmers to author completely new operators. Thus both classes of framework work well when visualization creators have software engineering expertise, but are prohibitive to more general audiences such as web designers.

With our prior work on Protovis [2, 13] we have instead advocated for *declarative, domain specific languages* (DSLs) for visualization design. By decoupling specification from execution details, declarative systems allow language users to focus on the specifics of their application domain, while freeing language developers to optimize processing. Similar to Protovis, D3 provides a declarative framework for mapping data to visual elements. However, unlike Protovis and other grammar-based declarative models [39, 40], D3 does not strictly impose a toolkit-specific lexicon of graphical marks. Instead, D3 directly maps data attributes to elements in the *document object model*.

Whether or not this design move is advantageous—or even possible—depends on context: many programming environments do not provide a standardized scenegraph abstraction. Moreover, toolkit-specific scenegraph abstractions have compelling benefits. As demonstrated in prior work [13], custom abstractions can facilitate portability (cross-platform deployment) and performance optimization. A curated lexicon of graphical marks can also improve notational efficiency [2]. Consequently, we maintain that toolkit-specific representations continue to be an important component of many visualization models, and we address this with D3's helper modules (see §3.4).

The technical constraints and entrenched standards of the web have led us to a different approach for browser-based visualization. The browser environment does not provide the same optimization opportunities as compiled programming languages; instead, the overhead of mapping an internal representation to the DOM introduces performance bottlenecks. Intermediate representations can also complicate debugging, as the mapping between code (written in terms of abstract graphical marks) and inspectable output (e.g., SVG elements in the DOM) is often unclear. Custom abstractions may additionally limit expressiveness: they must be revisited to take advantage of new browser features and due to encapsulation may be unable to exploit supporting technologies such as CSS.

D3 is designed to sidestep these problems and complement web standards. Critically, D3 also introduces features that may inform other visualization frameworks: query-driven selection and data binding to scenegraph elements, document transformation as an atomic operation, and immediate property evaluation semantics. In the next section, we describe the design of the D3 system. We then go on to review our design choices and their associated trade-offs in greater detail.

3 Design

D3's atomic operand is the **selection**: a filtered set of elements queried from the current document. **Operators** act on selections, modifying content. **Data joins** bind input data to elements, enabling functional operators that depend on data, and producing **enter and exit** subselections for the creation and destruction of elements in correspondence with data. While operators apply instantaneously by default, animated **transitions** interpolate attributes and styles smoothly over time. Special operators called **event handlers** respond to user input and enable interaction. Numerous helper **modules**, such as layouts and scales, simplify common visualization tasks.

3.1 Selections

D3 adopts the W3C Selectors API to identify document elements for **selection**; this mini-language consists of predicates that filter elements by tag ("tag"), class (".class"), unique identifier ("#id"), attribute ("[name=value]"), containment ("parent child"), adjacency ("before ∼ after"), and various other facets. Predicates can be intersected (".a.b") or unioned (".a, .b"), resulting in a rich but concise selection method.

JavaScript

a
```
var x = function(d, i) { return i * 25; },
    y = function(d, i) { return 160 - d * 80; };
```

b
```
var svg = d3.select("body").append("svg:svg")
    .data([[1, 1.2, 1.7, 1.5, .7, .5, .2]]);
```

c
```
svg.append("svg:path")
    .attr("class", "area")
    .attr("d", d3.svg.area().x(x).y0(160).y1(y));
```

d
```
svg.append("svg:path")
    .attr("class", "line")
    .attr("d", d3.svg.line().x(x).y(y));
```

e
```
var g = svg.selectAll("g")
    .data(d3.range(0, 2, .5))
  .enter().append("svg:g");
```

f
```
g.append("svg:line")
    .attr("class", function(d) { return d ? "minor rule" : "rule"; })
    .attr("x2", 160).attr("y1", y).attr("y2", y);
```

g
```
g.append("svg:text")
    .attr("x", 164).attr("y", y).attr("dy", ".3em")
    .text(d3.format(",.1"));
```

CSS

h
```
svg {
  width: 160px;
  height: 160px;
  font: 10px sans-serif;
  padding: 20px;
}

.area {
  fill: lightblue;
}

.line {
  fill: none;
  stroke: black;
  stroke-width: 1.5px;
}

.rule.minor {
  stroke: white;
}

.rule {
  stroke: black;
}
```

Fig. 5. Specification of the area chart shown in Figure 4. (a) Define scale functions for position encoding. (b) Add an SVG container to the document body and bind data. (c) Add a path element for the area. (d) Add a path element to emphasize the top line. (e) Add containers for reference values. (f) Add reference lines. (g) Add reference labels. (h) Assign colors and other aesthetics with CSS.

The global `d3`, also serving as a namespace, exports `select` and `selectAll` methods for obtaining selections. These methods accept the selector mini-language; the former selects only the *first* element that matches the predicates, while the latter selects *all* matching elements in document traversal order. These methods also accept node references directly, for when nodes are accessed through external means such as a third-party library or developer tool.

Any number of **operators** can be applied to selected elements. These operators wrap the W3C DOM API, setting attributes (`attr`), styles (`style`), properties (`property`), HTML (`html`) and text (`text`) content. Operator values are specified either as constants or functions; the latter are evaluated for each element. While the built-in operators satisfy most needs, the `each` operator invokes an arbitrary JavaScript callback for total generality. Since each selection is simply an array, elements can also be accessed directly (e.g., `[0]`).

D3 supports method chaining for brevity when applying multiple operators: the operator return value is the selection. (For example, the pie chart in Figure 7 is a single statement.) The `append` and `insert` operators add a new element for each element in the current selection, returning the added nodes, thus allowing the convenient creation of nested structures. The `remove` operator discards selected elements.

Whereas the top-level select methods query the entire document, a selection's `select` and `selectAll` operators restrict queries to descendants of each selected element; we call this **subselection**. For example, `d3.selectAll("p").select("b")` returns the first bold ("b") elements in every paragraph ("p") element.

Subselecting via `selectAll` groups elements by ancestor. Thus, `d3.selectAll("p").selectAll("b")` groups by paragraph, while `d3.selectAll("p b")` returns a flat selection. Subselecting via `select` is similar, but preserves groups and propagates data. Grouping plays an important role in the data join (see §3.2), and functional operators may depend on the numeric index of the current element within its group (as in the `x` scale of Figure 5).

3.2 Data

The `data` operator binds input data to selected nodes. D3 uses format agnostic processing [13]: **data** is specified as an array of arbitrary values, such as numbers, strings or objects. Once data is bound to elements, it is passed to functional operators as the first argument (by convention, `d`), along with the numeric index (`i`). These arguments were chosen for parity with JavaScript's built-in array methods, and deviates from Protovis, which supplies extra arguments for any enclosing panel data. This approach simplifies D3's selection structure (requiring only one level of grouping) and avoids variable arguments.

By default, data is joined to elements by index: the first element to the first datum, and so on. For precise control over data-element correspondence, a **key** function [13] can be passed to the `data` operator. Matching key values preserve object constancy across transitions.

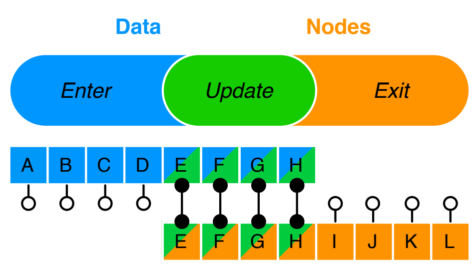

Fig. 6. When new data (blue) are joined with old nodes (orange), three subselections result: *enter*, *update* and *exit*.

If data or elements are leftover after computing the data join, these are available in the **enter** and **exit** subselections, respectively. The *entering* data have no corresponding nodes; the *exiting* nodes have no corresponding data. For example, if data is joined to the empty selection, the `enter` operator returns placeholder nodes for each incoming datum; these nodes can then be instantiated via `append` or `insert`. Similarly, if new data is joined to an existing selection, the `exit` operator returns elements bound to outgoing data to allow removal. In terms of relational algebra, given data D and nodes N, the enter selection is $D \triangleright N$ (left), the exit selection is $N \triangleright D$ (right), and the update selection is $D \bowtie N$ (inner). The updating nodes are simply returned by the `data` operator, convenient for the common case where the enter and exit selections are empty.

The delineation of enter, update and exit allows precise control of the element lifecycle. Properties that are constant for the life of the element are set once on enter, while dynamic properties are recomputed per update. Animated transitions (see §3.3) can be defined for each of the three states. More generally, data joins enable exact data-element correspondence; although this is nonessential for static visualizations, it is crucial for efficient dynamic visualizations.

Data is "sticky"; once bound to nodes, it is available on subsequent re-selection without again requiring the `data` operator. This simplifies subsequent transforms, as well as the implementation of key functions: new data can be compared directly to old data, rather than requiring the data key to be serialized in the document. Data can also be used to reorder (`sort`) or cull elements (`filter`).

3.3 Interaction and Animation

The document object model supports **event listeners**: callback functions that receive user input events targeted at specific elements. D3's `on` operator exposes this functionality for native event types. For consistency with other functional operators, the callback receives the data and index as arguments (`d`, `i`), allowing data-driven interaction. The targeted node is `this`, and the current event is `d3.event`. Listeners may coexist on elements through namespaces (e.g., "click.foo").

D3's focus on transformations simplifies the specification of scene changes in response to user events; the semantics are the same as initialization. Furthermore, **animated transitions** can be derived from selections using the `transition` operator. Transitions export the `style` and `attr` operators of selections with identical syntax, but interpolate from the current to specified value gradually over time. To stagger animation for individual elements, the delay and duration of transitions can be specified as functional operators. Easing can also be customized; standard easing functions [17, 26] such as "elastic", "cubic-in-out" and "linear" are specified by name.

Powering D3's transitions is a collection of **interpolators** for diverse types: numbers; strings with embedded numbers (e.g., font sizes, path data); RGB and HSL colors; and arbitrary nested arrays or objects. If needed, custom interpolators can be specified. An example of such customization is animating value changes in a pie chart; the bound arc data are interpolated in polar coordinates, rather than interpolating the Cartesian coordinates of the path strings.

Transitions dispatch events to registered listeners as each element finishes animating, allowing sequential transitions and post-animation cleanup such as removing exiting elements. Due to staggering, elements may finish at different times. D3 automatically manages transition scheduling, guaranteeing per-element exclusivity and efficient, consistent timing through a unified timer queue. This optimized design easily scales to thousands of concurrent timers.

3.4 Modules

D3's kernel, as described in previous sections, achieves flexibility through representational transparency; this also minimizes the library's conceptual surface area by presenting less to learn. Yet more is needed to alleviate the burden of common tasks. Although we strive to enable custom visualization design, we recognize Tufte's principle [36]: "Don't get it original, get it right." D3's optional modules encapsulate reusable solutions to common problems, increasing efficiency and demonstrating the utility of higher-order programming through functional operators.

As a replacement for the specialized graphical primitives of Protovis, the `d3.svg` module provides various **shapes** suitable for charting. The `arc` function, for example, builds elliptical arcs as for pie and donut charts by mapping arbitrary data to paths; typically this function is bound to the "d" attribute of SVG `path` elements (as in Figure 7). Note that the radii and angles of the arcs can be specified either as constants or as functions—in the latter case, the functions are evaluated per element with access to data—identical to D3's core operators. Thus, helper shapes provide specialized representations without new semantics and without encapsulating the underlying form. Additional shapes are provided for areas, lines, scatterplot symbols, and the like.

Augmenting its interpolators, D3's **scales** simplify visual encoding. These scales are similar to those of Protovis, supporting both ordinal and quantitative (linear, logarithmic, exponential, quantile) values. We have also packaged Cynthia Brewer's useful color scales [10].

Layouts supply reusable, flexible visualization techniques by generating abstract data structures. The `partition` layout, for example, computes a two-dimensional spatial subdivision of a hierarchy; each node has a closed range in x and y. The nodes are bound to arcs for a sunburst [32] ($x \mapsto \theta$, $y \mapsto r$), or rectangles for an icicle tree. The `chord` layout computes an angular partition from a weighted adjacency matrix, enabling radial diagrams in the style of Circos [20]. The `force` layout combines physical simulation and iterative constraint relaxation [7] for stable graph layout. The `stack` layout computes the y_0 baseline for stacked graphs [11, 4], while the squarified `treemap` layout [31, 3] computes another spatial partition well-suited for animation (see §5.1). More layouts are in development.

Interaction techniques are reused through **behaviors**. The `zoom` behavior implements panning and zooming by listening to mouse events; on pan or zoom, a custom event is dispatched to report a two-dimensional translation and scale. This event can be used for either geometric or semantic zooming [27].

Functional operators have surprising depth. For example, the `geo` module exports a `path` operator for projecting geographic data to pixel coordinates. The projection is configurable, such as Albers equal-area (for choropleth and cartograms where area conservation is required), or spherical Mercator for overlaying web-based tile maps. The `path` operator supports the GeoJSON format [34], including boundaries with disconnected areas and holes, as well as centroid and bounding box computation. The `geom` module exports various geometric operators, including Voronoi tessellation, marching squares, convex hulls, polygon clipping and quadtrees.

D3 also includes sundry data-processing utilities, such as `nest` and `cross` operators, a comma-separated values (CSV) parser, date and number formats, etc. These are extremely useful for visualization, but sufficiently distinct that we may bundle them separately in the future. Future work is needed in this area; a rich collection of statistical methods, as in R [35], would be particularly valuable.

4 Design Rationale

D3 is most closely related to our prior work on Protovis [2, 13], a declarative language for visualization design. Although they seek similar goals, D3 and Protovis differ in the type of visualizations they enable and the method of implementation. To put the contributions of D3 in context, we describe our design rationale by focusing on three differentiating factors: implicit or explicit transformation, deferred or immediate evaluation, and access to a native representation. Whereas Protovis excels at concise specifications of static scenes, D3's **transformations** make dynamic visualizations easier to implement. By adopting **immediate evaluation** of operators and the browser's **native representation**, D3 improves compatibility and debugging.

4.1 Transformation

Transformations happen implicitly in Protovis: the data or property definitions are changed, and a call to `render` updates the display by recomputing property values and regenerating the scenegraph. This is convenient but slow; without dependency information, Protovis must re-evaluate *all* properties, even those whose definitions or input data have not changed. In addition, Protovis must then propagate the changes to the intermediate scenegraph out to the native SVG.

In D3, designers specify transformations of scenes (scene changes), as opposed to representations (the scenes themselves). In both cases the specifications are data-driven, but transformations better enable dynamic visualizations through explicit control over which elements are mutated, added or removed, and how so. This eliminates redundant computation, touching only the elements and attributes that need updating, rather than the entire scenegraph.

Explicit transformations naturally extend to animated transitions, where attributes or styles are smoothly interpolated over time. We experimented with transitions in Protovis [13], influencing our design of enter and exit (see §3.2), but its high-level property descriptions make arbitrary scene changes difficult. This is apparent in how Protovis modifies internal state in response to user events, allowing localized changes to the representation. The "magic" local context is convenient

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

```
d3.select("body").append("svg:svg")                          b
    .data([[1, 1.2, 1.7, 1.5, .7]])
    .attr("width", 150)
    .attr("height", 150)
  .selectAll("path")
    .data(d3.layout.pie())
  .enter().append("svg:path")
    .attr("transform", "translate(75,75)")
    .attr("d", d3.svg.arc().outerRadius(70));
```

Fig. 7. A simple pie chart, . (a) Protovis; (b) D3.

for primitive modes of interaction, but not generalizable (for example, one cannot modify elements *other* than the one that received the user event). Additionally, the automatic context is a frequent source of user confusion as it temporarily overrides system behavior.

Transformations have an additional benefit that they can modify existing documents, decoupling manipulation from generation. This could enable a hybrid architecture where visualizations are initially constructed on the server and dynamic behavior is bound on the client.

4.2 Immediate Evaluation

By deferring evaluation of property functions to the rendering phase, Protovis allows implicit re-evaluation of properties. Although convenient, this can cause errors if references captured via closure change (or disappear). For example, a global variable may be inadvertently overwritten by another chart on the same page, remaining undetected until interaction triggers a redraw. This language weakness is exacerbated by pervasive misunderstanding of JavaScript's `var` keyword, which is scoped by function rather than by block, as is typical of other languages. To tighten the scope of reference capture, D3 applies operators immediately; for example, D3's `attr` operator immediately sets attributes on selected nodes and then returns.

Immediate evaluation reduces internal control flow, moving it up to user code. Protovis, in contrast, has hidden control flow that is revealed only when the system crashes—another confusing consequence of delayed evaluation. Immediacy is also more compatible with standard JavaScript organizational constructs, such as functions and loops. Protovis cannot generate arbitrary hierarchical scenegraphs because the hierarchy depth is fixed to the number of nested panels declared in code, whereas D3's stateless evaluation allows transforms to be refactored into functions invoked recursively by the `each` operator.

Internal mechanics complicate the implementation of Protovis layouts, as developers must understand the order in which properties are evaluated and the meaning of specialized, internal callbacks. D3's simplified control flow allows layouts to be decoupled from property evaluation. D3 layouts (see §3.4) are simply helper classes that create or modify arbitrary data structures, such as a description of a chord diagram [20], or positions of nodes in a force-directed graph. The user then binds the layout data to attributes and elements as desired.

4.3 Native Representation

One of the key contributions of Protovis is its choice of graphical primitives, called *marks*. These marks were chosen to satisfy the needs of common chart types: `Line` for line charts, `Wedge` for pie charts, and so on. Protovis achieves greater expressiveness than charting libraries because these simple shapes can be composed in various ways.

Protovis marks are intentionally homogeneous: properties have the same meaning across mark types. This enables *prototypal inheritance*, where derived marks reuse property definitions from existing marks, reducing verbosity. It also facilitates iterative design, as mark types can be changed without breaking related code. Related marks such as text labels are easy to specify in Protovis using *anchors*.

Abandoning a specialized representation for a standard one, such as SVG, relinquishes these advantages. For example, inheritance is not appropriate for SVG's heterogeneous shapes (e.g., attributes "cx" for circles *vs.* "x" for rectangles). On the other hand, the native representation supports CSS for sharing simple property definitions, and has advantages as previously discussed including interoperability, documentation and expressiveness.

Some of the benefits of specialization can be recovered through simple helper classes, without the cost of encapsulation. D3's `arc` class allows the convenient specification of pie chart wedges using SVG `path` elements and elliptical arc path segments (as in Figure 7). The output is identical to the Protovis `Wedge`, except native elements improve tool compatibility and debugging. However, we note this decoupling does incur a slight decrease in notational efficiency.

Finally, a subtle yet significant advantage of native representation is that selections can be retrieved from the document at any time. In order to modify a Protovis visualization, one needs to modify the underlying data or property definitions, and then redraw. This requires bookkeeping (e.g., `var`) for affected marks in the scene. Shared property definitions make it difficult to change specific marks—such as the mark under the mouse cursor—without global side-effects. D3, in contrast, uses selectors to identify document elements through a variety of means (such as tag names, class attributes, and associated data), making local modifications trivial. Since selections are transient, they can also overlap for greater flexibility than single inheritance.

5 Example Applications

Over the course of development, we have built numerous visualizations with D3, including real-world applications, tests of the framework's capability, and pedagogical examples for new users. We now describe several example applications to convey representative usage and unique capabilities. For brevity, full source code is not included but is available online [6].

5.1 Animated HTML Treemaps

Using the `treemap` layout, we created a squarified treemap of classes in the Flare [9] package hierarchy. Each node is mapped to a rectangular HTML `div` element; although HTML is less expressive than SVG, it is supported on older browsers and demonstrates the framework's flexibility. The x, y, Δx and Δy computed by the layout are mapped to positional styles. For example, the `style` operator for "left" is defined as `function(d) { return d.x+"px"; }`. Similarly, the ordinal color scale `d3.scale.category20` sets the background color of nodes by package, while the `text` operator generates labels for class names.

Two area encodings are specified via the `value` operator on the layout: by file size (`d.value`), and by file count (`1`). Thus, in the latter case, each leaf node has equal size. Buttons with click event handlers toggle between the two encodings, initiating animated transitions. The treemap layout is configured "sticky", such that the allocation of nodes into squarified horizontal and vertical rows is preserved across updates; this allows nodes to be resized smoothly, without shuffling or occlusion that would impede perception of changing values. Although this results in a suboptimal layout for one of the two states, the results are acceptable (see Figure 2). If desired, one could extend the layout to compromise multiple states (by averaging values prior to layout), or, a sequenced animation could resize and then reorient.

The static treemap is 21 lines of JavaScript—a negligible increase over the 17 lines required for Protovis. Adding interaction and animation expands the specification to 54 lines.

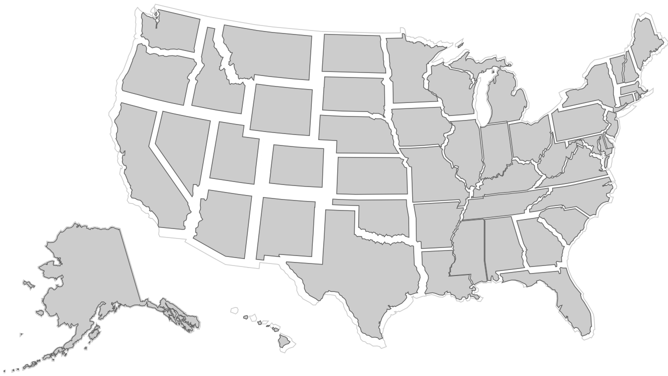

Fig. 8. Noncontiguous cartogram of obesity rates (BMI $\geq$ 30) made with D3. Values range from 10.0% (Colorado) to 20.1% (Indiana).

5.2 Noncontiguous Cartograms

D3's `geo` module simplifies the specification of geographic visualizations. To demonstrate this, we built a noncontiguous cartogram [25] that encodes values for geographic regions as area by scaling each region around its projected centroid.

A discontinuous Albers equal-area projection shows the 48 states, Hawaii and Alaska as recommended by the USGS. The state boundaries are loaded asynchronously as GeoJSON, and then mapped to SVG `path` elements using the `geo.path` operator. The state boundaries were previously downloaded from the U.S. Census Bureau, simplified via MapShaper, and converted to GeoJSON using GDAL.

Three copies of the states are generated: one in the background with a thick gray stroke for a halo, one in the middle with white fill to mask internal strokes, and one in the foreground to encode data. The gray halo effect for the country outline is helpful to assist viewers in perceiving the distortion of area.

The states are scaled around their centroids using SVG's "transform" attribute. To scale around a position other than the origin, multiple transforms are concatenated: "translate(x, y) scale(k) translate($-x$, $-y$)". The x and y values are computed by the `centroid` method of the `path` operator, while k is proportional to the square root of the input value—here the obesity rate reported by the CDC, as of 2008. To minimize overlap on adjacent states, $k \leq 1$.

This example requires 34 lines of JavaScript, not including data or comments. Stroke widths and colors are specified using CSS.

5.3 Bézier Curves Explained

D3 is not limited to standard data visualizations; mapping arbitrary data to DOM elements enables a wide variety of data-driven graphics. An interesting example comes from D3 contributor Jason Davies, who designed a tutorial on the construction of parametric Bézier curves. The tutorial is both animated and interactive: as the parameter t animates from 0 to 1, control points can be moved to affect the curve. The intermediate interpolation steps are shown as colored spans (yellow for quadratic, blue for cubic and green for quartic).

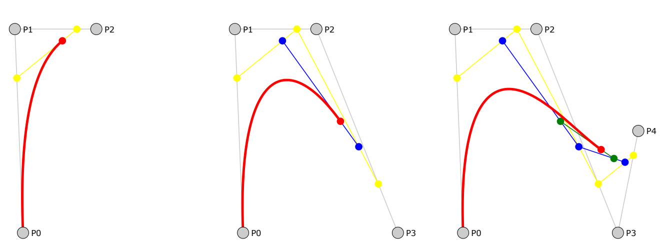

Fig. 9. Visual explanation of Bézier curve interpolation by Jason Davies. From left to right: quadratic, cubic, and quartic curves with t=0.76.

SVG `path` elements display the curves, lines connect the control and interpolation points, and `circle` elements show the control points. Event handlers on the circles respond to mouse events to allow drag-and-drop. The backing data is an array of five control points (in x and y); slices of this array generate small multiples for lower-order curves. Thus, moving a control point in one curve simultaneously updates the corresponding control point in the others. The red path is a piecewise linear discretization of the partial Bézier curve for the current t. The path is incrementally constructed as t advances, and cached to optimize performance.

This example is 139 lines of JavaScript, not including comments. Some styles are set with CSS while others are set from JavaScript.

6 Performance Benchmarks

By using explicit transformations of a native representation, D3 can avoid unnecessary computation (transformations can be limited to selected attributes) and reduce overhead (the DOM is modified directly, eliminating the indirection of an intermediate scenegraph). These design decisions improve performance compared to a higher-level framework such as Protovis. We now substantiate this claim through a pair of performance benchmarks comparing equivalent visualizations constructed with D3 and Protovis.

In addition, much recent fanfare concerns the increasing graphical and interactive capabilities native to modern web browsers (typically under the mantle of "HTML5"). Previously, designers relied upon proprietary plug-ins, namely the Adobe Flash Player, to provide interactive graphics. To assess the current state-of-the-art, we include Flash-based visualizations in our benchmarks.

6.1 Methods

We compared initialization times and frame rates for D3, Protovis, and Flash using two visualizations: an interactive scatterplot matrix supporting brushing and linking [1] across four dimensions and an animated stacked graph [37]. We thus compared a total of six different visualization designs. Figure 11 shows examples of the two visualization types. Both benchmark metrics are important for web-based visualization: longer page load times have been shown to increase user attrition [22], while a sufficient frame rate is necessary for fluent interaction and animation.

We simulate interaction to benchmark the scatterplot matrix. On each frame we randomly select a constituent plot and two coordinates within it; these coordinates define a rectangular selection region for brushing and linking. In response, each scatterplot highlights the points contained within the selection. Both D3 and Protovis render points using SVG `circle` elements. Within Flash, we represent each point with its own underlying Flash `Sprite` object. Improved Flash rendering performance is possible by rendering multiple points within a single `Sprite`. However, this complicates mouse event processing for single points—one has to implement hit testing manually. SVG provides event callbacks for all shape elements. To provide a fair comparison, we maintain similar functionality across platforms.

For the stacked graph, we continuously animate between two fixed sets of data values. The D3 and Protovis implementations use the stacked graph layout routines bundled with each framework. The Protovis instance uses animated transition support introduced in version 3.3. In Flash, we use the stacked graph layout and animation support provided by the Flare toolkit [9].

For each visualization we measure both the initialization time (the time from page load to initial display of the visualization) and average frame rate. Initializations were repeated ten times and averaged. As the visualization cycles through simulated interactions or completed animations, we record average frame rates for 10-second intervals over a period of 5 minutes. We then compute the mean and variance of frame rate samples. We repeat this analysis over an increasing number of data points ranging from 500 to 20,000. In all cases the standard deviations are smaller than the means by 1–2 orders of magnitude, and so we omit them presently.

We performed our benchmarks on a MacBook Pro with a dual-core 2.66 GHz processor and 8GB RAM running MacOS X 10.6.7. We

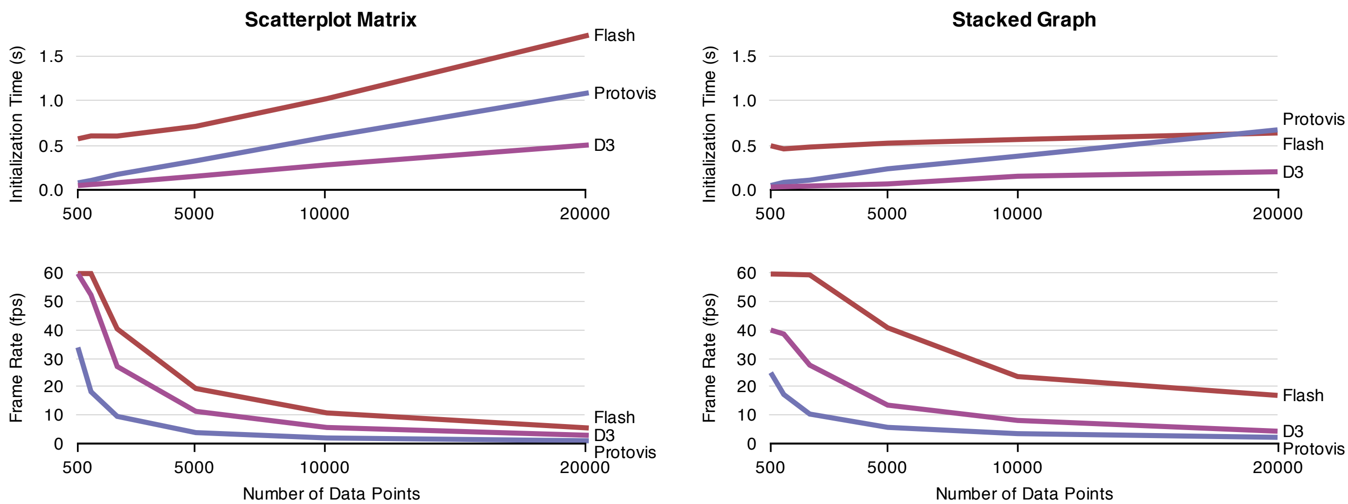

Fig. 10. Performance benchmarks. Initialization times (top) and frames rates (bottom) for a scatterplot matrix (left) and stacked graph (right).

conducted the benchmarks inside the Google Chrome browser version 11.0 beta with the Adobe Flash Player 10.2 plug-in.

6.2 Results and Discussion

Figure 10 presents our benchmarking results. For both visualizations, the initialization time from page load to visualization view is typically faster for browser-native tools. D3 results in significantly faster page loads: twice as fast as Protovis and over three times as fast as Flash. Presumably this discrepancy is due to initialization of the Flash plug-in. As we calculate load times by triggering a browser refresh, our results take into account time savings due to caching previously-loaded Flash libraries. We also note that our Flash visualizations do not make use of an application framework such as Adobe Flex; doing so further increases load times by over a second.

With respect to frame rate, Flash provides the best performance. As the number of data points increases, Flash-based visualizations exhibit 2–2.5 times more frames per second than D3. Meanwhile, D3 shows improved scalability among browser-native tools, exhibiting at least double Protovis' frame rate as the data set size increases. This matches performance gains "from 30 to around 90" frames per second reported by Davies, who previously implemented the Bézier curve tutorial (see §5.3) in Protovis. Others have similarly observed "much faster" performance in D3.

Moreover, our comparison to Protovis is conservative, as in our benchmarks the majority of the scene must be redrawn on each frame. This provides a useful bound on performance, but obscures the common case of more localized updates. By limiting updates to the changing parts of a scene, D3 transforms provide greater scalability than Protovis. D3 also allows more control over document structure, allowing further optimization; for example, SVG's `use` element efficiently replicates shapes, while CSS3 provides hardware acceleration of certain animated transitions.

Our results confirm that D3's use of explicit transformations and native representation deliver improved performance: page load times and frame rates in D3 outperform Protovis by at least a factor of two. D3 visualizations load at least three times faster than equivalent Flash-based examples. However, our results also indicate that browser vendors still have some distance to cover in improving SVG rendering performance. Flash provides consistently higher frame rates as the number of data points increases.

7 Feedback and Observations

We (and our users) have solved diverse visualization tasks using D3 that would be difficult or impossible with Protovis. Examples include pure HTML displays (§5.1), multi-stage animations [15], and interactive visualizations of complex structures, including a force-directed graph with "expand-on-demand" clusters and convex hulls around leaf nodes (see Figure 1). The ease with which transitions can be implemented is also evident. One designer chose D3 for a recent visualization contest, highlighting the emotional impact of dynamic graphics: "These transitions are amazing! Just playing around with them gives such great effects and inspiration for more."

While we can quantify performance, accessibility is far more difficult to measure. The true test of D3's design will be in user adoption; initial feedback has been positive. A Protovis expert writes, "The transformations are actually very easy to work with, perhaps even more simple than in Protovis. It's very straightforward." However, one user found the learning curve "much steeper than Protovis", while another writes, "It took me a little while to get my head around your interface and general philosophy, but that process has given me valuable insights into the nature and meaning of our data." Part of the issue may be the complexity of the SVG specification: "The key [to] learning D3 at this stage seems to be to study the SVG spec, and to *inspect the SVG generated by D3* [emphasis added]. After a few iterations it all begins to make perfect sense." Users thus appreciate compatibility with developer tools.

We also find that post-hoc manipulation of visualizations through the developer console is a unique and compelling benefit of D3's design. Using "sticky" data, elements can be selected and new operators applied to change appearance or behavior. This facilitates rapid iteration: for example, we adjusted the color scale of one user's chart to improve differentiation (Figure 4), and added event listeners to another's to coordinate views. Combined with the ability to view source on any visualization, we have high hopes for D3's collaborative potential.

By building on key standards, D3 keeps pace with the evolving technological ecosystem of the web, improving *expressiveness* and *accessibility*. We believe D3 is well-positioned to let designers immediately take advantage of new browser features as they are added. While work remains to expand our collection of specialized modules, D3's core language provides an *efficient* foundation for specifying rich, dynamic visualizations on the web.

Acknowledgments

The authors thank Nelson Minar, Jason Davies, Vijay Chakravarthy, Jan Willem Tulp and Tom Carden for their insight and support. This work was partially supported by The Boeing Company.

References

[1] R. A. Becker and W. S. Cleveland. Brushing scatterplots. *Technometrics*, 29:127–142, May 1987.

[2] M. Bostock and J. Heer. Protovis: A graphical toolkit for visualization. *IEEE Trans Vis and Comp Graphics*, 15(6):1121–1128, 2009.

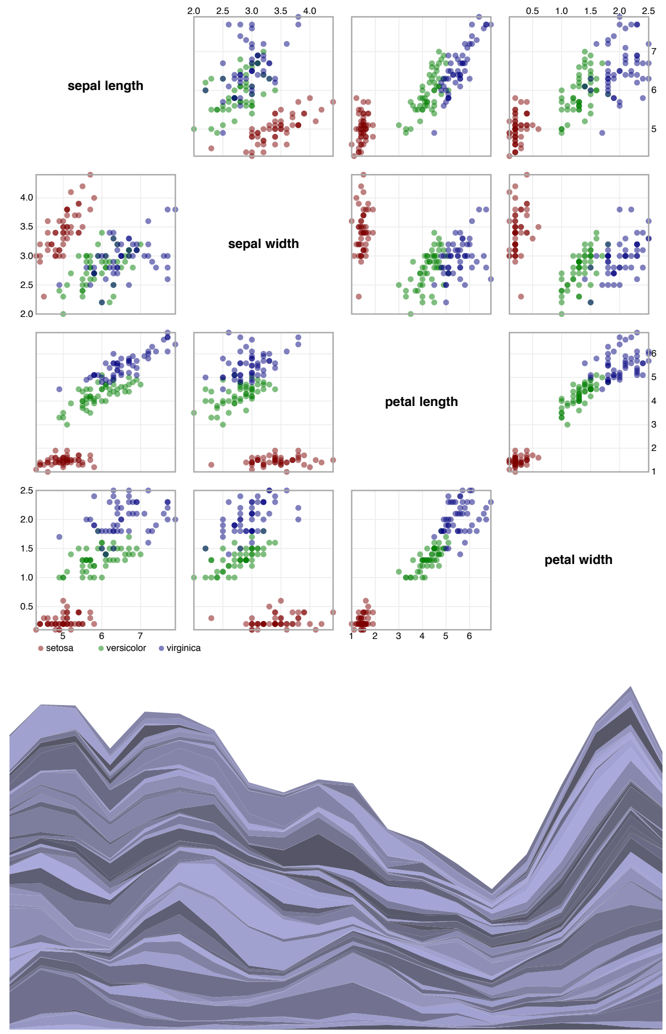

Fig. 11. Visualizations used in our benchmarks. (a) Scatterplot matrix with brushing & linking. (b) Animated stacked graph.

[3] D. M. Bruls, C. Huizing, and J. J. van Wijk. Squarified treemaps. In *Joint Eurographics and IEEE TCVG Symposium on Visualization*, pages 33–42, 1999.
[4] L. Byron and M. Wattenberg. Stacked graphs – geometry & aesthetics. *IEEE Trans. Vis. and Comp. Graphics*, 14(6):1245–1252, 2008.
[5] S. K. Card, J. D. Mackinlay, and B. Shneiderman, editors. *Readings in information visualization: using vision to think*. Morgan Kaufmann, San Francisco, CA, 1999.
[6] d3.js. `http://mbostock.github.com/d3/`, Mar 2011.
[7] T. Dwyer. Scalable, versatile and simple constrained graph layout. In *EuroVis*, 2009.
[8] J.-D. Fekete. The InfoVis Toolkit. In *IEEE InfoVis*, pages 167–174, 2004.
[9] Flare. `http://flare.prefuse.org`, Mar 2011.
[10] M. A. Harrower and C. A. Brewer. Colorbrewer.org: An online tool for selecting color schemes for maps. *The Cartographic Journal*, 40:27–37, 2003.
[11] S. Havre, B. Hetzler, and L. Nowell. ThemeRiver: Visualizing theme changes over time. In *IEEE InfoVis*, page 115, 2000.
[12] J. Heer and M. Agrawala. Software design patterns for information visualization. *IEEE Trans Vis and Comp Graphics*, 12(5):853–860, 2006.
[13] J. Heer and M. Bostock. Declarative language design for interactive visualization. *IEEE Trans Vis and Comp Graphics*, 16(6):1149–1156, 2010.
[14] J. Heer, S. K. Card, and J. A. Landay. prefuse: a toolkit for interactive information visualization. In *Proc. ACM CHI*, pages 421–430, 2005.
[15] J. Heer and G. G. Robertson. Animated transitions in statistical data graphics. *IEEE Trans Vis and Comp Graphics*, 13(6):1240–1247, 2007.
[16] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28:196, December 1996.
[17] S. Hudson and J. T. Stasko. Animation support in a user interface toolkit: Flexible, robust, and reusable abstractions. In *ACM UIST*, pages 57–67, 1993.
[18] JavaScript InfoVis Toolkit. `http://thejit.org/`, Mar 2011.
[19] jQuery. `http://jquery.com/`, Mar 2011.
[20] M. Krzywinski, J. Schein, I. Birol, J. Connors, R. Gascoyne, D. Horsman, S. J. Jones, and M. A. Marra. Circos: An information aesthetic for comparative genomics. *Genome Research*, 19(9):1639–1645, Sep 2009.
[21] H. W. Lie. *Cascading Style Sheets*. PhD thesis, University of Oslo, 2005.
[22] M. Mayer. Google I/O keynote. `http://www.youtube.com/watch?v=6x0cAzQ7PVs`, 2008.
[23] MooTools. `http://mootools.net/`, Mar 2011.
[24] D. A. Norman. *The Psychology of Everyday Things*. Basic Books, New York, NY, 1988.
[25] J. M. Olson. Noncontiguous area cartograms. *The Professional Geographer*, 28:371–380, November 1976.
[26] R. Penner. Robert penner's easing equations. `http://www.robertpenner.com/easing/`, Mar 2011.
[27] K. Perlin and D. Fox. Pad: an alternative approach to the computer interface. In *ACM SIGGRAPH 93*, pages 57–64, 1993.
[28] Processing.js. `http://processingjs.org`, Mar 2011.
[29] Prototype. `http://www.prototypejs.org/`, Mar 2011.
[30] Raphaël". `http://raphaeljs.com/`, Mar 2011.
[31] B. Shneiderman. Tree visualization with tree-maps: 2-d space-filling approach. *ACM Transactions on Graphics*, 11:92–99, 1992.
[32] J. Stasko and E. Zhang. Focus+context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations. In *IEEE InfoVis*, pages 57–64, 2000.
[33] The Dojo Toolkit. `http://dojotoolkit.org/`, Mar 2011.
[34] The GeoJSON Format Specification. `http://geojson.org/`, Jun 2008.
[35] The R Project for Statistical Computing. `http://www.r-project.org/`, Mar 2011.
[36] E. T. Tufte. Graphing software. `http://www.edwardtufte.com/bboard/q-and-a-fetch-msg?msg_id=00000p`, 2001.
[37] M. Wattenberg. Baby names, visualization, and social data analysis. In *IEEE InfoVis*, pages 1–7, 2005.
[38] C. E. Weaver. Building highly-coordinated visualizations in Improvise. In *Proc. IEEE InfoVis*, pages 159–166, 2004.
[39] H. Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer, 2009.
[40] L. Wilkinson. *The Grammar of Graphics (Statistics and Computing)*. Springer-Verlag, Secaucus, NJ, 2005.
[41] XSL Transformations. `http://www.w3.org/TR/xslt`, Nov 1999.