Declarative Language Design for Interactive Visualization

Jeffrey Heer and Michael Bostock

Abstract—We investigate the design of declarative, domain-specific languages for constructing interactive visualizations. By separating specification from execution, declarative languages can simplify development, enable unobtrusive optimization, and support retargeting across platforms. We describe the design of the Protovis specification language and its implementation within an object-oriented, statically-typed programming language (Java). We demonstrate how to support rich visualizations without requiring a toolkit-specific data model and extend Protovis to enable declarative specification of animated transitions. To support cross-platform deployment, we introduce rendering and event-handling infrastructures decoupled from the runtime platform, letting designers retarget visualization specifications (e.g., from desktop to mobile phone) with reduced effort. We also explore optimizations such as runtime compilation of visualization specifications, parallelized execution, and hardware-accelerated rendering. We present benchmark studies measuring the performance gains provided by these optimizations and compare performance to existing Java-based visualization tools, demonstrating scalability improvements exceeding an order of magnitude.

Index Terms—Information visualization, user interfaces, toolkits, domain specific languages, declarative languages, optimization.

1 Introduction

Declarative languages often simplify programming tasks by requiring that a developer specify *what* the results of a computation should be rather than *how* the results should be computed. The separation of specification from execution allows language users to focus on the specifics of their application domain, while freeing language developers to optimize processing. For example, mark-up languages such as HTML and CSS have enabled millions of novice programmers to develop web pages, while database query languages such as SQL and MDX insulate database users from the specifics of query planning and execution. In contrast, most information visualization toolkits (e.g., [6, 7, 9, 10, 23]) adhere to an imperative programming model that requires visualization designers to contend with software engineering concerns, particularly when creating novel or customized graphics [1].

Moreover, contemporary visualization design tools must address a number of new technical challenges. Not least among these is the increasing heterogeneity of commodity hardware and interactive devices. Visualization tools should ideally support interfaces ranging from traditional desktop applications, to browser-based web clients, to multi-touch mobile devices. Furthermore, visualization tools should effectively capitalize on hardware trends such as multi-core computing and specialized graphics hardware. While point designs exist for each of these areas, the field currently lacks a consistent approach to visualization design and deployment across heterogeneous platforms.

To address these issues, we argue for a break with current component model architectures and instead advocate the design of declarative, domain-specific languages for interactive visualization. Our prior work on Protovis [1] — an embedded domain-specific language (DSL) for web-based visualization in JavaScript — has demonstrated that a declarative language can simplify visualization specification while supporting a high degree of expressiveness and customization. In this paper, we extend this line of work and introduce an implementation of Protovis in the Java programming language. The system is the result of an exploration of implementation strategies for declarative visualization languages, and supports a diversity of data types, multiple hardware devices, varied graphics and interaction infrastructures, and platform-specific performance optimizations. More specifically, we seek to address the following design goals:

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

Manuscript received 31 March 2010; accepted 1 August 2010; posted online 24 October 2010; mailed on 16 October 2010.
For information on obtaining reprints of this article, please send email to: tvcg@computer.org.

Declarative language design and implementation. In practice, designers must choose between many visualization systems, balancing trade-offs among *expressiveness* ("Can I build it?"), *efficiency* ("How long will it take?") and *accessibility* ("Do I know how?"). In previous work [1], we designed Protovis to balance these concerns and facilitate visualization design. In the current work, we demonstrate how the Protovis design can be implemented within a statically-typed, object-oriented programming language through a sequence of specification, property binding, property evaluation, and rendering stages. We also contribute extensions to the Protovis language design to enable declarative specification of expressive animated transitions.

Cross-platform deployment. Everyday computing increasingly involves a diversity of devices, creating a corresponding challenge for visualization tools to support heterogeneous platforms. However, most visualization tools today target a single platform. This is unfortunate, as a designer's investment in a particular tool may not transfer to other computing environments. In response, we introduce a rendering and event-handling infrastructure that abstracts above the host windowing system. Our system supports a variety of renderers (e.g., OpenGL via JOGL, Java2D, and Android for mobile devices) and interaction paradigms (e.g., mouse-based or touch-based interaction). As a result, developers can create renderers or event dispatchers for new platforms as needed, letting designers retarget their visualization specifications across platforms with reduced effort.

Format agnostic processing. Most visualization systems [6, 7, 9, 10, 23] require developers to corral their data into a toolkit-specific data representation. As a result, programmers may have to map their domain-specific representations to what the toolkit demands. Data adhering to an unsupported format (e.g., semi-structured data) must be suitably transformed, often consuming resources due to data replication. In practice we have found that modifying a visualization specification to suit a new data set is often easier than reconfiguring the data to fit a specific toolkit. Protovis allows designers to mould the visualization specification to the data using anonymous functions to extract data properties. We demonstrate how this approach, more representative of dynamically-typed functional languages, can be applied within a statically-typed language such as Java, thereby allowing any arbitrary, iterable collection of objects to drive a visualization.

Optimization. By decoupling specification from implementation, developers can implement language optimizations without interfering with the work of designers. We investigate optimization techniques including runtime-compilation of visualization specifications, multi-threaded parallel evaluation and rendering, and hardware-accelerated graphics. We present benchmark studies measuring the performance gains provided by these optimizations and compare Protovis performance to prefuse [10], a Java-based visualization API, demonstrating scalability improvements exceeding an order of magnitude.

2 Motivation and Related Work

An array of options exists for creating interactive visualizations. Standard charting programs such as Excel and online tools such as ManyEyes [22] utilize a *chart typology* [25]: users select from a palette of pre-defined visualizations. While often easy to use, such systems do not permit novel designs or customization. At the other extreme, a programmer may use a graphics API such as OpenGL, Processing or Adobe Flash. With these low-level tools the task of creating a new visualization typically requires a significant software engineering effort in an imperative programming style.

Falling somewhere in-between are visualization frameworks such as the InfoVis Toolkit [6], Improvise [23], prefuse [10], and Flare [7]. The former [6, 23] provide a class hierarchy of visualization widgets and new visualizations are introduced either by subclassing an existing component or creating a new one. The latter [10, 7] build visualizations using composable operators for tasks such as data transformation, layout, and visual (color, shape, size) encodings. This subdivision enables construction of custom visualizations using lower-level building blocks, similar to data-flow systems in scientific visualization [15]. However, we have observed that in practice many novel visualizations require programmers to author completely new operators. Recent frameworks also introduce optimizations such as interruptible multi-threaded execution [19] and GPU processing [16]. Unfortunately, these approaches impose a burden on developers, who must contend with increasingly complex APIs.

An alternative approach is to formulate a *declarative, domain specific language* (DSL) [17] for visualization design. Our hypothesis is that by allowing designers to specify visualizations directly in terms of data-representative graphical marks, we can simplify construction while preserving an expressive design space. By decoupling specification from execution, a declarative approach can unobtrusively support performance optimization and retargeting across runtime platforms.

We take inspiration from existing declarative DSLs such as HTML/CSS [14] and SQL; they are used by millions and insulate users from platform and optimization issues. However, designing a standalone language is a difficult task [17] and may complicate integration, as visualizations are often used as components within larger applications. Thus, we believe that the most promising approach is the design of *embedded* DSLs [12]. By implementing a DSL for visualization within a host programming language, designers can use familiar syntactic constructs, leverage the capabilities of the host language, and integrate with other projects. Already, this approach has proven popular in the functional programming community (e.g., [5, 12]) and is used to facilitate massively parallel programming [2, 17].

We are not the first to note the benefits of declarative languages for visualization: researchers have introduced myriad languages for visual analysis. Examples include Wilkinson's Grammar of Graphics [25], Wickham's ggplot2 [24], Slingsby et al.'s HiVE [20] language for hierarchical layouts, and the VizQL formalism of Tableau and Polaris [21]. These languages provide a high level of abstraction and support rapid analysis, but do not provide fine-grained control over graphics and interaction. We target an intermediate level of abstraction that permits a wide array of custom visual designs, embeds within a host language, and yet avoids the tedium of low-level graphics tools.

Others have advanced arguments for this style of declarative visualization language. Cottam et al. [3] describe a declarative language for "point-implantation" graphics such as scatter plots and star maps, and note that it accelerated iterative development. They later generalize their approach in Stencil [4], a visualization model that shares some commonalities with Protovis. For example, their *Layer* and *Glyph* abstractions are similar in respects to Protovis *Panels* and *Marks*. However, they describe an abstract model only, and do not include implementation or optimization details. Duke et al. [5] introduce DSLs for scientific visualization embedded in the Haskell programming language, and argue that information visualization might benefit from a similar strategy. In the current work, we investigate the design and implementation of declarative languages for interactive information visualization, similarly realized as embedded DSLs that utilize functional programming techniques.

3 Language Design

Protovis takes a graphical approach to data visualization, composing custom views of data with simple graphical primitives like bars and dots. These primitives are called **marks**, and each mark encodes data visually through dynamic **properties** such as color and position. In essence, a mark is simply a collection of bound properties for an associated graphical form. Figure 1 shows these graphical primitives.

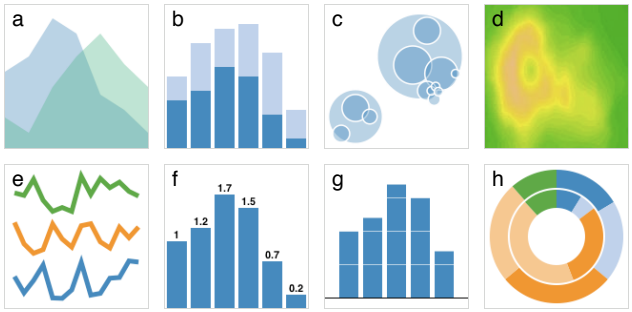

Fig. 1: Examples of primitive mark types. (a-h) **Area**; **Bar**; **Dot**; **Image**; **Line**; **Label** and Bar; **Rule** and Bar; **Wedge**.

Marks are associated with data: a mark is generated once per associated datum, mapping the datum to visual properties. 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.

Although marks are simple by themselves, one can combine them to make rich, interactive visualizations. To facilitate this, Protovis supports **panels** and **inheritance**. A panel is a container for marks; the contained marks are replicated for each datum on the panel. 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. Most marks share the same basic properties for consistency and to facilitate inheritance.

Prototypal inheritance in Protovis also enables a "mix-in" functionality: a mark can directly **extend** another, effectively inserting an additional prototype mark in the inheritance hierarchy. This provides a convenient abstraction and extensibility mechanism. For example, one may consolidate property definitions by defining a generic mark as a set of shared properties. Other marks can then simply extend the generic mark to inherit those properties outside of the standard inheritance tree. Alternatively, a layout algorithm can interface with Protovis using a mark instance with bound **left** and **top** properties. An existing mark can extend the layout mark to inherit the position properties.

We provide additional details, including evaluations of the language's expressiveness and usability, elsewhere [1]. We now focus on novel extensions to the language design and implementation.

3.1 Protovis-Java

Our first implementation of the Protovis language model [1] uses JavaScript and SVG (scalable vector graphics) to enable web-based visualizations. As a dynamically-typed functional language, JavaScript naturally lends itself to the task of defining mark properties as anonymous functions, and the system remains under active development. However, while in-browser performance has improved dramatically in recent years, it currently still falls short of the performance possible within traditional compiled programs. Designers may also want to include visualizations as components within larger applications.

In response, we are also developing an implementation of Protovis in the Java programming language. The system is a research prototype for exploring the potential of Protovis' declarative approach within a statically-typed object-oriented language, including optimization techniques that are not yet applicable within a browser. We chose Java

because it enables cross-platform deployment (including mobile devices), supports myriad rendering options, and includes standardized multi-threading support. However, the choice of Java does introduce limitations; we discuss these at the end of the paper.

3.2 Visualization Specification

The Protovis-Java specification for a basic bar chart is shown in Figure 2. The specification takes a collection of `Point2D` objects and encodes the y-component as a bar chart. An `index` value denoting the current position in the input data list determines the horizontal position. The specification is quite similar to its JavaScript equivalent, with properties specified using a method-chaining convention: each property setter returns the mark instance. However, the specification does contain two notable departures: the use of string literals to define anonymous functions and the **datatype** property.

Java does not support anonymous *functions* directly; designers must create a new *class* adhering to a known interface. Protovis provides a `Property` interface to support this form of specification. However, writing a new class definition for each property is tedious and the extra boilerplate code hampers legibility. Instead, designers can define anonymous property functions using string literals: Java code defining the property, bracketed by double curly braces. These dynamic properties are subsequently interpreted or compiled. Associated values, such as the current `data` and `index`, are in scope by default and `return` statements are omitted. Otherwise, these functions use the syntax and semantics of regular Java code.

Unlike other visualization systems, Protovis places few constraints on the input data—each mark must simply be provided with an `Iterable` collection. Thus any arbitrary collection of objects can drive a visualization. By defining suitable property functions, data elements can be structured objects or even nested data structures. The optional **datatype** property specifies the Java class type of the input data. The property enables runtime type checking and removes the need for manual typecasts within anonymous functions. If left unspecified, data are treated as top-level `Object` instances.

3.3 Interaction

Figure 3 extends our bar chart example to highlight elements on mouse hover. The specification creates a new bound variable (`i`) that stores the index of the active element; the fill color is selected in response to this value. Event handlers are defined similarly to visual properties, also permitting anonymous functions specified as string literals. An additional `event` variable, accessible within the callback's scope, provides access to details such as the key pressed. Event objects are specific to Protovis, decoupled from the windowing system.

3.4 Animation

Our Java system also contributes a novel design feature: declarative specification of animated transitions. Protovis provides keyframe animation by first computing starting and ending values and then interpolating properties. By default, existing property settings determine starting values and an update computes new target values. Animated transitions are requested by providing a duration parameter to the `update` method. The update method returns a `Transition` instance; invoking the `play` method commences the animation. Multiple transitions can be concatenated to create staged animations.

We introduce new properties to support animation. The **key** property determines the correspondence between data elements across updates: identical key values imply object constancy. By default, the datum itself is used as the key. The **ease** property accepts an easing function [13] to control the interpolation rate per-element, e.g., to provide slow-in slow-out pacing. The **delay** property adjusts the onset time for an item's animation, enabling pauses or staggered animation [11].

The aforementioned properties are sufficient for scenes in which the underlying data does not change. Additional mechanisms are needed to specify animation behavior when data elements are added or removed; for example, to fade items in and out. The **enter** and **exit** methods each take a collection of properties to define the state of entering and exiting elements. Syntactically, these methods mirror the

```
List<Point2D> data; // list of data points
Scene vis = new Scene().width(w).height(h).scene();

Mark bar = vis.add("Bar")
    .data(data).datatype(Point2D.class)
    .left("{{index * 5}}") // x-coord by data sort order
    .height("{{data.getY()}}") // y-coord by Point2D y value
    .bottom(0)
    .width(3);

vis.update();
```

Fig. 2: A simple bar chart visualization backed by Point2D objects.

```
bar.def("i", -1)
    .fill("{{Fill.solid(i == index"
          + "? 0xff0000 : 0x1f77b4)}}")
    .mouseover("{{i(index).update()}}")
    .mouseout("{{i(-1).update()}}");
```

Fig. 3: Adding interaction. Bars highlight red on mouse hover.

```
// new elements grow out of the baseline
Mark enter = new Mark()
    .left("{{item.left + 5}}") // begin right-shifted
    .bottom(0).height(0).width(0);

// exiting elements shrink to the baseline
Mark exit = new Mark()
    .left("{{item.left - 3}}") // finish left-shifted
    .bottom("{{item.bottom}}").height(0).width(0);

// add animation properties to bars
bar.key("{{data.hashCode()}}") // key by hash code
    .delay("{{0.005 * index}}") // stagger onset
    .ease(Easing.Poly(2.2)) // slow-in, slow-out pacing
    .enter(enter) // on enter
    .exit(exit); // on exit

vis.update(1).play();
```

Fig. 4: Adding animated behaviors to the bar chart.

```
// new elements drop in "from the sky"
Mark enter = new Mark()
    .left("{{item.left + 5}}")
    .bottom(h);

// exiting elements "blow up" as they fade out
Mark exit = new Mark()
    .alpha(0) // fade out
    .left("{{item.left - 5}}")
    .bottom("{{item.bottom - 5}}")
    .width("{{item.width + 10}}")
    .height("{{item.height + 10}}");
```

Fig. 5: Alternative animation behaviors.

extend method: they accept a `Mark` instance containing a set of properties to invoke upon entrance or exit from the scene. The default enter and exit behavior sets the **alpha** property to zero (fully transparent), causing new items to fade in and exiting items to fade out.

This mechanism enables an expressive range of animation designs. In Figure 4, new bars spring up from a height of zero, while exiting bars shrink down to the baseline. In Figure 5, new bars fall from the "sky", while exiting bars "blow up" in size while fading out, similar to a video game. Though perhaps not representative of best visualization practices, these examples help illustrate the variety of possible animated behaviors. As shown in Figure 8, the **enter** and **exit** properties also support effects such as fading axis gridlines in and out during a scale change [11]: one need only keep a reference to the prior scale settings in order to compute the entering position of new gridlines.

```
// [[3,4,5,3], [3,5,1,2]]
List<List<Integer>> data;

Scene vis = new Scene();
vis.add("Panel")
    .data(data)
    .height(50)
   .add("Line")
    .datatype(int.class)
    .left("{{index * 50}}")
    .bottom("{{data * 10}}")
    .stroke(Stroke.solid(1,
            0x3a68a4))
   .add("Dot");

vis.update();
```

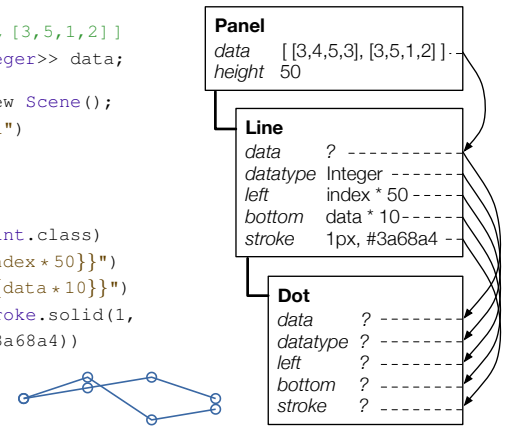

Fig. 6: Left: Specification of a line chart with multiple series. Right: Mark hierarchy with arrows indicating the results of property binding. Property values for the **Dot** mark are inherited from the **Line** mark.

4 Language Implementation

Protovis instantiates visualization specifications using a multi-stage pipeline. First, Protovis processes the input data and evaluates properties to create an abstract *scenegraph* describing the visual scene. Rendering and interaction components then operate on this scenegraph, handling all integration with the underlying windowing platform. This process subdivides into six distinct phases:

- **Bind.** Traverse the mark hierarchy to resolve property definitions for each mark specification.
- **Build.** Build the scenegraph. Evaluate the *data* and *visible* properties and create a scenegraph node per datum.
- **Evaluate.** Evaluate all bound properties and store the results on the corresponding scenegraph nodes.
- **Interpolate.** During animation, advance interpolators according to the current timestep, updating affected scenegraph nodes.
- **Render.** Draw the scenegraph to a display.
- **Event.** Dispatch interaction events to registered callbacks.

With the sole exception of event handling, each of these phases is overseen by a central scheduler thread, which delegates work to additional threads as needed (see §5). When a visualization update is invoked, a new task is queued in the scheduler thread, which then commences the bind, build, and evaluate phases. The interpolate phase is triggered by a recurring scheduler task that requests animation updates in accordance with the desired frame rate. Render phases are automatically triggered upon completion of an update (bind-build-evaluate) or interpolation step. We now describe each phase in greater detail.

4.1 Bind

The bind phase traverses the mark specification hierarchy to resolve all visual property definitions. Each property is stored as an instance of a `Property` interface. These property instances come in three basic varieties: *constants* store a constant property value, *compiled* properties provide pre-defined compiled functions, and *dynamic* properties contain a string that must either be interpreted or compiled prior to property evaluation. (There is also a fourth class of properties—*event handlers*—which we will discuss later.)

For each mark, Protovis traverses the mark hierarchy to collect bound properties (see Figure 6). For a given property name, the first property definition encountered is used as the bound property. Protovis begins with the properties defined directly on the mark itself, proceeds through each mix-in registered via the **extend** method (visiting the most recently added first), then walks up the prototypal inheritance chain, and finishes with the defaults for the given mark type. The result of this traversal is an associative array consisting of all bound properties for the given mark. If it is the first instantiation of the mark, or if the collected properties differ from a previous instantiation, Protovis also constructs a new `Evaluator`: an object that performs the subsequent build and evaluate phases for the given mark.

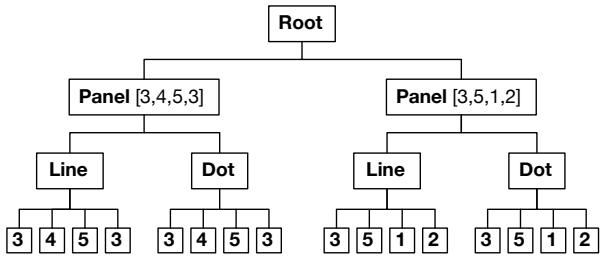

Fig. 7: Scenegraph resulting from the specification in Figure 6.

If Protovis is running on a JVM that provides compiler access as part of the standard API (e.g., Java SE 1.6), it generates source code for a new `Evaluator` and compiles it on-the-fly, resulting in a new compiled class instance. When generating the source code, *compiled* properties are included as member variables and simply invoked as necessary. Constant values are similarly included as member variables and assigned. Dynamic values are inlined within the source file and compiled at runtime. Runtime compilation imposes only a small overhead; in our tests most compilation tasks complete in 15 ms or less. We cache source code and compiled classes to prevent unnecessary compilation. As discussed in Section 5.1, we have found that dynamic compilation provides up to a 2× performance improvement.

For platforms that do not provide compiler access at runtime (e.g., Android), we instead interpret dynamic properties. Protovis uses a standard *Expression* design pattern [8, 9]: the text of each dynamic property is parsed into a tree of operators that compute the property values, and a context (or "environment") object is used to access local variables, such as the current index and other property values.

Alternatively, we could introduce a pre-processing stage prior to compilation, rewriting dynamic properties into full class definitions. However, this would limit updates to mark specifications at runtime, providing no way to evaluate new properties specified as string literals.

4.2 Build

The build phase constructs a scenegraph from the data. Each data value has a corresponding node in the scenegraph that stores all computed visual properties (e.g., **left**, **top**, **fill**, etc). The mark hierarchy is traversed in depth-first order and each mark's `Evaluator` evaluates the **data** property to retrieve the data to visualize.

Branching within the scenegraph is achieved using **Panel** marks. A panel mark with more than one data element results in multiple panel nodes in the scenegraph, useful for creating small multiples or layered visualizations. For panel marks, the **visible** property is evaluated to determine if child marks should be processed. If a panel node is not visible, neither are its children and so their evaluation is culled. If true, child marks' evaluators are invoked recursively. The end result of this process is a complete scenegraph structure, as in Figure 7. Note that siblings from the same mark instance are grouped together in the tree.

When a visualization update is performed multiple times, the build phase is responsible for updating the scenegraph structure. When the number of data elements has changed, the evaluator adds or removes nodes from the scene. If an animated transition has been requested, special care must be taken to ensure proper node correspondence. Protovis uses values provided by the **key** property to match data elements to existing scenegraph nodes, and in the process note when new data enters the scene and old data exits the scene.

To track this state, Protovis maintains a collection of per node status flags: a node is marked as *born* when it is newly created, as a *dying* node when flagged for removal (i.e., its backing data element is no longer returned by the **data** function), and as *dead* when it should be removed from the scene. For example, if a datum is removed, its corresponding node will first be marked as *dying* by the build phase. Once the animated transition (e.g., fade out) has completed, the animation interpolator will mark the node as *dead*. The build phase will then remove the node upon the next update. These flags also affect subsequent evaluation of **enter** and **exit** properties.

When the build phase completes for the scene, an update event is fired. This event enables extensions to operate with the guarantee that the scenegraph structure is in place. For example, a force-directed graph layout might advance a physical simulation; new position values can then be queried in the subsequent evaluation phase. However, this approach does introduce limitations: for example, the build and evaluate phases can not overlap, preventing pipelining.

4.3 Evaluate

The evaluate phase is the last stage of visual encoding. Protovis evaluates the mark properties and stores the results on the scenegraph nodes. Protovis traverses the scenegraph from the root down and for each sibling group, the corresponding mark `Evaluator` assigns visual property values to the nodes. The evaluator also sets a *dirty* flag on nodes whose values change as a result.

When an animation is requested, the evaluator receives an additional `Transition` instance as an argument (c.f., [11]). This object collects per-mark starting and ending values for property interpolation. If the the *born* flag is set, the **enter** properties are evaluated to determine the node's starting state. If the *dying* flag is set, the **exit** properties are evaluated to determine the node's ending state.

4.4 Interpolate

The interpolate phase advances the set of interpolators within a *Transition* to animate the scenegraph state. The interpolators are initialized during the preceding evaluate phase; interpolation is only performed for values that change between states, avoiding unneeded computation. Then, for each frame of the animation, the interpolators are stepped to calculate a new scenegraph state. Users can specify stylized animations using the **ease** mark property, which specifies a pacing function (e.g., slow-in slow out [13]), and the **delay** property, which delays animation onset to create pauses or stagger node movement [11].

4.5 Render

The render phase draws the scenegraph to a display. Protovis renderers are decoupled from the mark specification and evaluation machinery, enabling the use of platform-specific renderers. We have built renderers for AWT/Java2D, OpenGL via JOGL, and Android (with OpenGL ES to follow); Figure 8 shows an example of Protovis running on an Android phone. For each renderer we provide a corresponding class implementing a *Display* interface. Displays encapsulate a renderer and one or more visualizations into a user interface component that can then be integrated into an application. Displays also provide methods for affine transformation (e.g., pan, zoom, rotate) of the rendered view.

Renderers perform a depth-first traversal of the scenegraph and render each node according to its mark type and visual properties. By default, the rendering order is determined by the order of mark groups in the scenegraph. Users can modify this behavior using the **depth** property, which controls the render order of mark groups within a panel. Renderers are also responsible for loading and storing images used when rendering **Image** marks. Our OpenGL renderer stores each image as a texture in video memory. When rendering **Label** marks in OpenGL, we first render the text using a software renderer (Java2D) and then load the resulting bitmap as a texture.

4.6 Event

The event phase dispatches interaction events (e.g., keyboard, mouse, or touch-screen events) to callbacks registered as mark properties (see Fig. 3). Similar to other properties, event handlers can be specified as standard Java classes, or as text strings that are then interpreted or compiled. Event processing follows standard practice: an event loop pulls events from a queue, resolves the target node(s), and then dispatches the event to the appropriate callbacks. Protovis also decouples event handling from the underlying platform. Prior to dispatch, platform level events (e.g., mouse and key events from a *Display*) are intercepted and translated into Protovis' own event model, which supports key, mouse, and multi-touch events. This level of abstraction allows a single **click** event property to provide the same behavior on a desktop PC and an Android mobile phone.

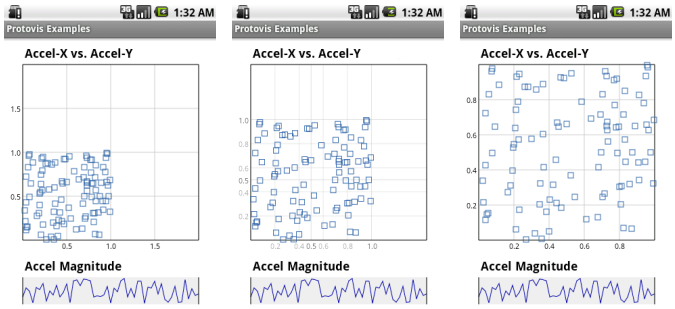

Fig. 8: Protovis on an Android mobile phone. The image sequence shows a touch-initiated animated scale transition in a scatter plot.

5 Language Optimization

The internal implementation details of the phases presented in the previous section are hidden from Protovis users, and so within each stage we are free to investigate a variety of implementation and optimization strategies without impacting designers' visualization specifications. In this section, we describe optimization techniques and parallelization strategies and present the results of benchmark studies measuring the performance impacts of each. Readers uninterested in low-level optimization details may safely skip to the section summary (§5.5).

All benchmarks were conducted on a MacPro system running Mac OS X 10.6.2. The test machine had a quad-core 2.66 GHz Intel Xeon Processor with per-core 256K L2 caches, a shared 8MB L3 cache, and a main memory of 8GB using 1066 MHz DDR3 RAM. The processor interconnect speed between cores was 4.8 GT/s. We ran each benchmark on a standard distribution Java Virtual Machine version 1.6.0 given 1GB of heap space (`-Xmx1024M` parameter). For each timing measurement, we report average values over 100 iterations; variances were a small fraction of the means and are omitted from our charts.

For our parallelization techniques we employ a task queue pattern [18]: processing is divided into a set of tasks, which are then queued. A collection of worker threads request tasks from the queue and execute them in parallel. Our implementation exposes this functionality as a general thread pool that also can be utilized by language extensions such as layout algorithms. As our benchmarks were run on a quad-core machine, at most four threads can be processed simultaneously. However, we vary the number of worker threads from 1 to 6 to assess the effect of oversubscription.

5.1 Encoding: Bind, Build, Evaluate

The bind, build, and evaluate stages populate a scenegraph and evaluate the visual properties at each node. For these encoding stages we explored three forms of optimization: parallel processing, runtime compilation of `Evaluators`, and pruning redundant updates.

5.1.1 Parallelization

Each of the bind, build, and evaluate stages are amenable to parallelization. The bind phase is "embarrasingly parallel": property resolution for each mark can be conducted in parallel without any dependencies or interactions. However, as the total number of nodes in the mark specification tree is typically small, this phase executes very quickly and so parallelization provides little benefit. As such, we currently perform the bind phase in serial.

We have instead explored parallelization strategies for the build and evaluate stages. Unfortunately, processing within both stages has a number of dependencies, precluding a naïve paralleization approach. For instance, in the build phase a panel's data property must be evaluated before any child marks can be processed. Similarly, property evaluation for a mark may depend on the value of prior nodes in the scenegraph, such as the parent panel's width and height.

As a result, we must track these dependencies and only generate parallel tasks once the dependencies have been met. For both the build and evaluate phases, we break processing of each mark (or, for evaluation, each scenegraph sibling group) into a single task. Upon com-

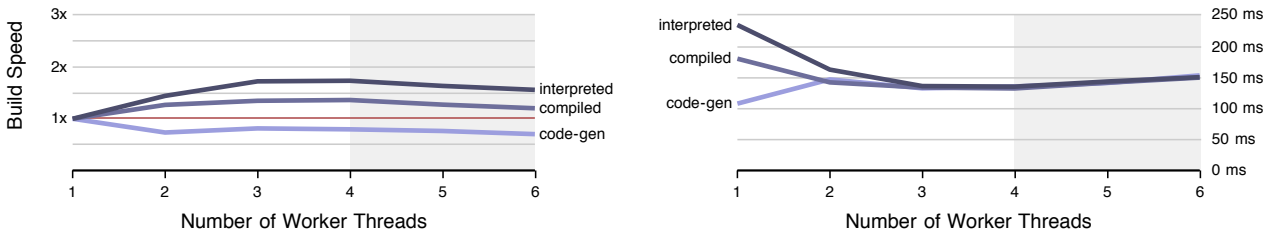

Fig. 9: Encoding (bind-build-evaluate) benchmarks for 1,000,000 elements by evaluation strategy and thread count. (a) Average performance improvement factor using single-threaded evaluation as a baseline (higher is better). (b) Average encoding time in ms (lower is better).

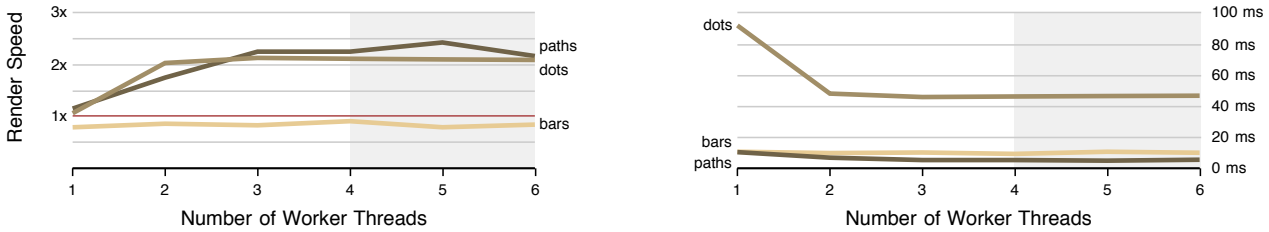

Fig. 10: Rendering benchmarks for 50,000 elements by visualization type. (a) Average performance improvement factor using a single rendering thread (no worker threads) as a baseline (higher is better). (b) Average per-frame render time in ms (lower is better).

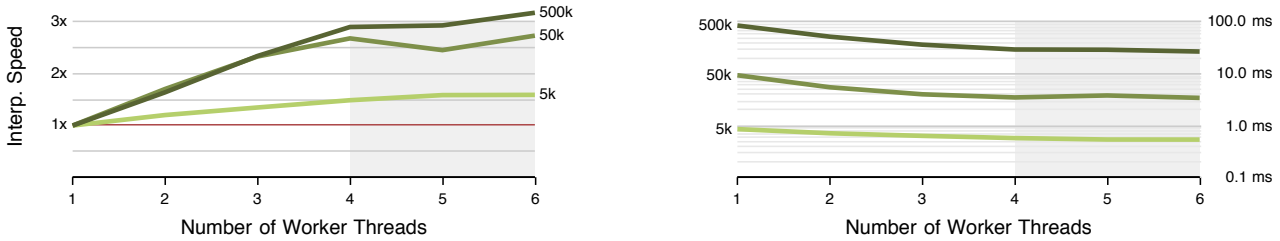

Fig. 11: Animated interpolation benchmarks by workload (# of elements). (a) Average performance improvement factor using single-threaded interpolation as a baseline (higher is better). (b) Average per-frame interpolation step time in ms, plotted on a $\log_{10}$ scale (lower is better).

pletion of a task, a new set of tasks is submitted to the task queue for the child marks. We exploit loop-parallelism by breaking up the evaluation of a sibling group into multiple parallel tasks. We evenly subdivide a group into as many tasks as there are worker threads.

5.1.2 Runtime Code-Generation and Compilation

As described in section 3.2, Protovis allows users to define property functions using string literals that are processed at runtime. These functions can either be interpreted—parsed into a tree of operators as in the Expression [8] design pattern—or instead compiled into a new class at runtime by leveraging compiler access provided by the standard Java API. The latter option introduces optimization opportunities, as we can avoid the overhead of unnecessary method calls and streamline code generation by placing constants and bound variables inline. In our benchmarks we compared three variants of property evaluation:

- *Interpreted* properties that are parsed into an operator tree and evaluated at runtime.
- *Code-gen*erated properties that are dynamically compiled at runtime to create a new Evaluator instance.
- *Compiled* properties written as regular Java classes and subject to the standard compilation process.

5.1.3 Encoding Benchmarks

Our benchmark consisted of computing visual encodings for a scatter plot of 1 million elements. Spatial position was randomly assigned within the width and height of the parent panel; each point was encoded as a stroked, blue circle with a radius varied according to the `index` value. Evaluation of non-constant properties was varied using the property evaluation strategies described above. Although a single plot with a million elements is rare, it provides a useful measure of scalability; e.g., consider the more plausible case of a 10×10 scatter plot matrix where each plot contains 10,000 points.

The performance data is shown in Figure 9. Figure 9(a) shows the relative change in performance using the single-threaded case as a baseline. For both the compiled and interpreted conditions, we see that execution time improves as the number of threads is increased, with best performance when the thread count is either 3 or 4. For higher threads, performance worsened. In all cases, the performance improvement is less than a factor of two. The results for runtime compilation show a different picture: single-threaded execution outperforms parallel processing. Profiling reveals that in the single-threaded condition less time is spent in the property evaluation stage. Our hypothesis is that by streamlining code-generation we are able to generate methods that the JVM just-in-time compiler can more effectively optimize. Parallelization worsens performance in this case, likely due to increased overhead mitigating any runtime optimizations.

Figure 9(b) shows the average time in milliseconds of the bind-build-evaluate phase. We see that as the number of threads increases, the performance difference between different evaluation strategies disappears. We also see clear benefits for single-threaded evaluation of runtime-compiled properties: updates are up to 2.2 times faster than

the other approaches when executed serially, and 1.2 times faster than the peak parallel performance shared by all three approaches. These results argue for single-threaded execution of optimized evaluators, but also show that nearly equivalent performance can be achieved by parallel processing of less-efficient evaluation strategies—a useful result when runtime compilation is not an option.

We also compared performance in an equivalent visual encoding task using the prefuse visualization toolkit [10]. We measured the time required to compute encodings for an identical scatter plot using a standard prefuse ActionList. The average visual encoding time for prefuse was 784 ms—over 7 times slower than Protovis' peak performance and over 3 times slower than Protovis' poorest performance. Profiling reveals that the gap is primarily due to row and column lookup overhead in prefuse's toolkit specific data structures, reinforcing one benefit of Protovis' data agnostic approach. However, we note that in practice prefuse's software renderer (Java2D) is the more significant performance bottleneck.

5.1.4 Pruning Redundant Updates

Next, we assessed the benefits of pruning unnecessary computation during Protovis' evaluation phase; many property functions will return the same value on reinvocation if the data is unchanged. To assess the optimization potential, we used the same million-element scatterplot example and compared three conditions: (a) re-evaluating all properties, (b) avoiding re-assignment of constant values based on a property dirty flag, and (c) looping through all scenegraph nodes but assigning nothing. The final condition was included in order to estimate a lower bound on processing time. Figure 12 shows the results: simply culling redundant constant assignments reduces running time by roughly 25%, while skipping all property functions establishes a rough lower bound of a 3× performance gain. These results suggest that more aggressive pruning—by explicitly modeling the dependencies among property functions to perform lazy evaluation—could yield further benefits. However, rendering time and expensive layout computations (e.g., force-directed layout) typically dominate such gains.

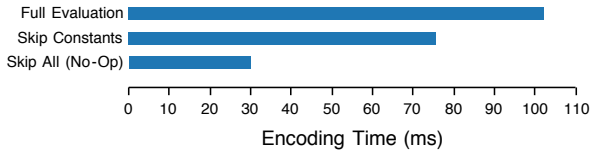

Fig. 12: Encoding time for 1,000,000 elements, by pruning strategy. Evaluators created via code-generation are used in each example.

5.2 Render

A primary performance bottleneck in many information visualization systems is rendering, particularly among the tools that use software renderers [7, 10, 23]. As described in Section 4.5, Protovis' declarative design enables seamless switching between different rendering targets, including hardware-accelerated graphics via OpenGL. Unsurprisingly, this switch alone can improve performance by an order of magnitude. We apply common OpenGL optimizations, such as using vertex buffers, to maximize rendering performance. Protovis' grouping of sibling marks in the scenegraph facilitates this process by batching similar geometries together. For opaque marks, we use the depth buffer to cull unnecessary rendering. Finally, our renderers support a "cache-as-bitmap" option for panels (exposed to users via the **cache** property): non-volatile subtrees in the scenegraph are rendered once and cached as a texture in video memory.

We also sought to parallelize rendering with a single graphics card. As OpenGL is single-threaded, we implemented a master/worker pattern [18]: worker threads compute the geometry (e.g., vertex and color buffers) and a single master thread sends it to the graphics card. The master thread generates tasks via pre-order traversal of the scenegraph and ensures that geometry and color data are sent to the graphics card in that order. Parallelism should improve performance in situations where non-trivial processing is needed to prepare the geometry.

We benchmarked Protovis rendering performance using both a standard serial renderer and a master/worker parallel renderer. We tested rendering of three different visualizations of 50,000 elements, each requiring a greater processing workload to prepare the geometry:

- *bars*: A stacked bar chart consisting of simple rectangles.
- *paths*: A small-multiples line chart with varying line widths, connected via miter joins.
- *dots*: A scatter plot of stroked circles, requiring discretization of the circumference into line segments.

The results in Figure 10 show mixed benefits for parallelized rendering. The *bars* condition, which consists of rectangles that require virtually no processing prior to rendering, suffers from the overhead induced by parallel task management. The other two conditions, however, gain a 2× performance increase by delegating miter join calculation and shape discretization to worker threads. These benefits manifest with 2-3 threads; more threads do not provide additional benefits. We also compared these results with Protovis' Java2D software renderer: hardware acceleration improved rendering by 4.4×, 10.5×, and 14.6× for bars, paths, and dots respectively.

5.3 Interpolate

Parallelizing interpolation during animation is easy: once starting and ending values have been computed, each scenegraph node can be considered in isolation. As a result, we can run multiple simultaneous transitions in parallel, and can exploit loop-parallelism by breaking a transition consisting of multiple scenegraph nodes into parallel subtasks. Figure 11 shows the result of parallelizing interpolation for three different workloads consisting of 5,000; 50,000; and 500,000 interpolated elements in a scatter plot display. In each case, only spatial position properties (left, top, right, bottom) were interpolated. In each case, increased parallelism improves the animation processing time, with negligible benefits for oversubscription. When drawing animated elements as single pixel points, parallelism improves the frame rate in the 500k element condition from 9.1 fps to 21.3 fps.

5.4 Application Performance

In addition to benchmarks of specific pipeline stages, we tested overall application performance in a graph exploration tool. The application involves visual encoding, layout calculation, and rendering and so stresses multiple parts of the infrastructure. We measured frame rates during interactive graph layout, computing one iteration of a force-directed layout per frame. We used an artificial graph consisting of N nodes in a linear chain, with N varied from 100 to 100k by factors of 10. We connected each node to the next 10 nodes in sequence; thus the graph with N=100k nodes had nearly 1 million edges. We compared Protovis using serial code-generated evaluators and parallelized rendering (4 worker threads) to a nearly identical application created in prefuse. As shown in Table 1, Protovis consistently has frame rates an order of magnitude higher, up to 20 times faster for large graphs.

Table 1: Average frame rate of graph layout in frames/sec.

Nodes	Edges	Protovis	prefuse	Difference
100	955	66.00	44.45	1.5×
1,000	9,955	66.00	5.94	11.2×
10,000	99,955	11.76	0.56	20.1×
100,000	999,955	0.93	0.05	18.6×

5.5 Summary

In the bind-build-evaluate stages, we found that dynamic compilation of serially-executed `Evaluators` leads to the best performance (up to a 2× speed-up). The advantage for single-threaded evaluation most likely stems from JVM runtime optimizations enabled by our code-generation scheme. For other property evaluation approaches, we find that parallel execution provides a mild benefit (up to 1.6×). Modeling the dependencies among properties may improve this further. In the rendering stage, we found that parallelized rendering using worker threads to prepare geometry and color data provides over 2× improvement for processing-heavy marks (e.g., miter joins and circular

discretization), and that performance plateaus at 2-3 worker threads. However, parallelization overhead degrades performance when rendering simple rectangles. We also found that animation interpolation benefits from parallelization and scales well with the number of available cores. Finally, combining these optimizations we find that Protovis provides up to 20× higher frame rates than prefuse in a graph exploration application. These performance improvements help visual analysis scale to larger data sets.

However, more important than any specific optimization, the use of a declarative specification language has enabled us to explore a space of optimizations without impacting how users create visualizations. We believe that a declarative approach provides an ideal framework in which to explore additional sophisticated optimizations. For example, Piringer et al.'s [19] threading strategies are applicable within Protovis, as our task-based parallelization can support interrupt-driven early termination with minimal modification. More sophisticated uses of GPU hardware might also be explored, as in McDonnel and Elmqvist's [16] image-space operations model, while insulating visualization designers from the vagaries of shading languages. A shared language model for interactive visualization can serve as a platform for continued research and facilitate the dissemination of improved techniques.

6 Discussion

We presented the design, implementation, and performance evaluation of Protovis, a declarative, embedded domain-specific language for specifying interactive visualizations. Building on prior work [1], we extended the design of the Protovis language to include declarative specification of animated transitions. We explained how the language can be implemented in a statically-typed object-oriented language and enable rich visualizations without requiring a toolkit-specific data model. We then demonstrated how our declarative approach supports retargeting across multiple rendering and interaction platforms.

Next, we introduced optimizations such as runtime compilation of visualization specifications and parallel processing of visual encoding, interpolation, and rendering stages. Within each category, benchmark studies show that our optimizations can improve Protovis performance by factors of 2 to 3. In the case of animated interpolation, these improvements continue to scale with the number of processor cores. We also compared Protovis performance with prefuse, a popular Java visualization tool; a network visualization built with Protovis has up to 20 times higher frame rates than an equivalent prefuse-based visualization. While these improvements increase the scalability of visual analysis, they also serve to demonstrate the benefits of a declarative approach: because the control flow is not the user's responsibility, the system can make optimization decisions on the user's behalf.

A potential downside of declarative languages is that because the control flow is not visible, the system might break in unexpected ways that are difficult to debug. These crashes expose the underlying control flow, as implemented by the system (rather than the user), potentially adding confusion. For this reason, declarative systems should have well-defined behaviors for invalid input and can benefit from improved debugging support. New visual interfaces might support direct manipulation of declarative visualization specifications, and enable brushing and linking between specification code and running visualizations. A specification editor might serve as a live interpreter, enabling designers to sculpt their visualizations iteratively while working with real data.

While our use of the Java programming language has supported our prototyping process, it is less well-suited to production use. Though expedient, our use of string constants to specify anonymous functions is clearly sub-optimal: it violates language norms and stymies IDE support and debugging. In the future, we expect the insights gained in the current research will be best applied in other programming languages. For example, as JavaScript performance continues to improve, and additions such as worker threads and WebGL rendering become commonplace, some of the approaches explored here can be transferred to the JavaScript implementation of Protovis.

Strongly-typed object-oriented languages that include functional programming capabilities also provide attractive targets. For example, C# now supports anonymous (lambda) functions and enables declarative data manipulation via the LINQ language extension. The Scala language combines functional and object-oriented paradigms, can target both the JVM and the .NET CLR, and is being extended to support staged compilation via "lifting" of program components [2]. This enables developers to insert customized routines for manipulating a program's abstract syntax tree and performing code generation. As a result, dependency detection and optimized code generation may be possible while conforming to the norms of the programming language. These developments bode well for the evolution of accessible, expressive, and scalable languages for visualization.

Protovis source code, including all performance benchmarks, is available online at `http://protovis.org/protovis-java`.

References

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

[2] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language virtualization for heterogeneous parallel computing. In *Onward!*, 2010.

[3] J. A. Cottam and A. Lumsdaine. ThisStar: Declarative visualization prototype. In *IEEE Information Visualization Posters*, 2007.

[4] J. A. Cottam and A. Lumsdaine. Stencil: A conceptual model for representation and interaction. In *International Information Visualisation Conference*, pages 51–56, 2008.

[5] D. J. Duke, R. Borgo, M. Wallace, and C. Runciman. Huge data but small programs: Visualization design via multiple embedded DSLs. In *Practical Aspects of Declarative Languages*, pages 31–45. Springer, 2009.

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

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

[8] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Upper Saddle River, NJ, 1994.

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

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

[11] J. Heer and G. G. Robertson. Animated transitions in statistical data graphics. *IEEE Trans. Vis. and Comp. Graphics*, 13(6):1240–1247, 2007.

[12] P. Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4), 1996.

[13] 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.

[14] H. W. Lie. *Cascading Style Sheets*. PhD thesis, University of Oslo, 2005.

[15] B. Lucas, G. D. Abram, N. S. Collins, D. A. Epstein, D. L. Gresh, and K. P. McAuliffe. An architecture for a scientific visualization system. In *IEEE Visualization*, pages 107–114. IEEE Computer Society Press, 1992.

[16] B. McDonnel and N. Elmqvist. Towards utilizing GPUs in information visualization: A model and implementation of image-space operations. *IEEE Trans. Vis. and Comp. Graphics*, 15(6):1105–1112, 2009.

[17] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[18] A Pattern Language for Parallel Programming. `http://parlab.eecs.berkeley.edu/wiki/patterns`, March 2009.

[19] H. Piringer, C. Tominski, P. Muigg, and W. Berger. A multi-threading architecture to support interactive visual exploration. *IEEE Trans. Vis. and Comp. Graphics*, 15(6):1113–1120, 2009.

[20] A. Slingsby, J. Dykes, and J. Wood. Configuring hierarchical layouts to address research questions. *IEEE Trans. Vis. and Comp. Graphics*, 15(6):977–984, 2009.

[21] 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.

[22] 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.

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

[24] H. Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer, 2009.

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