# ProtoWiz – Moderately Complex Visualizations for the Ambitious Non-Programmer

Michael Cohen and Thomas Schluchter

**Abstract**—We present ProtoWiz, a browser-based frontend to the powerful JavaScript visualization library Protovis. ProtoWiz allows users with a non-technical background to explore the possibilites of Protovis in a visual environment rather than in code.

**Index Terms**—ProtoWiz, Direct manipulation, User testing, code generation.

## Introduction

Domain-specific visualization toolkits such as Protovis enable users with pre-existing programming experience (or a willingness to learn) to create data-driven visualizations concisely and quickly. Protovis seeks to strike a unique balance between '*expressiveness* ("Can I build it?"), *efficiency* ("How long will it take?") and *accessibility* ("Do I know how?")'[2] In designing Protovis, its authors were seeking a happy medium on these dimensions, between highly accessible and efficient "closed" visualization systems such as Microsoft Excel and Tableau/Polaris, and ultimately expressive low-level graphics libraries such as Processing (or low-level design tools such as Adobe Illustrator). Their explicit target audience was web developers, who likely have some pre-existing familiarity with cascading style sheets (CSS) and possibly with JavaScript, but who would find a full-fledged graphics toolkit like Processing daunting.

We argue that Protovis succeeds admirably in retaining much of the accessibility and efficiency of high level tools while allowing virtually unlimited expressiveness, at least within the domain of visualizations in two spatial dimensions. However, we also argue that there remains a great deal of unexplored space in the continuum of trade-offs between expressiveness, efficiency and accessibility. In particular, Protovis leans heavily towards expressiveness, and by targeting an audience with prior coding experience it naturally limits the other two dimensions. Accessibility is limited because users not comfortable with writing and testing code will face a steep combined learning curve as they learn both Protovis and JavaScript. Efficiency suffers in a more subtle way; although Bostock and Heer demonstrate that Protovis is remarkably efficient in terms of lines of code required to produce high-value results, the time required to find the *correct* few lines of code may be quite long for the novice programmer. Protovis likely has significant advantages over learning to perform the equivalent operations in Processing or Open GL, but efficiency barriers remain that may discourage potential users who technically have the skills to meet basic accessibility requirements.

Fortunately, Protovis is more than a JavaScript programming toolkit. It is also an elegant conceptual framework that breaks down two-dimensional visualization into its fundamental building blocks of panels, data, scales, marks and their properties. Although Protovis makes clever use of JavaScript inheritance relations and functional programming to express these concepts concisely, the concepts themselves are not JavaScript-dependent, or dependent on computer programming in general. Rather they are closely related to visual ideas that are quite accessible to non-programmers, and particularly designers. Therefore, Protovis provides us with an excellent foundation to explore the "middle" of the expressiveness-accessibility-efficiency continuum. That is, it inspires the question: can we create a framework that captures most (if not all) of the expressiveness of Protovis, while substantially improving its accessibility and efficiency? As an initial response to this challenge, we present ProtoWiz, a browser-based interface for constructing novel visualizations in Protovis.

ProtoWiz allows users to assemble data, scales and mark properties using familiar HTML form controls and drag-and-drop interactivity. The Protovis code corresponding to the form-based specification is regenerated on the fly after each property change, and is then parsed and rendered immediately so that the user receives continuous feedback on the results of her changes. Once the visualization has reached a satisfactory state (or a state where further development requires directly editing the code) the user can export the JavaScript code to cut-and-paste into her own web page.

We believe that Protovis fills a substantial "gap" in the field of visualization tools available. It is far more expressive than closed tools such as Excel, while having a substantially gentler learning curve than Protovis itself. Practically speaking, we believe that ProtoWiz can play a role in the visualization world analogous to the role played by WYSIWYG tools such as Adobe Dreamweaver in web development. In particular, ProtoWiz serves three valuable functions for three classes of users:

- It allows non-programmers to explore a substantial fraction of the Protovis design space with lower barriers to entry. When the complexity of the desired visualization is simple-to-moderate, ProtoWiz allows quite rapid completion of the entire project, with virtually no typing of code.
- It serves as a helpful introduction and teaching tool for users who are interested in eventually graduating to the greater expressiveness of direct Protovis coding. It does this by producing concise, idiomatic Protovis code that closely matches the style of the Protovis example gallery, and also by allowing users to rapidly explore ideas with a "safety net" of immediate feedback about the validity and aesthetics of the encoding strategy being attempted.
- Although ProtoWiz has little to offer by way of accessibility to experienced Protovis coders, they can still benefit from its efficiency advantages. Experienced Protovis developers will find it useful to quickly "scaffold" the basic structure of more complex projects.


- *Michael Cohen is with the Energy and Resources Group at the University of California Berkeley, E-Mail: macohen@berkeley.edu.*
- *Thomas Schluchter is with the School of Information at the University of California Berkeley, E-Mail: thomas@ischool.berkeley.edu.*

# 1 Related Work

The existing tool most similar to ProtoWiz is Tableau (and its predecessor Polaris [11]). Like ProtoWiz, Tableau offers a highly interactive way of transforming data into visualizations using a combination of form input and drag-and-drop interactivity coupled with immediate feedback. As Bostock and Heer [2] point out, however, Tableau is ultimately a closed system and offers limited expressiveness. Furthermore, even within its ostensible range of expressiveness (i.e. available chart types) Tableau can be constrained by its close coupling to relational database concepts. In fact, part of the inspiration for ProtoWiz stemmed from the frustration that one the authors of this paper experienced when attempting to use Tableau to plot two data sets that did not have a relational connection on a single set of axes. Although Tableau allowed rapid and intuitive exploration of the two individual plots, there was no way within Tableau to superimpose them. Clearly, this example called for a more graphical approach, but at the time this required resorting to image editing tools, or sacrificing all the efficiency and accessibility of Tableau to re-code the individual plots in Protovis or a similar toolkit. This was a strong indication that there was an unexplored gap in the range of available visualization toolkits.

We drew general inspiration for the idea that direct manipulation of parameter manipulation to display can enable non-programmer users to engage fluidly with data from the work of Ahlberg and Shneiderman on "tight coupling." [1]

ProtoWiz uses a "templating" approach to code generation, guiding users to define appropriate properties for each mark and assign those properties appropriate values by presenting the most likely options in drop-down lists. Perhaps the most well-known existing templated programming language is Scratch [7], which is designed to introduce school-age students to programming (often without them being explicitly aware of the introduction). Scratch appeals to students because of its focus on storytelling and animation, and because its templated, color-coded nature makes it easier to assemble programs without running into confusing syntax errors. ProtoWiz similarly takes advantage of the templating approach to ease the programming learning curve. Like Scratch, ProtoWiz generates some code implicitly "behind the scenes", allowing the user to focus on manipulating the important, story-centric variables. Unlike Scratch, the complete code generated by ProtoWiz is only a button click away, making it more appropriate as a learning tool for adults who will likely be interested in understanding what is going on behind the scenes, and perhaps moving on to writing (or editing) their own Protovis code in short order.

# 2 ProtoWiz Architecture and Design

ProtoWiz' code generation capabilities rely on an architecture that generalizes key concepts of Protovis and makes them available as an internal programming interface. We describe the components of this programming interface in turn.

## 2.1 Properties

ProtoWiz draws heavily from Protovis' marks-with-properties architecture, and in fact extends this architecture to encompass certain aspects of the code that are not "properties" per se in Protovis, to provide a clearer, more consistent interface for users.

The basic unit of manipulation in ProtoWiz is a *property*. A property is an object (specifically, a function object) with the following key capabilities:

- *Form generation*. Each property generates a snippet of Document Object Model (DOM) code, which we refer to as its *form*, that will display its current value and allow users to manipulate that value. The form is tailored to present the most common and valid choices for that property. For instance, many properties are commonly defined using a Protovis scale applied to a data column; thus, these properties in ProtoWiz present the user with drop-downs that are pre-populated with the list of available data fields and defined scales. When the value of one of the property's form fields is changed, this triggers an update to the property's internal value, using its *accessor*.
- *Accessor*. The basic property object is actually a JavaScript function that serves as an accessor for the property's internal value. Calling this function with no parameters returns the current value of the property. Calling it with an argument sets the value of the property. The setter performs five tasks: 1) basic validation of the new value, 2) updating the property's internal value to reflect the new value, if it is valid, 3) updating the property's form to reflect its new value, 4) if necessary, updating other properties' forms and values when they are dependent on the current property (see *form updating* below for an example), 5) re-generating the protovis code for the visualization and re-rendering it, providing immediate feedback to the user on the consequences of the change.
- *Code generation*. Each property can generate a code snippet based on its value. This snippet will generally take the form "*.propertyName*(*expanded propertyValue*)". This format lends itself to method chaining with no additional "glue" required; code from each of a mark's properties is simply concatenated. There are notable exceptions to this pattern, which are addressed below.
- *Form updating*. Some (but not all) properties will need to update the options presented to the user when other properties are changed. The most obvious example is the "scale and field" type property outlined above. The options presented by the form will need to change when a scale is added or removed, or when the data fields available change (e.g. when a mark's data property changes). One subtlety here is that the change in form options may imply a required change in the underlying property value. For instance, if a property was based on field "x" in the data set, and the mark is now using a new data set with no field called "x" then the property will tend to be left in a state that will result in a parsing error when the visualization is rendered. ProtoWiz attempts to account for this by blanking out form fields that would otherwise be left with an inappropriate value. Although we believe this is the "lesser of two evils" compared to leaving the field in an error-generating state, it could still cause confusion if the user does not understand why the field was blanked out.

ProtoWiz properties are arranged into a hierarchy to maximize code re-use and efficiency where functionality is shared. All properties ultimately descend from the basic abstract *property*. So, for instance, *scaleAndFieldProperty* is an abstraction that descends from *property* and provides the common capabilities of properties that are frequently defined by applying a scale to a data field. *posProperty* and *styleProperty* are further abstractions based on *scaleAndFieldProperty*. *posProperty* adds a "side" selector so the user can choose which direction the position is calculated from, whereas *styleProperty* adds an alpha (opacity) selector. Finally, concrete properties such as *xProperty* and *yProperty* descend from *posProperty*, and *strokeStyleProperty* and *fillStyleProperty* descend from *styleProperty*.

Note that the concrete properties in ProtoWiz do not always correspond one-to-one with the properties of Protovis marks. For instance, ProtoWiz defines *xProperty*, which allows users to define the "left" or "right" Protovis property of the mark, but not both. This is one instance where a small amount of expressiveness was

intentionally traded off for a significant accessibility improvement. Good reasons to define both the left and right property of a mark are rare (and in any case the same result can almost always be obtained by defining width instead) and the potential for confusion when a new user naively sets the left, right and width properties simultaneously is high. By making it explicit in the interface that the left and right properties are mutually exclusive, and by allowing the user to rapidly switch between them and immediately see the difference, we are able to short-circuit what may be a painful learning experience for the Protovis novice.

## 2.2 Marks, Scales and Data Sets

A mark in ProtoWiz is essentially a collection of properties. Since values are stored on the individual properties, marks do not need accessors of their own. However, marks perform important coordination functions that parallel the other three capabilities of properties at a higher level of aggregation.

- *Form generation*. The mark has its own *form*, which is essentially a lightweight DOM wrapper to which the properties can attach their own forms. At creation time, a mark calls upon all of its properties to generate and append their forms to the mark's form. These initial forms are always present throughout the life of the mark, though they may be rendered invisible depending on the mark's type (see custom properties, below).
- *Code generation*. For the most part, a mark can simply loop through its properties and concatenate their code into an overall Protovis description of the mark. However, there are certain ProtoWiz custom properties that must be handled specially, such as name, parent and type. More on this under "custom properties" below.
- *Form updating*. Each mark provides a convenient method to update all of its field-and-scale based properties since this is a common operation. The details of updating each property are left to the property itself.

A key conceptual abstraction in ProtoWiz is that ProtoWiz scales and data sets are implemented as objects based upon ProtoWiz marks. (That is, scales and data sets are "descended from" marks, in the prototypal sense.) Although they are implemented quite differently in Protovis, from a user interface perspective, ProtoWiz scales and data sets perform exactly the same function as marks – that is, they aggregate a collection of properties and generate code that correctly specifies the aggregate unit. Because scales, especially, do not rely as heavily on the repetitive method-chaining syntax of marks, the aggregate code-generation procedure for scales is slightly more complex than that of marks. In spite of this minor additional complexity, there are clear architectural and code-reuse advantages to instantiating marks, scales and data sets using a consistent paradigm.

For example, a quantitative scale's domain is often defined in terms of the maximum and minimum values in a particular data field. In the ProtoWiz architecture, we simply re-use the *dataProperty* that we created for marks and add it to scales as well. Users can set this property to choose a data set to base the scale's domain on, which changes the drop-down options available in the *domainProperty*'s form. Since Protovis scales do not really have a data property, the scale's *dataProperty* is simply ignored in the scale's code-generation method. By applying the ProtoWiz property architecture, we are able to efficiently re-use code on the development side while also providing the user with the most consistent experience possible, hiding away some implementation details that are not relevant to the conceptual design of visualizations.

## 2.3 Custom properties

In addition to making available nearly all of the properties of Protovis marks (exceptions include properties dealing with interactivity, such as the "events" property) ProtoWiz defines a few additional properties that support a level of efficient interaction that would be impossible in a directly-coded Protovis visualization. Most notable are the *nameProperty*, the *typeProperty* and the *parentProperty*.

*nameProperty* is where users enter a (required) name for each mark. This name identifies the mark in the layers area of the user interface, and also serves as the JavaScript identifier for the mark in the generated Protovis code. The *nameProperty* form will immediately correct an attempt to use an inappropriate identifier, for instance by replacing space characters with underscores and adding a unique number on to the end of a duplicated name; this prevents the user from having to puzzle through an entire class of trivial errors related to naming, and provides direct guidance on what constitutes an appropriate name in JavaScript without sacrificing flexibility.

In a rare intentional departure from Protovis idiom, ProtoWiz assigns every mark an identifier, even one that could be defined simply by chaining an ".add" directly onto the end of its parent mark. There are two reasons for this. First, it makes code generation considerably simpler and more general – e.g., the case where two marks need to inherit from the same parent mark can be handled without any branching or special cases. Second, it makes it easier for a novice user to examine the complete generated code and see which blocks correspond to which marks, facilitating the matching of final code to the mental/user interface model. We deemed that these advantages were worth the minor deviation from convention (and, in any case, in any Protovis visualization at least some marks will have names, so the pattern should be familiar enough).

In Protovis, each mark type is represented by a separate class that defines the effect of property methods relevant to itself. Because different mark types share many properties with the same meaning, it is easy for marks to inherit useful information from a parent mark of a different type, and it is relatively simple to change the type of a mark to explore a different representation. However, there are a couple of challenges inherent in the code-based approach to mark creation. First, because many properties are shared but many are not, it takes practice to learn which properties will create the desired effects with each mark type. For instance, *width* is used with bars, whereas *lineWidth* is used with dots and lines. Second, maintaining clean code when switching between property types can require significant trial and error at first. For instance, suppose that a user is creating a step chart in various colors using a line mark with "step-before" *interpolation* and *segmented* set to true, but then decides that a bar mark may better form the desired representation. It is easy enough to change a pv.Bar to a pv.Line, but then the user is faced with the challenge of property housekeeping. For instance, does the new bar also need to be *segmented*? Is that required (like it was for the multi-colored line), optional, or meaningless for a bar? For tidiness and to avoid unexpected behavior it is often desirable to delete properties that are no longer needed. The user can comment them out while exploring, but has to remember to uncomment the correct ones if she reverses her choice of mark type.

To an experienced programmer, learning which properties go with which marks and keeping track of which are relevant as different mark types are explored may seem trivial. However, to a novice or non-programmer they can appear to be formidable barriers that lead to time drained consulting references and/or frustrating lack of response when one believes one has made a valid change. Furthermore, neither activity adds value to the visualization – they are simple memorization and housekeeping tasks. ProtoWiz's contribution to minimizing these challenges is the *typeProperty*. In ProtoWiz a mark's type at creation time is

simply a starting point, and can be changed at any time by choosing a different type from the *typeProperty*'s form's select box. Aside from changing the type declared in the generated code, the *typeProperty* hides all properties that are not relevant to the new mark type. Furthermore, hidden properties always generate an empty string when their code method is called. In other words, both the user interface and the generated code immediately direct the users attention to the relevant properties for the chosen mark type. However, the former value of the hidden properties is still stored in the underlying property object, so if the user changes her mind about a given type change, reversing it will transparently restore her former values, leading to a highly accessible and efficient exploration process.

The *parentProperty* allows the user to select a mark's immediate ancestor on the inheritance chain. Users can do this by selecting the name of any mark on a lower later from a generated drop-down list. Parent names update automatically when the referenced mark's name changes, allowing users to be more flexible about names than they would be in a coding environment where name changes would require explicit follow-up using find-and-replace to ensure consistency.

### 2.4 Graphical decomposition

ProtoWiz follows closely Protovis' paradigm of graphical decomposition. Like Protovis, ProtoWiz requires users to mentally decompose the desired output into a combination of dots, bars, wedges, labels, and so on. Additionally, ProtoWiz makes explicit the idea of *layers*, which exist only implicitly in Protovis. Because Scalable Vector Graphics (SVG, the final output format of Protovis) does not support explicit z-ordering, marks are simply rendered in the order that they are coded, with the newest mark on top of the previous mark. It is possible to control z-ordering by rearranging the order of mark definitions, but this is unlikely to be intuitive to novices. ProtoWiz makes the idea of ordering in depth explicit by wrapping each mark's form in a draggable container, and supporting dynamic re-ordering of the containers, which is immediately reflected in the draw order on the canvas. Rearranging layers has the potential to break inheritance relationships between marks (since inheritance, like z-order, must go in code order) so ProtoWiz provides a warning if a user is about to make such a change. The layers metaphor not only makes the underlying operation of Protovis and SVG clearer, it also leverages a user's potential familiarity with other layer-based tools such as Adobe Photoshop.

### 2.5 Form field flexibility and code expansion

While the ProtoWiz drop-down menus alone are adequate for the creation of many simple visualizations, expressiveness demands that users be able to enter simple snippets of code occasionally. Our design goal is to make this coding feel similar in complexity to the code written for an Excel formula (which many users will be familiar with) or the expression builders in, for example, Tableau or desktop databases such as FileMaker and Microsoft Access.

To enable the input of custom code, we made use of the simpleCombo [10] plug-in for jQuery [5] (which itself was used extensively to build the interface). simpleCombo allows standard drop-down menus to be outfitted with a first option element that can be edited by entering keystrokes. In the interface we have colored these drop-downs cyan to indicate their unusual nature. If a user decides to select a pre-populated option instead, the custom option remains in the list unless it is typed over later on.

To simplify the task of writing code to define properties, ProtoWiz requires only that users explicitly write the code for data fields, operators, and/or literal values needed to generate the appropriate property values. Users do not need to understand that sometimes they are writing the body of a function while other times they are not. ProtoWiz scans the entered values at code-generation time and prepends them with *function(d)* (if the user references *d.foo*) or *function()* (if the user references *this*). This generates working code based on a simplified specification, but allows the user to defer understanding of the inner workings of anonymous functions which can be quite mysterious to novices.

## 3 Evaluation

For the purpose of evaluating the implementation approach, we conducted two separate evaluation steps:

1. We benchmarked ProtoWiz against selected visualizations from the Protovis examples gallery [9] to see whether it was possible to recreate them using ProtoWiz alone, and compared the code that ProtoWiz generates to Protovis example code.
2. We conducted user tests with a prototype of the system. The goal here was to evaluate whether the user interface supported non-programmers in the creation of moderately complex visualizations.

### 3.1 Benchmark visualizations and generated code

The following visualizations could be reproduced using the ProtoWiz interface:

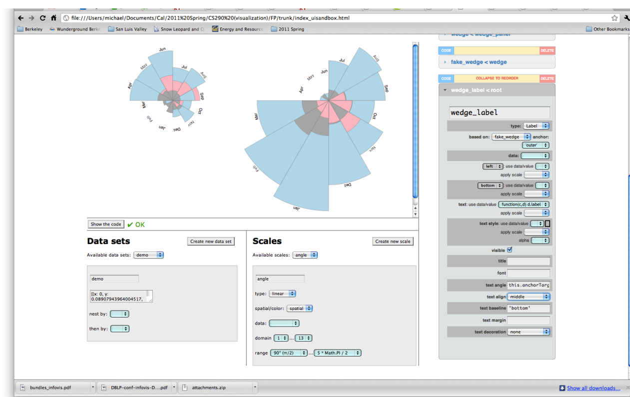

Figure 1: Nightingale's Rose reproduced in ProtoWiz

Although Nightingale's Rose (Figure 1) is reproducible within ProtoWiz, there are some significant challenges to doing so. First, because ProtoWiz does not yet support most of Protovis' data transformations, it is necessary to add columns to the data set that specify the "max" radius of each slice (for label positioning), the number of the month, and the abbreviation of the month. We do not consider this a major shortcoming as most users interested in creating data-centric visualizations will have some experience with spreadsheets and would likely prefer doing the data manipulation in a spreadsheet to learning the equivalent commands in ProtoWiz in any case. Doing data manipulation in the spreadsheet may require toggling back and forth between the two programs as a user discovers a need for an additional column, but since a data set can be re-pasted without disrupting the property definitions, we believe this is a minor annoyance.

More problematic are property values that are valid in Protovis but cannot be used in ProtoWiz. In the original Rose example, the strokeStyle is defined as this.fillStyle().darker(). Unfortunately, this construction generates a rendering error in ProtoWiz because in the generated code the strokeStyle property is declared before the fillStyle property, and therefore the fillStyle is not yet available to use when strokeStyle is declared. (It would be possible to work around this by reversing the dependency. That is, darkening the original color scale, applying it to the strokeStyle, and then defining the fillStyle as this.strokeStyle().lighter() instead. However, this is an arbitrary and unnecessary asymmetry to impose on users.) Future versions of ProtoWiz may be able to detect dependent properties and reorder them as necessary. For the current example, the strokeStyle was simply defined as a thin gray

line instead, with no relation to the fillStyle. In this case, the visual difference is barely noticeable, but it does point to one area where working purely in ProtoWiz places a hard limit on expressiveness.

### Figure 1: Protovis example code

Finally, there are some property definitions that can be created in ProtoWiz, but only by writing substantial code snippets that imply a prior familiarity with Protovis. For instance, in order to generate the Rose, a user must currently enter the following code into the interface verbatim:

- function(d) causes.sort(function(a, b) d[b] - d[a])
- function(c, d) radius(d[c])
- this.anchorTarget().midAngle() + Math.PI / 2

On the one hand, the fact that the Rose can be created with only a few "real" lines of code can be considered a significant step forward. On the other hand, it argues for further refinement of the interface (and especially for the addition of an expression builder) so that non-programmers have a reasonable chance of producing visualizations at this level of sophistication.

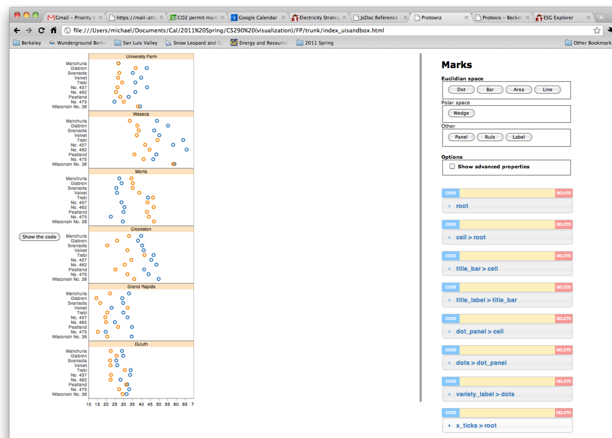

Figure 2: Becker's barley visualization is a good example of the use of small multiples

Support for small multiples (cf. Figure ) was added by allowing multiple panels in the same visualization. Currently, the data roll-up functionality built into Protovis is not available through the ProtoWiz interface, so that the order of panels doesn't quite match the original example. This feature will be included in a future version.

ProtoWiz generates concise, idiomatic code that will be familiar to browsers of the Protovis example gallery. To demonstrate this, we present the code to generate scales and a single dot mark for a simple scatter plot, both from the example gallery and generated by ProtoWiz. Minor differences are noticeable; for instance, ProtoWiz wraps this.strokeStyle() in an unnecessary (but harmless) pv.color() because it does not recognize that this.strokeStyle() is already a color value. However, on the whole the code is quite similar, which should help users apply insights from the example gallery to ProtoWiz projects, and apply insights from ProtoWiz-based projects to future projects coded by hand.

**a**

```
x = pv.Scale.linear(0, 99).range(0, w);
y = pv.Scale.linear(0, 1).range(0, h);
c = pv.Scale.log(1, 100).range('#ff7f0e',
'#8c564b');


exampleMark = root.add(pv.Dot)
     .data(demo)
     .left(function(d) x(d.x))
     .bottom(function(d) y(d.y))
     .size(function(d) d.z)

     .strokeStyle(function(d) c(d.z))
     .fillStyle(function()
pv.color(this.strokeStyle()).alpha(0.2))
     .title(function(d) d.z.toFixed(1));
```

**b**

```
x = pv.Scale.linear(0, 99).range(0, w),
y = pv.Scale.linear(0, 1).range(0, h),
c = pv.Scale.log(1, 100).range("orange",
"brown");

vis.add(pv.Panel)
     .data(data)
     .add(pv.Dot)
     .left(function(d) x(d.x))
     .bottom(function(d) y(d.y))
     .strokeStyle(function(d) c(d.z))
     .fillStyle(function()
this.strokeStyle().alpha(.2))
     .size(function(d) d.z)
     .title(function(d) d.z.toFixed(1));
```

Figure 3: (a) shows Protovis example code, (b) shows the equivalent code generated by ProtoWiz

## 3.2 User tests

In addition to confirming that the internal architecture of ProtoWiz provides access to most features of Protovis, we were interested in learning whether the user interface enabled relatively inexperienced individuals to produce a visualization in a reasonable amount of time.

### 3.2.1 Participants

A total of five participants was recruited from the School of Information and the Energy and Resources Group, 4 males and 1 female. All participants were interested in information visualizations and had been exposed to software applications for generating them. Excel was mentioned as a common tool, and most participants had used at least one other software package, including R, STATA and Matlab. None of our participants claimed to have extensive or professional programming experience, but three people mentioned that if they copied code from examples, they could usually adapt it to their own purposes. This fits well the user group we are targeting: end-user programmers as defined in [6]: "people with expertise in other domains working towards goals for which they need computational support". This is important to remember because it speaks to the probability that advanced programming concepts are going to be understood by our target audience. As Ko et al. detail, most end-user programmers know what they need to do in a certain language to achieve a certain result, but they rarely understand the computer science behind it and are likely to be thrown off by unexpected errors. As we will see, this has ramifications for the design and further development of ProtoWiz' user interface.

Although some of our participants had looked at Protovis before, none had actually used it in any of their projects. The reasons for this ranged from "the frustrating lack of documentation"[1] to a lack of time to "dive in". Although these quotes are only anecdotal evidence, they may suggest that Protovis can be a challenge to learn for a certain user population.

### 3.2.2 Methodology

After an initial assessment interview to gauge the participants' experience with various visualization tool and techniques, participants were given a brief introduction to the interface. For the purpose of the user test, the application was pre-loaded with a three-dimensional dataset containing 100 rows. The initial display showed an x-axis with rules, a y-axis with rules and a scatterplot of circles based on the data. Participants were asked to explore the interface for 10-15 minutes and try any functionality they were curious about. They were encouraged to think aloud while playing around with the interface. To ensure that we would learn about specific problems with regards to the expressiveness of the interface, as much as possible we only responded to questions and did not jump in to assist the participants with a task they seemed to be struggling with.

After the unstructured exploration of the interface, participants were given a printout of an example in the Protovis gallery ("Gas and driving", Figure 4) based on a data set containing historical data on driving habits (miles driven per capita each year) and gasoline prices (adjusted for inflation). The participants were asked to reproduce the graph as faithfully as they could in 20 minutes. Before we started measuring time, we gave an explanation of the graphical decomposition approach in order to enable everyone to develop an appropriate strategy (analyse graph for different mark types, determine their order, add them in order and attach data to them). The strategy was not made explicit. As with the interface exploration task, we only responded to direct questions and recorded the participants' thoughts along the way.

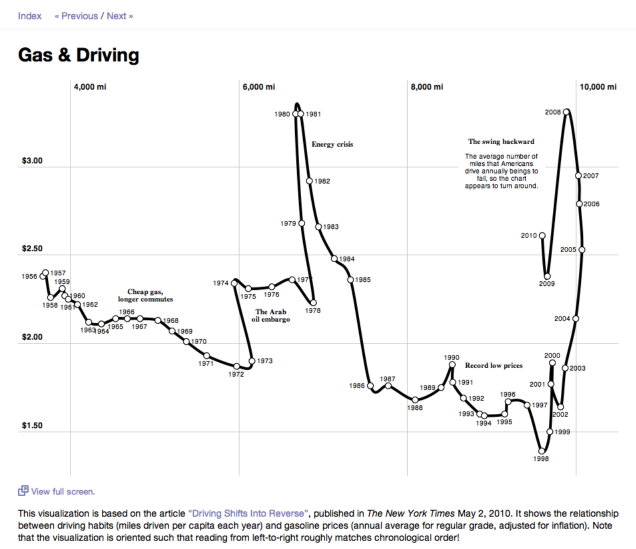


Figure 4: Example visualization from the Protovis Gallery

[1] *This should be taken less at a criticism of the documentation efforts of the Protovis team and rather as an indication that end-user programmers' needs are different from those of professional software developers. The participant voicing this complaint did not have enough programming experience to be able to use API documentation productively.*

### 3.2.3 Results

All participants were able to reproduce the example graph with varying degrees of fidelity. In one case, the graphic produced by the ProtoWiz user was nearly a pixel-perfect replica (see Figure 5). Also, participants varied in their ability to devise an appropriate strategy for creating the visualization unaided. The reasons for these varying difficulties are discussed below.

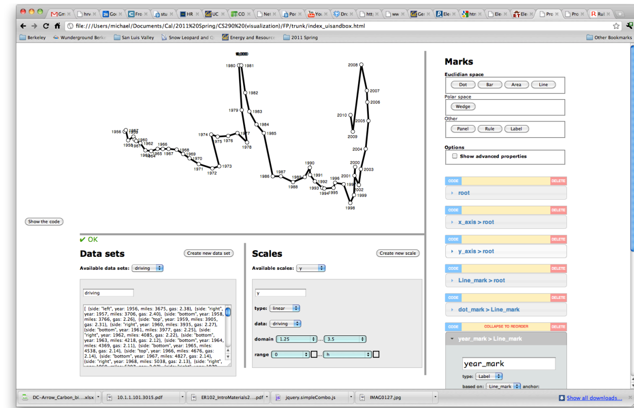


Figure 5: Re-creation of the example graph by one of our participants inside ProtoWiz

The concept of a mark that is associated with data appeared to be foreign to most of our users. Several times, we needed to explain that marks are conceptually different from their data-driven instantiations on the display. This proved to be a critical point as the only way to add a new visual element to the canvas is to create a new mark in the interface. Several participants tried to modify existing marks to add visual elements to the display and expressed initial confusion about the changes in the visualization.

When the concept of a mark was understood, most other mark-related operations (re-ordering by dragging, basing a mark on another mark etc.) were easily understood as well.

Further, the relationship of marks and scales was a stumbling block in many cases. In the user interface of ProtoWiz, Scales inhabit a different part of the screen than marks. The reasoning behind this design decision was that scales are non-visible elements that merely provide transformation functions to the marks that consume data points. Scales translate values from the data domain into the visual domain – which is why they need to be applied to the data that is being displayed. This connection did not seem readily apparent to the participants. In particular, there was considerable insecurity around the difference between applying and changing a scale. Both applying a scale to a given mark (or switching one scale for another) in the mark interface and altering the scale's properties in the scale interface immediately changed the visualization. Participants with a weak understanding of scales could not immediately make a connection between the changes they saw and the actions they had taken. This was compounded by some participants' tendencies to change many properties in many places hoping to get some effect. (A typical remark would be "Oh, that didn't do anything!").

This tendency had the most impact in positioning new marks which is usually the first and most essential step. For a mark to display itself correctly, it needs (at a minimum) a data source. Because ProtoWiz supports multiple data sets, setting the data source can be a two-step process: selecting a data set to draw from, and then setting the appropriate data value from the data set as the mark's data source. Due to the absence of strong typing in JavaScript, creating reliable type checking mechanisms is very difficult, and we had to opt not to do this. As a consequence, users were able to, for example, assign ordinal data values from the data set to marks that require numeric specifications to position themselves. The visualization fails gracefully in this case, but some

participants had to be nudged to reconsider their choice of data source.

For all marks that display themselves in Euclidian space, the specification of minimally necessary properties (horizontal position, vertical position and appearance) could be an initial barrier to understanding the relationship between mark manipulation and changes in the visualization display. The referenced Protovis example contained a line mark, a dot mark and a label mark, all of which need to be positioned in Euclidian space, and underspecifying marks leads to results that some participants struggled to interpret. For example, specifying a line mark with only an x-axis property leads to a flat horizontal line. Depending on the position of the browser's viewport, this change in display could be overlooked, especially when participants were focused on understanding the various form elements.

The accordion widget that contained the form elements for each mark contributed to some of these difficulties. To implement the layers metaphor explained in section 2.4, mark form elements are stacked on top of each other. To maximize screen space for each form element, an accordion widget organizes the display of the individual marks' forms. When a user adds a number of marks to the visualization, each new mark gets appended to the list of marks to reflect the order of Protovis code. When working with marks that were positioned at the bottom of the list, it was possible for the visualization to be partly scrolled outside the viewport. This led to the impression that nothing was happening on the screen when in fact, the upper half of the visualization might have changed.

Participants frequently commented on the ordering of form elements. In some instances, certain controls were not positioned in placed where they would have been expected. Because of the way form elements are currently generated on the fly when a mark updates its controls, the ProtoWiz interface has basic logical grouping of elements. This could easily be enhanced to create semantically consistent structures for control groups that are labelled according to their visual impact (e.g. "Positioning", "Colors" etc.) The layout of the form controls was deemed by some participants too be "overwhelming"; these cases should be resolved by applying both semantic grouping as described above and a refined visual hierarchy.

The existence of multiple data sets was generally well understood. Some participants were led to believe that they could directly alter the data by inputting new values which is not currently supported. In the long term, the HTML <textarea> that holds the JSON notation as a string should be replaced by a tabular representation of the data. This would also help users with discovering properties of the data that affect the display of the visualization. The example graph that we asked participants to re-create had labels aligned with the data points in a seemingly arbitrary fashion. A closer look at the data set would have revealed to the participants that each data row actually contained positioning information for the label to avoid visual occlusion. Had the data been presented in a tabular format, this might have been more obvious.

Data sets are in the current implementation grouped with scales because both represent non-visual parts of the visualization code. As with scales, some participants were expecting that selecting a data set in the data set viewer would apply the data to the entire visualization. Again, the concept that marks are conceptually separate from their data is most likely the cause for these issues.

A consistent source of questions were marks of the type Label. Most participants did not think of labels as marks, but rather as accessories *to* marks. Consequently, labels were often assumed to be controlled by the marks with which participants wanted to associate them. Almost every participant had to be pointed to the existence of a "Label" button in the panel from which they had created the other additional marks in their visualization.

#### 3.2.4 Discussion

Overall, ProtoWiz has proven to be a very successful tool for the kind of task that we tested it against. The fact that users without significant previous exposure to Protovis were able to reproduce a graph with minimal amounts of training proves that the concept is valid and should be pursued further. There are, however, numerous aspects to the current implementation of ProtoWiz that should be given more consideration.

The first and most prominent aspect is the observed difficulty of some novel users to understand the relationship between their manipulation of the interface and the resulting visualization. It appears that the gulf of evaluation and the gulf of execution are widened by the idiosyncratic nature of Protovis' approach to visualization.

The gulf of evaluation as Hutchins et al. [4] describe it, concerns the ability of the user to tie states of the generated visualization back to the settings of the interface. A number of remedies could be applied to the current implementation to support a clearer understanding of how the state of the visualization reflects the state of the mark forms, data sets and scales. Some of these are reflected in section 4, Limitations and future work.

The gulf of execution is the distance between the user's mental model of the application and the software's modelling of its domain. This becomes highly relevant when users form a strategy for attaining the goals they set. In our evaluation, this shows when users are clicking buttons and guessing what they will do ("Maybe this one? No, I'm doing it wrong…").

While some of the causes for this lie in the technical implementation of the interface, the larger issue seems to lie in the distance between Protovis' model of building visualizations and the users' understanding. This raises an interesting conceptual issue. Currently, ProtoWiz' manipulation interface mirrors very closely the underlying concepts; this extends down to the naming of properties. That is, to a large degree, intentional. We want ProtoWiz to be both a tool for quick and effective visualization creation, and a learning instrument. By showing users the code that is generated from their settings, and by enabling them to copy-paste the complete code to an HTML document for further development, we want to encourage users of ProtoWiz to further their understanding of ProtoVis.

As soon as one introduces abstractions from the Protovis model of creating visualizations, it becomes more difficult to correlate the state of the mark controls with the generated code. On the other hand, strict adherence to the ProtoVis model makes the tool more opaque and introduces a steeper learning curve for users who are newcomers to both Protovis and ProtoWiz.

The interesting design challenges lie in the space between. There are numerous ways to preserve the Protovis idiom in the interface while guiding users more strongly. In the areas where users experienced the most initial difficulty, there are the greatest opportunities for further improvement:

- Relationships between marks, scales and data
- Visualizing inheritance relationships both on a the level of the entire mark and on the level of its properties
- Either protecting the user from inadvertently breaking parent-child relationships or building in smart recovery mechanisms

Improvements

## 4 Limitations and Future Work

ProtoWiz is an ambitious concept, and much work remains to be done to fulfill the ultimate vision. Here we divide the possibilities into "limitations" which we do not intend to address in the near future because we believe they are conceptually inappropriate or out of our original scope, and "future work" that we believe will yield more value for the time invested. Of course, if the project

becomes popular and gains momentum, today's accepted limitations could become tomorrow's future work.

## 4.1 Limitations

### 4.1.1 Interactivity

The current version of Protovis (3.2) offers quite limited native support for interactivity. Interactivity is mainly achieved by writing standard JavaScript event handlers and attaching them to marks with convenience methods. Because this is a true JavaScript programming task and does not take advantage of the elegant mark-and-property architecture, we have left it out of scope for ProtoWiz. If Protovis in the future incorporates interactivity (or animated transitions) more directly, we would revisit this scoping decision. In the meantime, programmers are free to add interactivity to the generated Protovis code, of course.

### 4.1.2 Data

ProtoWiz currently supports the creation of data sets via pasting in JSON-formatted data. Although this format will be unfamiliar to non-programmers, there are simple and free web-based tools that will convert Excel spreadsheets or tabular data to JSON, such as Mr. Data Converter [8]. Thus, we do not expect data import limitations to be a significant barrier to the use of ProtoWiz, and see more research and practical value in focusing on simplifying the Protovis authoring process, at least in the near term.

### 4.1.3 Import of existing Protovis code

We have received several inquiries as to whether ProtoWiz will support importing of existing Protovis code into the user interface. While we certainly agree that this would be a valuable feature, it also implies the entirely new and mostly unrelated challenge of implementing a JavaScript parser. We would welcome contributions along these lines but intend to stay focused on the core user interface challenges for now.

### 4.1.4 Layouts

Protovis layouts would be cumbersome to include in the ProtoWiz architecture, as layouts define numerous idiosyncratic properties that would need to be defined individually in the ProtoWiz hierarchy. They also are not central to Protovis' strength in the flexible, "decompositional" creation of novel visualizations. However, excluding layouts does significantly constrain ProtoWiz's expressiveness especially in the important area of graph visualization, including hierarchies and networks. Of all the original limitations in scope, this is the most likely to be revisited first.

## 4.2 Future Work

Aside from the above structural or scoping limitations, there are numerous interface modifications that would clearly improve expressiveness, efficiency and/or accessibility but which have not yet been implemented due to time constraints. Some of these engender interesting research questions of their own. We briefly address them here.

### 4.2.1 Direct interaction with the canvas

The canvas is an important source of immediate feedback regarding the state of the visualization, but is a completely "passive" interface component, creating an artificial divide between the property definitions and their product. A more direct sense of manipulation could be provided by allowing users to click on an existing mark in the canvas to open its mark form, and/or fading the opacity of marks whose forms are not currently open for editing to draw attention to the mark being edited. More ambitiously, we can imagine enabling direct dragging and dropping of marks on the canvas to modify certain properties such as size. Developing an intuitive toolbox of direct transformations and applying those transformations intelligently to other instances of the mark would be a difficult but potentially quite fertile design challenge.

### 4.2.2 Improved error status messages

Currently, Protowiz provides feedback on the "renderability" of the generated code, either by displaying a check mark and "OK" in the status area, or by displaying an "X" mark along with the JavaScript error message encountered during rendering. This error message is quite low-level and cryptic, even for moderately experienced programmers. Two possible approaches to remedying this problem are 1) maintaining a translation table that relates JavaScript errors to their common causes in Protovis, and 2) pre- or re-parsing individual properties in a "sandbox" context in an attempt to detect where the error is being generated and why. It is an open question as to whether either of these approaches (or a combination) could cover a large enough fraction of possible issues to be worthwhile.

### 4.2.3 Expression editing

Another approach to reducing time spent addressing errors is to provide better guidance on what constitutes correct code. In our original conception of ProtoWiz we envisioned including an "expression builder" similar to Tableau that would allow guided construction of more complex operations on data fields and scales (or literals). Unfortunately this proved to be too ambitious for the time available, but there are clear precedents and reason to believe that users comfortable coding at the level of Excel formulas will find this approach familiar and usable, while allowing for more expressiveness than the current drop-down paradigm. This would also allow a more explicit introduction to common Protovis techniques such as using the ternary conditional operator to control a property based on some boolean decision about the data.

### 4.2.4 Drag-and-drop inheritance

Although there is some relationship between inheritance (as determined by the parentProperty) and visual layering in the ProtoWiz interface, we believe that the relationship could be made far more explicit by allowing users to drag-and-drop mark forms directly on to other mark forms to create (or break) inheritance relationships. Child mark forms could appear "nested" inside parent mark forms. This would allow us to do away with the drop-down parentProperty altogether and represent both inheritance and layering using a unified visual metaphor. Unfortunately the standard jQuery "accordion" component used to create the layer interface does not support this behavior, but it should be fully achievable with customization effort.

### 4.2.5 Undo, redo, save and load

ProtoWiz does not currently track changes in the internal state of its collection of marks and properties, and thus offers none of these common conveniences. Ahlberg and Schneiderman sensibly include "reversibility" among their requirements for "tight coupling,"[1] and thus we consider the lack of undo functionality to be a conceptual as well as practical weakness. Fortunately, there are no structural limitations to adding these features.

### 4.2.6 Tooltips

In general, ProtoWiz intentionally displays the exact property names used by Protovis. In many cases, these are easily understood, but some are obscure; for example "stroke style" where users would probably find "stroke color" more descriptive. This exemplifies an underlying design tension in ProtoWiz – that is, to what extent should the interface surface Protovis (or javascript) idiosyncrasies to help users learn to code, versus hiding those idiosyncrasies in order to make the tool more accessible to novices? Adding tooltips to each property would be a middle ground, allowing the Protovis names to remain "official" and become familiar while enabling quick discovery of their underlying meanings by new users.

#### 4.2.7 Missing Protovis features

A few basic components of Protovis are not yet supported due to their need for idiosyncratic or custom property definitions. These include:

- The Image mark type
- Drop-down guidance for the "font" and "text baseline" properties
- Formatting (e.g., of dates)
- Data manipulation.

Data nesting is technically supported by the custom *nestProperty* but in an extremely rudimentary way that would be difficult for users not already familiar with Protovis to get working correctly. Other data manipulations (e.g. rollups) have not been addressed at all.

## 5 Conclusion

We have presented ProtoWiz, a browser-based frontend for Protovis. Through a flexible and extensible architecture, we were able to replicate most of Protovis' functionality and automatically generate form controls that set an underlying Protovis visualization's properties.

## References

[1] C. Ahlberg and B. Shneiderman, "Visual Information Seeking: Tight Coupling of Dynamic Query Filters with Starfield Displays." [Online]. Available: ftp://ftp.cs.umd.edu/pub/hcil/Reports-Abstracts-Bibliography/3131html/3131.html. [Accessed: 06-Apr-2011].

[2] M. Bostock and J. Heer, "Protovis: A graphical toolkit for visualization," IEEE Transactions on Visualization and Computer Graphics, p. 1121–1128, 2009.

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

[4] Hutchins, Edwin L., Hollan, James D. and Norman, Donald A. (1985) "Direct Manipulation Interfaces", Human-Computer Interaction, 1: 4, 311—338

[5] "jQuery: The Write Less, Do More, JavaScript Library." [Online]. Available: http://jquery.com/. [Accessed: 06-May-2011].

[6] A. J. Ko, et al., "The State of the Art in End-User Software Engineering." ACM Computing Surveys. Accepted for publication.

[7] J. H. Maloney, K. Peppler, Y. Kafai, M. Resnick, and N. Rusk, "Programming by choice: urban youth learning programming with scratch," ACM SIGCSE Bulletin, vol. 40, no. 1, p. 367–371, 2008.

[8] "Mr. Data Converter - Transforming spreadsheets into web-friendly data since 2010." [Online]. Available: http://www.shancarter.com/data_converter/index.html. [Accessed: 06-May-2011].

[9] "Protovis: Gallery." [Online]. Available: http://vis.stanford.edu/protovis/ex/. [Accessed: 06-May-2011].

[10] "Simple Combo Box jQuery plugin << Semicolon." [Online]. Available: http://www.thunderguy.com/semicolon/2009/07/16/simple-combo-box-jquery-plugin/. [Accessed: 06-May-2011].

[11] C. Stolte, D. Tang, and P. Hanrahan, "Polaris: A system for query, analysis, and visualization of multidimensional relational databases," IEEE Transactions on Visualization and Computer Graphics, p. 52–65, 2002.