SIGRAD 2012
A. Kerren and S. Seipel (Editors)

Usability Analysis of Custom Visualization Tools

Mohammad A. Kuhail, Soren Lauesen, Kostas Pantazos, and Xu Shangjin

IT University of Copenhagen, Denmark

Abstract

Many visualization tools allow the implementation of custom (non-standard) visualizations, but they differ in approach. The approaches vary from imperative to declarative programming. Moreover, some tools provide environments that assist designers in implementing visualizations. Which approach supports designers best in implementing custom visualizations? What is lacking? To answer these questions, we compared the approaches of four recent visualization tools that support custom visualizations using an example. Further, we evaluated the approaches using the framework of the Cognitive Dimensions of Notations (CDs). Our findings favour notations that use declarative rather than imperative programming, and environments that allow exploration rather than dialogue-dependant ones.

Categories and Subject Descriptors (according to ACM CCS): H.3.4 [Systems and Software]: Performance evaluation (efficiency and effectiveness)—H.5.2 [User Interfaces]: Evaluation/methodology—

1. Introduction

Charting tools (e.g. MS Excel) support standard visualizations such as line charts, bar charts, etc. Designers only need to select data, and choose their visualizations based on predefined templates. However, *custom visualizations* are tailored to a specific need, and cannot be built like standard visualizations. Further, designers might not be exactly sure about what the desired visualization should look like. They need to explore various avenues to implement the visualization. It is a trial and error approach.

A visualization tool allows the user to set up a graphical presentation of data. Many visualization tools allow the implementation of custom visualizations, but they vary in approach. For instance, some tools [Fek04, HCL05] provide modules (e.g. visual objects, layout mechanisms) that can be used with traditional programming languages. Other tools [BH09] provide declarative domain specific languages that can combine visual objects, and define their properties to show data. Moreover, some tools [KPX13] provide development environments that use cognitive artefacts such as interface builder, direct manipulation, etc. to enhance designers' cognitive ability and ease the process of visualization design.

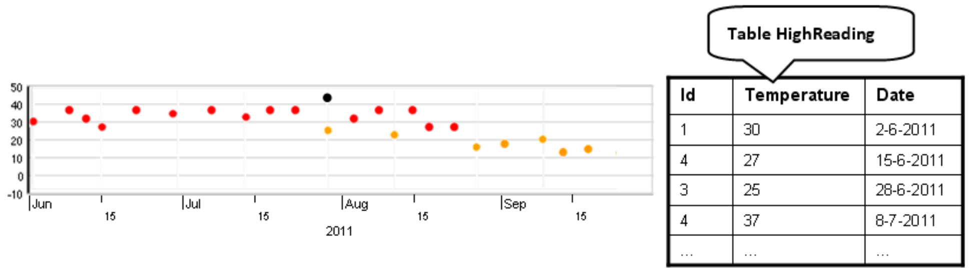
How do the existing approaches support custom visualizations? To what extent are the approaches sufficiently accessible to designers? What is lacking? To answer these questions, we selected four recent visualization tools: Prefuse, Improvise, Protovis, and Uvis. We investigated the tool approaches, and compared the solutions of the tools to a custom visualization. Furthermore, we used the framework of the Cognitive Dimensions of Notations (CDs) [Gre89], a framework for evaluating a notional system and the environments it is manipulated in, to evaluate the tools and identify areas that need improvement.

Our findings are in favour of notations that use declarative programming rather than imperative programming, and environments that support exploration rather than restrictive dialogue-dependant ones.

2. Related Work

A common approach to evaluating visualization tools is to conduct an experiment or an evaluation study that evaluates how well several tools support tasks . Examples can be found at [Byr99,SEH00,PGB02]. This approach provides some insight about the usability of the evaluated tools, but it is task-specific.

Broad evaluations of visualization tools exist. For instance, a recent study was carried out to evaluate how visualizations are constructed from a user perspective [PL12]. While this approach is insightful, it does not give details



Id	Temperature	Date
1	30	2-6-2011
4	27	15-6-2011
3	25	28-6-2011
4	37	8-7-2011
...	...	...

Figure 1: *A custom x-y graph based on table HighReading*

about the specifics (e.g. functionality, utility, etc.) of the tools. Another recent study compared the visualization and analysis functionalities, and environments of various visualization tools. However, little critique was given about the usability of the tools [HC12]. The authors of Protovis evaluated the accessibility (the effort required to create or modify a visualization) of Protovis [BH09], Flare [Fla], and Processing [Pro] using a subset of dimensions from the CDs framework [BH09]. The evaluation was brief, but it shed some light on a few aspects of the tools.

3. Evaluation Settings

3.1. Selection of Visualization Tools

We selected four visualization tools for evaluation: Prefuse [HCL05], Improvise [Wea04] , Protovis [BH09], and Uvis [LKP*13, KPL12, KL12]. We selected the tools based on support for custom visualizations, how recent they are, whether they are general-purpose, and difference of approaches.

All the selected tools support the creation of custom visualizations, have been developed in the last decade, and are general-purpose. Also the tools have different approaches to visualization creation. We only selected a representative tool from tools similar in approach or cognitive artefacts. For instance, we excluded Flare [Fla] since it adapted its design from Prefuse. Likewise, we excluded D3 [BOH11] as it borrows a lot of its concepts (e.g. helper functions) from Protovis.

Since they do not have direct cognitive support for visualization creation, we excluded programming languages, Graphics API's and GUI development systems such as Processing [Pro], Java2D, and Piccolo [BGM04].

3.2. Design

We introduce the selected visualization tools, and the support they provide for visual mappings [CMS99]. Visual mapping is a key to visualization expressiveness and effectiveness [SJ07]. It requires four essential *elements*: visual objects that show data graphically, a mechanism to bind visual object properties to data, a mechanism that supports complex arrangement of visual objects, and an environment that helps designers implement visual mappings. We compare the selected tools according to these four elements. Table 1 provides a summary for the comparison.

We compare the selected tools using a custom x-y graph (Figure 1). The example shows high readings of temperature in a given city. The readings are taken from 1 June 2011 to 1 October 2011. The dots represent the readings, and so far it looks like a conventional chart. However, we want to customize the colour of the dots. If the dot is showing the highest temperature, it is black. Otherwise, if the dot is showing a temperature greater than 25, it is red. The rest of the dots are orange. Although the example is simple, it was selected because it can be made with the selected tools without advanced knowledge. Further, it does not favour any of the tools.

We use the CDs framework to evaluate the tool support for custom visualizations. To effectively use the framework, we need to understand the nature of the task of implementing a custom visualization, and which cognitive dimensions are important to look at. The task qualifies as an exploratory task since it is a combination of incrementation (adding information without altering the structure) and modification (changing the existing structure possibly without adding new content), and the desired end might not be known in advance [GB98]. The cognitive dimensions that are important to look at when designing or evaluating tool support for exploratory tasks are: abstractions, hidden dependencies, premature commitment, progressive evaluation, viscosity, visibility, and juxtaposability [GB98]. Based on that, we selected the aforementioned dimensions for the evaluation.

Ideally, systems that support exploratory tasks (e.g. implementing a custom visualization) should have low viscosity, few hidden dependencies, few premature commitments, few abstractions, and high visibility and juxtaposability [GB98]. We use this as a criterion for evaluating how well the tools support custom visualizations.

		Prefuse	**Improvise**	**Protovis** (Protoviewer)	**Uvis** (Uvis Environment)
Environment		N/A	• WYSIWYG • Selection	• WYSIWYG • Selection	• WYSIWYG • Direct manipulation • Error highlighting • Inspector
Visual objects	**Primitive**	Java Shape (Ellipse, etc.)	Glyph (Rectangle, Oval, etc.)	Mark (Dot, Bar, Wedge, etc.)	Ellipse, Triangle, Box, etc.
	Specialized	Graph, TreeMap, etc.	BarChart, MatrixView, etc.	N/A	TimeScale, Spiral, etc.
Binding visual properties to data		• **Class** (Action)	• **Expression** (Projections)	• **Expression** (Anonymous Functions)	• **Expression** (Formulas)
Complex layout		• **Visual objects** (e.g. Tree Map.) • **Layout class** (e.g. force directed.)	• **Visual objects** (e.g. Tree, Graph, etc.)	• **Layout property** (e.g. tree map, force directed.)	• **Visual objects** (e.g. tree map, TreeNode.)

Table 1: *A summary of the selected tool approaches*

4. Approaches

4.1. Prefuse

Prefuse is a visualization toolkit suited for advanced visualizations (e.g. tree maps, sunburst, etc.). It provides modules (e.g. functions, layout classes, etc.) suited for various visualization tasks. To create visualizations, the designer writes Java code that uses the modules.

Prefuse provides primitive geometric visual objects (e.g. rectangles, ellipses, etc.) and specialized visual objects that are suited for specific visualizations such as trees and graphs. Prefuse provides many Action subclasses that bind visual properties to data. The designer can use specialized objects or a Layout subclass to accomplish complex arrangement of visual objects.

Example: Figure 2 shows the specifications of a custom x-y graph with Prefuse. First, a visualization object is created and bound to data (lines 1-3). Prefuse uses an `AxisLayout` abstraction that supports plots (lines 4 and 5). The to-be-visualized fields are passed in the `AxisLayout` constructor.

Actions bind visual properties to data (lines 8-10). There are many types of actions. For instance, `ColorAction` can make a colour property (e.g. border colour or background colour) show data. The `orangeColor` variable makes all visual objects orange (line 8). It sets the `FILLCOLOR` (background colour) of all visual objects to orange. However, the `redColor` variable makes objects that conform to a condition red (line 9). The condition is specified by a predicate that checks if the temperature fields are greater than 25. This predicate is specified at line 6. The actions are attached with the visualization object (lines 11-15).

The axes are positioned using a `RenderFactory` class (lines 17-20), and tick marks of the axes are generated using an `AxisLabelLayout` class (lines 21-24). The tick marks are associated with their corresponding axes (line 25).

Finally, ellipses are chosen as visual objects to represent the temperature readings, and associated with the axes defined previously (lines 26-28).

Summary: There are many abstractions that designers have to know and create (e.g. `AxisLayout`, `RenderFactory`). The separation of actions from the visualizations, predicates, and their properties can facilitate the management of code and allow reuse, but might increase the gap between the problem and the solution (Norman's gulf of execution [Nor86]). A designer might be wondering "which visual object or property does this action relate to?".

```
   Visualization vis = new Visualization();
a  Display display = new Display(vis);
   vis.add("HighReading", data);

b  AxisLayout x_axis = new AxisLayout(" HighReading ", "Date", Constants.X_AXIS, VisiblePredicate.TRUE);
   AxisLayout y_axis = new AxisLayout(" HighReading ", "Temperature", Constants.Y_AXIS, VisiblePredicate.TRUE);

   Predicate p1 = (Predicate)ExpressionParser.parse("Tempreature > 25");
   Predicate p2 = (Predicate)ExpressionParser.parse("Tempreature=MAX(Tempreature)");
c
   ColorAction Orangecolor = new ColorAction("data", VisualItem.FILLCOLOR, ColorLib.getColor(255, 128,0));
   ColorAction redColor =new ColorAction("data", VisualItem.FILLCOLOR,p1, ColorLib.getColor(255, 0,0));
   ColorAction blackColor =new ColorAction("data", VisualItem.FILLCOLOR, p2, ColorLib.getColor(255, 0,0));

   ActionList draw = new ActionList();
   draw.add(x_axis);
   draw.add(y_axis);
d  draw.add(redColor); draw.add(orangeColor); draw.add(blackColor);
   vis.putAction("draw", draw);

   vis.setRendererFactory(new RendererFactory()
   { AbstractShapeRenderer sr = new ShapeRenderer(7);
   Renderer arY = new AxisRenderer(Constants.FAR_Right, Constants.CENTER);
   Renderer arX = new AxisRenderer(Constants.CENTER, Constants.FAR_BOTTOM);});
e
   AxisLabelLayout x_labels = new AxisLabelLayout("xlab", x_axis);
   AxisLabelLayout y_labels = new AxisLabelLayout("ylab", y_axis);
   draw.add(x_labels);
   draw.add(y_labels);
   y_axis.setRangeModel(new NumberRangeModel(-10, 50, -10, 50));

   final Ellipse2D TempreatureEllipse = new Ellipse2D.Double();
f  x_axis.setLayoutBounds(TemperatureEllipse);
   y_axis.setLayoutBounds(TemperatureEllipse);
```

Figure 2: *Creating a custom x-y graph with Prefuse. a) binding the visualization to data, b) defining time and numeric axes, c) defining a conditional visual mapping, d) associating the visual mappings with the visualization, e) defining tick marks and associating them with the axes. f) defining ellipses representing the temperature readings*

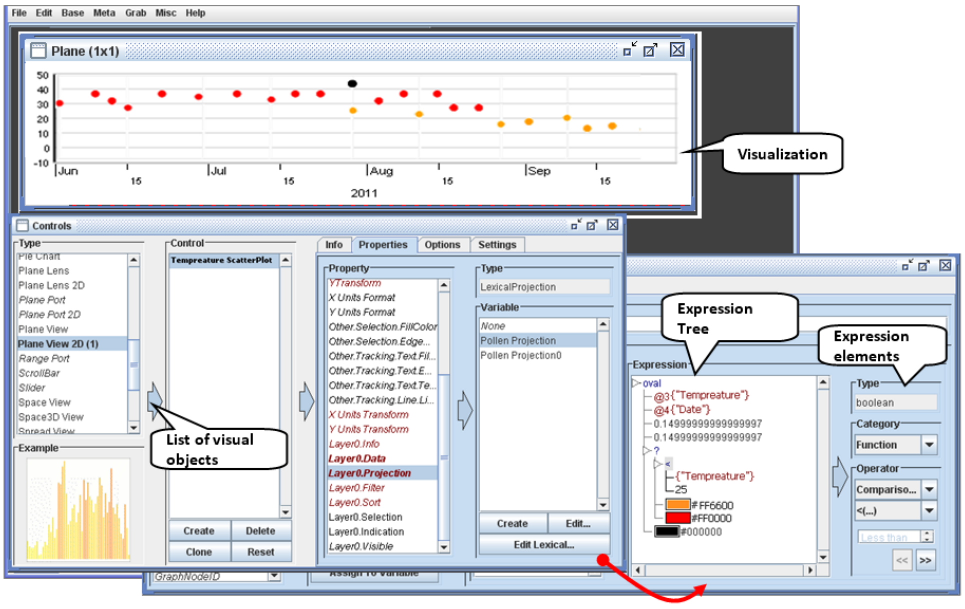
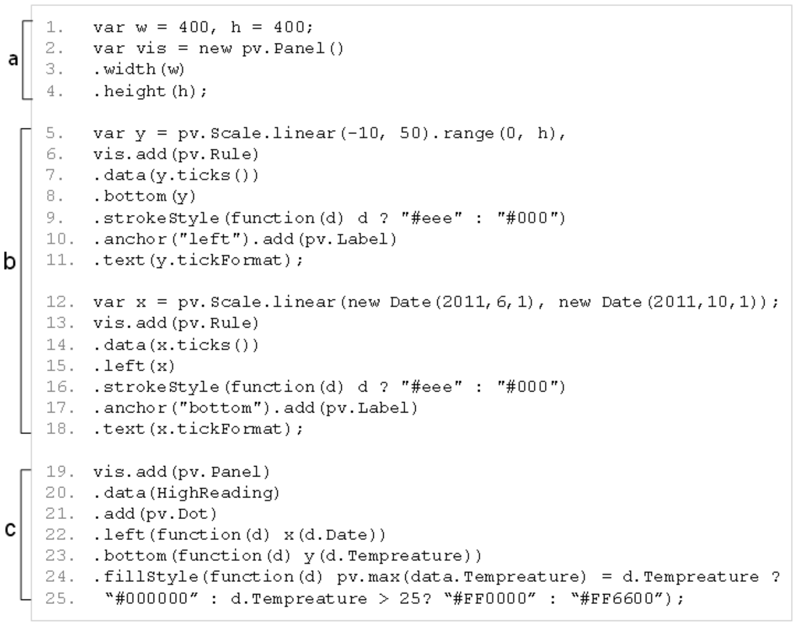


Figure 3: *Creating a custom x-y graph with Improvise*



```
var w = 400, h = 400;
var vis = new pv.Panel()
.width(w)
.height(h);

var y = pv.Scale.linear(-10, 50).range(0, h),
vis.add(pv.Rule)
.data(y.ticks())
.bottom(y)
.strokeStyle(function(d) d ? "#eee" : "#000")
.anchor("left").add(pv.Label)
.text(y.tickFormat);

var x = pv.Scale.linear(new Date(2011,6,1), new Date(2011,10,1));
vis.add(pv.Rule)
.data(x.ticks())
.left(x)
.strokeStyle(function(d) d ? "#eee" : "#000")
.anchor("bottom").add(pv.Label)
.text(x.tickFormat);

vis.add(pv.Panel)
.data(HighReading)
.add(pv.Dot)
.left(function(d) x(d.Date))
.bottom(function(d) y(d.Tempreature))
.fillStyle(function(d) pv.max(data.Tempreature) = d.Tempreature ?
 "#000000" : d.Tempreature > 25? "#FF0000" : "#FF6600");
```

Figure 4: *Creating a custom x-y graph with Protovis. a) defining the visualization. b) defining the numeric (temperature) and time scales (axes). c) defining dots and visually mapping them to temperature and date fields according to the scales*

4.2. Improvise

Improvise is a visualization system that mainly supports coordinated visualizations. It provides primitive and specialized properties whose visual properties can show data using expressions. The expressions can be conditional, logical, mathematical, etc. Improvise provides specialized objects that support complex layouts such as trees. Designers use a development environment to create a visualization. They navigate from panel to panel to accomplish visual mappings. Each panel has a distinct purpose. For instance, one panel shows the available visual objects and their properties. Another panel shows the variables that can be used in expressions.

Example: To define a x-y graph, the designer chooses `Plane View 2D` object from the list of visual objects. To define visual mappings for the visual object, the designer chooses `Layer.Projection` from the list of properties. He clicks "Create" to create a new projection (visual mapping). This leads him to a new panel (`Lexicon`) where he can define expressions.

We want to define this expression for the background colour property.

`Temperature > 25 ? "red": "orange"`

This expression has to be built step-by-step using combo boxes that provide the available Expression elements (Figure 3). First, the designer creates the conditional part of the expression by choosing `Function` from the `Category` combo box, and `Other` and `?(boolean,Color,Color)`. Improvise shows the result as a conditional expression tree with default colours as results for the true and false expressions.

Second, the designer can manipulate the conditional statement parts by clicking the tree nodes. To create a comparison condition, the designer chooses `Function` from the `Category` combo box, and `Comparison` and `>(...)` from the `Operator` combo boxes. Third, to make one of the nodes refer to the `Temperature` field, the designer clicks the node and chooses `Attribute` from the `Category` combo-box. Improvise displays the available fields, and the designer just selects (clicks) it.

Summary: In general, visual mappings rely heavily on dialogues. For instance, even a simple expression takes long to create. The environment forces the designer to use comboboxes that have the expression elements. It is not easy to find the expression elements. Moreover, the longer the expression, the harder it is to read.

4.3. Protovis

Protovis is a JavaScript-based visualization toolkit that uses a declarative domain specific language that can map data into primitive visual objects (e.g. bar, dot, etc.) and their properties. Protovis does not provide specialized objects for visualizations with complex layouts, but provides a Layout property that can arrange visual objects in various ways. Protovis can be extended with a development environment called Protoviewer [Aka11].

Example: Figure 4 shows the specifications of a custom x-y graph with Protovis. First, a visualization object is defined (lines 1-4). Protovis uses non-visual scale classes for creating time and numeric axes (lines 5-11). The designer uses them to generate tick data. `Rule` and `Label` visual objects are used to draw the axes based on the tick data (lines 12-18).

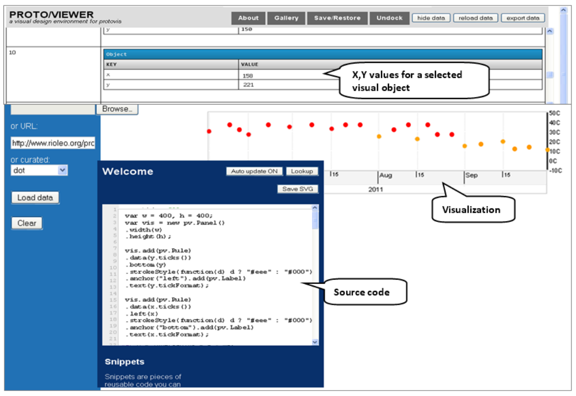
`Dot` objects are bound to data (an array that corresponds to the `HighReading` table) (lines 19-21). The `Left` and `Bottom` properties position the `Dot` objects horizontally and vertically (lines 22 and 23). The designer specified expressions for the two properties that call functions provided by the scales that calculate the positions based on temperature and date fields. Finally, a conditional expression for the `FillStyle` (background colour property) sets the colour of dots that show the highest temperature black. Otherwise, it sets the colour red for dots showing temperature greater than 25 red. Otherwise, they are orange (lines 24 and 25).

Development Environment (Protoviewer): The visualization can be built with the Protoviewer development environment (Figure 5). This has several advantages. Designers can see the resulting visualization immediately as they are modifying the source code. Moreover, clicking a visual object, designers can view the position values (x and y) of the object. This can help inspecting the object.

Summary: Protovis provides non-visual scale classes that facilitate the construction of axes. The axes are not defined directly. Instead, primitive objects such as `Label` and `Rule` are used for drawing the axes. This separation increases flexibility (e.g designers might obtain a custom axis in this way), but increases the steps of such a common task. Unlike Prefuse actions, the declarative expressions for the `Dot` visual objects are not separated from the visual properties. This increases visibility and understandability.

4.4. Uvis

Uvis is a visualization tool that allows creating custom visualizations based on relational data. To construct a visualization, the designer drags and drops visual objects (building blocks), binds their visual properties with data using spreadsheet-like formulas, and the environment shows the resulting visualization in a WYSIWYG fashion. To see properties of a visual object, the designer selects (clicks) the visual object, and the environment shows the visual properties of the object in a property grid. To make a visual property (e.g. `Height`) depend on data, the designer types a declarative spreadsheet-like expression (formula). Uvis supports conditional, logical, and mathematical formulas. Moreover, a formula can refer to data fields, visual properties, and functions. Uvis supports complex algorithms with specialized objects such as tree maps.



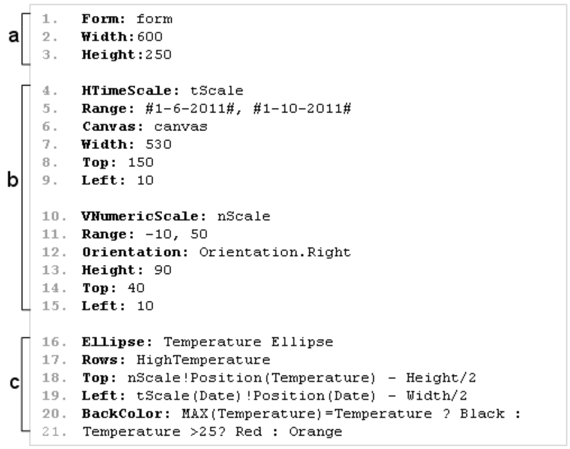
y	150

10

Object	
KEY	VALUE
x	158
y	221

```
var w = 400, h = 400;
var vis = new pv.Panel()
.width(w)
.height(h);

vis.add(pv.Rule)
.data(y.ticks())
.bottom(y)
.strokeStyle(function(d) d ? "#eee" : "#000")
.anchor("left").add(pv.Label)
.text(y.tickFormat);

vis.add(pv.Rule)
.data(x.ticks())
.left(x)
.strokeStyle(function(d) d ? "#eee" : "#000")
.anchor("bottom").add(pv.Label)
.text(x.tickFormat);
```

Figure 5: *Creating a custom x-y graph with Protovis environment (Protoviewer)*



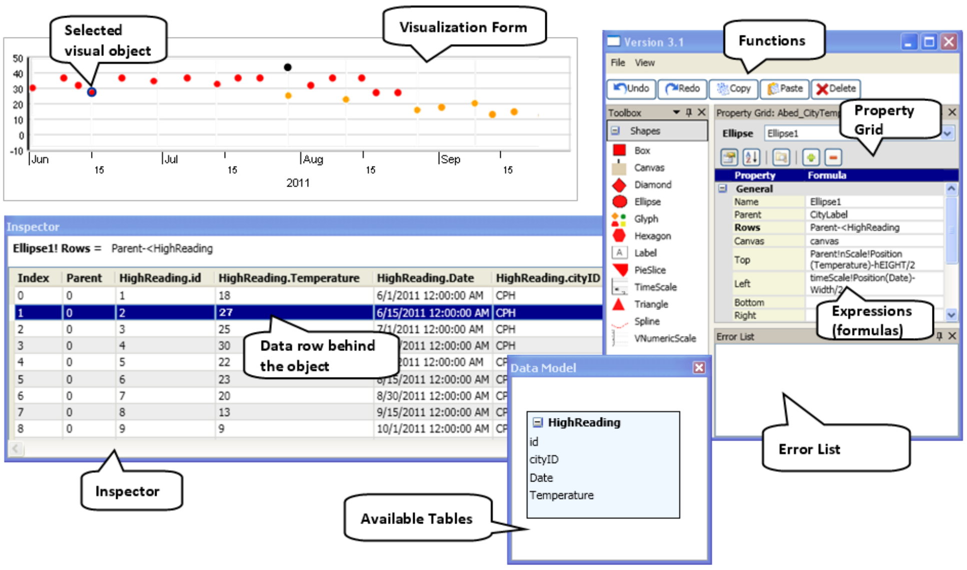
```
1.  Form: form
2.  Width:600
3.  Height:250

4.  HTimeScale: tScale
5.  Range: #1-6-2011#, #1-10-2011#
6.  Canvas: canvas
7.  Width: 530
8.  Top: 150
9.  Left: 10

10. VNumericScale: nScale
11. Range: -10, 50
12. Orientation: Orientation.Right
13. Height: 90
14. Top: 40
15. Left: 10

16. Ellipse: Temperature Ellipse
17. Rows: HighTemperature
18. Top: nScale!Position(Temperature) - Height/2
19. Left: tScale(Date)!Position(Date) - Width/2
20. BackColor: MAX(Temperature)=Temperature ? Black :
21. Temperature >25? Red : Orange
```

Figure 6: *The specifications of the custom x-y graph with Uvis*

Example: Figure 6 shows the textual specification of the custom x-y graph with Uvis and Figure 7 shows the environment where the chart was developed.

To create the time and numeric axes, the designer dragged `HTimeScale` and `VNumericScale` visual objects from the toolbox and dropped them on a form. The designer moved and resized them until they looked right. The environment sets position properties (i.e. `Top`, `Height`, etc.) accordingly. To define the range of time and numbers the scales show, the designer typed the value of the `Range` property in the property grid (lines 5 and 11 in Figure 6).

To create dots representing the temperature reading, the designer drags and drops an `Ellipse`. The designer typed formulas for the position properties (`Top` and `Left`). The formulas call position functions provided by the scales to calculate the positions based on temperature and date fields



Inspector

Ellipse1! Rows = Parent-<HighReading

Index	Parent	HighReading.id	HighReading.Temperature	HighReading.Date	HighReading.cityID
0	0	1	18	6/1/2011 12:00:00 AM	CPH
1	0	2	27	6/15/2011 12:00:00 AM	CPH
2	0	3	25	7/1/2011 12:00:00 AM	CPH
3	0	4	30	/2011 12:00:00 AM	CPH
4	0	5	22	011 12:00:00 AM	CP
5	0	6	23	/15/2011 12:00:00 AM	CP
6	0	7	20	8/30/2011 12:00:00 AM	CP
7	0	8	13	9/15/2011 12:00:00 AM	CP
8	0	9	9	10/1/2011 12:00:00 AM	CP

Ellipse Ellipse1

Property	Formula
General	
Name	Ellipse1
Parent	CityLabel
Rows	Parent-<HighReading
Canvas	canvas
Top	Parent!nScale!Position (Temperature)-hEIGHT/2
Left	timeScale!Position(Date)-Width/2
Bottom	
Right	

Data Model

HighReading
id
cityID
Date
Temperature

Figure 7: *Uvis environment*

(lines 18 and 19 in Figure 6). Finally, a conditional expression for the `BackColor` (background colour property) sets the colour of ellipses (line 21 in Figure 6).

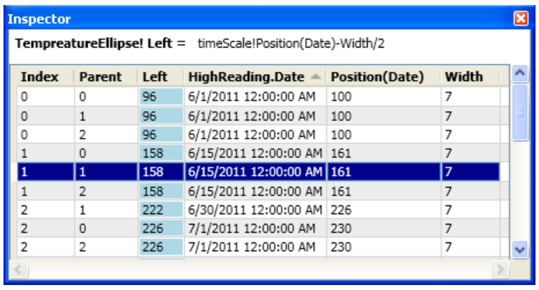
Development Environment: The environment has several advantages. Designers can drag, drop, resize visual objects (Direct manipulation), and they can see the resulting visualization immediately as they are updating the expressions.

The inspector shows data for a bundle of visual objects. It shows the data rows behind the visual objects (Figure 7). Further, it shows the values of an expression and its subexpressions (Figure 8).

Summary: Unlike Prefuse, Protovis, and Improvise, Uvis deals only with visible visual objects. Like Protovis, Uvis uses declarative expressions that directly define the visual properties, but there is no need to define variables, and the sequence of specifying the expressions is free. Like Improvise, the environment shows the available visual objects, but it allows the designers to drag, drop, and resize them (as long as the position and size properties do not have dynamic expressions) rather than textually setting them.

5. Cognitive Dimensions of Notations

This section evaluates how the selected tools perform in a relevant set of cognitive dimensions. The dimensions themselves are not sufficient to make a judgement. Therefore, we make the judgement based on what is desirable for exploratory tasks such as implementing a custom visualization. For instance, exploratory tasks require high visibility. Hence, tools that have high visibility rate high.


Inspector

TempreatureEllipse! Left = timeScale!Position(Date)-Width/2

Index	Parent	Left	HighReading.Date	Position(Date)	Width
0	0	96	6/1/2011 12:00:00 AM	100	7
0	1	96	6/1/2011 12:00:00 AM	100	7
0	2	96	6/1/2011 12:00:00 AM	100	7
1	0	158	6/15/2011 12:00:00 AM	161	7
1	1	158	6/15/2011 12:00:00 AM	161	7
1	2	158	6/15/2011 12:00:00 AM	161	7
2	1	222	6/30/2011 12:00:00 AM	226	7
2	0	226	7/1/2011 12:00:00 AM	230	7
2	2	226	7/1/2011 12:00:00 AM	230	7

Figure 8: *The inspector showing property Left expression values*

5.1. Abstractions

The abstractions dimension assesses the abstractions that encapsulate implementation details and the mechanism to manage them. Although abstractions can make the specifications shorter and sometimes fit the domain better, systems that require learning many abstractions have an *abstraction barrier*. Exploratory tasks do not tolerate many abstractions.

Prefuse is an example of a system that has an abstraction barrier. For instance, there are many subtypes of `Layout`, `RenderFactory`, and `Action` to learn. The abstractions can be extended programmatically by Java programming, but this requires in-depth knowledge of Java.

Protovis has fewer abstractions to learn than Prefuse, but some programming abstractions (e.g. variables, anonymous functions) are necessary to learn. Protovis abstractions can be extended programmatically with JavaScript.

Like Prefuse, Improvise has many abstractions. For instance, there are many panels and expression parts (e.g. conditional statements, functions, etc.) and the designers need to be aware of their meaning, and how to manipulate them, etc. Like Prefuse, Improvise abstractions can be extended with Java.

Uvis formulas resemble spreadsheet expressions, but obviously have more abstractions than spreadsheets. For instance, a Uvis formula can refer to data fields, visual properties, etc. However, Uvis has relatively few abstractions. For instance, there are no variables and rendering objects. Uvis does not allow defining new abstractions.

5.2. Hidden Dependencies

The hidden dependencies dimension assesses whether dependencies between entities are hidden or visible. Hidden dependencies slow down information finding and can potentially increase the risk of error. Exploratory tasks tolerate only a few hidden dependencies.

Most Prefuse abstractions have hidden dependencies. For example, the layout action implicitly overrides a specific visual mapping of size and position properties.

Protovis expressions can depend on variables. Such dependencies can be hard to see in textual specifications. More advanced visualizations use layout classes that position visual items implicitly (e.g. tree maps), or some operators such as "Parent" and "Sibling" that have hidden dependencies.

In Improvise, it is hard to derive the elements of an expression, particularly, if the expression contains variables or other sub-expressions. These can be viewed in other panels.

Uvis formulas can depend on other visual properties. The properties can have their own formulas, and so on. When designers change an expression, it is hard to know the implications of such a change. Furthermore, more advanced visualizations such as hierarchical visualizations use operators (e.g. `Parent`) that result in hidden dependencies.

All the surveyed tools except for Uvis do not explicitly show which particular visual property depends on which field. The Uvis environment shows that using the inspector (Figure 8).

5.3. Premature Commitment

The premature commitment dimension assesses whether there are any constraints on the order in which tasks must be accomplished. Premature commitment is harmful for exploratory tasks.

Since the specifications are program-like, Prefuse and Protovis impose constraints on the sequence in which visualizations are defined. For instance, if a property depends on another, the independent one has to be defined first.

Improvise imposes a strict sequence on how some things are done. Constructing the expression step-by-step is an example of strict sequencing, and having to navigate from panel to panel to carry out visual mappings is another one.

Uvis specifications are sequence-free. At run time, the kernel finds out the sequence of execution. If the designer types a formula that refers to a property that does not exist yet, Uvis kernel flags an error, but the application still runs.

5.4. Progressive Evaluation

The progressive evaluation dimension assesses how easy it is to evaluate and obtain feedback on an incomplete task. Progressive evaluation is important for exploratory tasks.

In Prefuse, it is not easy for a designer to obtain visual feedback of the specifications. The source code has to be run in another setting to obtain feedback.

Improvise bridges that gap with an immediate visual feedback feature. However, the visual feedback can be overshadowed with many editing panels.

Protoviewer and the Uvis environment provide a separate design panel that is updated immediately when the specifications are changed. The Uvis environment provides similar kinds of feedback as traditional environments such as highlighting erroneous formula parts, error, and warning lists. In addition, the environment shows the formula values in a separate panel that is updated when the formula changes (Figure 8).

5.5. Viscosity

The viscosity dimension assesses the cost of making small changes. It is costly to make a small change in viscous systems. Viscosity is harmful for exploratory tasks. We consider two types of viscosity. First, *repetitive viscosity* means a single goal-related change which requires many repetitive actions. Second, *knock-on viscosity* means a change in one part affects other related parts.

Prefuse is based on an object oriented language (Java.) Hence, inheritance can reduce repetitive viscosity. For instance, a change can be made in a parent class rather than all inheriting classes. Modern development environments can help with small knock-on changes such as changing a variable name that is used in many places (re-factoring.) Nevertheless, changing Prefuse specifications requires in-depth knowledge of the language constructs and programming concepts.

Like Prefuse, the Protovis language has low-repetitive viscosity since it supports inheritance for visual objects. Moreover, Protovis allows other changes easily, for instance, changing the visual object type. The environment (Protoviewer) does not have support for making changes.

Designers who are experienced with Improvise might find some things easy to change. For instance, variables that are referred to from many expressions can be changed in one setting. Otherwise, Improvise is highly viscous. For instance, changing some specialized visual object types (e.g. Plane View) is not possible. In general, a change in Improvise requires navigating across panels.

Like spreadsheets, simple visualizations in Uvis have low viscosity. However, viscosity grows with size. Uvis does not support inheritance, but designers can add properties that have formulas that other visual objects can refer to. In such a case, a change is only required in the designer property. Since Uvis formulas can refer to other formulas elsewhere, a change in one formula might affect other dependant formulas. The Uvis environment shows errors that result from such a change.

5.6. Visibility and Juxtaposability

The visibility dimension assesses the ability to view data components easily. Juxtaposability assesses the ability to view two similar components side by side. The two dimensions are generally discussed together due to similarity. Both dimensions are important for exploratory tasks.

What data components would a designer want to view when implementing a custom visualization? Many can be considered important. Examples include the currently-designed visualization, the available visual objects and their properties, the visual mappings, the available data, the visualized data, and errors. What needs to be viewed varies from task to task and designer to designer, but a possible solution is to give designers the ability to show or hide components.

Even if Prefuse is integrated with a development environment, only a few components can be visible in one setting. Traditional environments show the source code, the available visual objects, and a list of errors in one setting. However, the designer has to view the currently-designed visualization in another setting.

Protoviewer shows the currently-designed visualization as well as the specifications behind it. Furthermore, designers can view the position property values of a single selected visual object at a time. Protoviewer does not provide support for comparing the specifications of two similar visual objects.

Improvise shows the currently-designed visualization, but it can be over-shadowed by the editing panels. A panel can only show one expression at a time, and it occupies a lot of space. This does not allow comparing many expressions. Further, many data crucial for the task (e.g. data fields) are buried in combo boxes.

Uvis shows the currently-designed visualization, the properties (and the expressions defining them) of a selected visual object, and a list of errors. Upon selecting a visual object, Uvis shows the data behind that particular object. Further, to allow comparison, the data from other visual objects from the same data source are shown as well. It is also possible to see the defining expressions of all properties of a selected visual object. However, it is not possible to see expressions of two visual objects at the same time.

6. Conclusion

We summarize the findings of the comparative analysis and the evaluation with CDs as follows.

- **All the surveyed tools** suffer from low juxtaposability and high hidden dependencies with slightly different degrees.
- **All the surveyed tools except for Uvis** suffer from high premature commitment and low visibility with slightly different degrees.
- **Prefuse** uses a programmatic approach that relies on specialized modules. The main strength of this approach is the breadth of visualizations it can express due to the many modules it provides. However, there are many abstractions to learn even to construct a simple example like a custom x-y graph. Furthermore, even with a development environment, the approach suffers from low progressive feedback.
- **Improvise** uses an approach that is heavily dependant on dialogues (panels). The main strength of this approach is that the tool provides useful visual objects tailored for some tasks. However, the functionalities are not easy to find. For instance, the conditional expression is buried in a combo box item called "Other".
- **Protovis** uses an approach that relies on primitive visual objects and declarative expressions. The main strength of the approach is that the properties of the visual objects are directly specified. No middle-ware objects (e.g. Prefuse actions) are needed to link visual properties with expressions. However, some programming abstractions (e.g. variables) are still needed to learn the language.
- **Uvis** uses an approach that relies on declarative spreadsheet-like formulas for visual mappings, and a dedicated environment with many features (e.g. drag-drop, visual feedback, etc.). The approach has high visibility, low premature commitment, and relatively few abstractions to learn. However, the approach still suffers from high viscosity (especially when it is a large-sized application).

References

[Aka11] AKASAKA R.: Protoviewer: a web-based visual design environment for protovis. In *ACM SIGGRAPH 2011 Posters* (New York, NY, USA, 2011), SIGGRAPH '11, ACM, pp. 85:1–85:1. 24

[BGM04] BEDERSON B. B., GROSJEAN J., MEYER J.: Toolkit design for interactive structured graphics. *IEEE Trans. Software Eng. 30*, 8 (2004), 535–546. 20

[BH09] BOSTOCK M., HEER J.: Protovis: A graphical toolkit for visualization. *IEEE Trans. Vis. Comput. Graph. 15*, 6 (2009), 1121–1128. 19, 20

[BOH11] BOSTOCK M., OGIEVETSKY V., HEER J.: D^3 data-driven documents. *IEEE Trans. Vis. Comput. Graph. 17*, 12 (2011), 2301–2309. 20

[Byr99] BYRD D.: A scrollbar-based visualization for document navigation. In *ACM DL* (1999), pp. 122–129. 19

[CMS99] CARD S. K., MACKINLAY J. D., SHNEIDERMAN B.: *Readings in information visualization - using vision to think*. Academic Press, 1999. 20

[Fek04] FEKETE J.-D.: The infovis toolkit. In *INFOVIS* (2004), pp. 167–174. 19

[Fla] FLARE: Data visualization for the web. http://flare.prefuse.org/. [Online; accessed June-2012]. 20

[GB98] GREEN T., BLACKWELL A.: Cognitive dimensions of information artefacts: a tutorial. *T.R.G. Green and A.F. Blackwell 1*, 2 (1998). 20

[Gre89] GREEN T. R. G.: 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* (New York, NY, USA, 1989), Cambridge University Press, pp. 443–460. 19

[HC12] HARGER J., CROSSNO P.: Comparison of open-source visual analytics toolkits. In *SPIEConference on Visualization and Data Analysis* (2012). 20

[HCL05] HEER J., CARD S. K., LANDAY J. A.: prefuse: a toolkit for interactive information visualization. In *CHI* (2005), pp. 421–430. 19, 20

[KL12] KUHAIL M. A., LAUESEN S.: Customizable visualizations with formula-linked building blocks. In *GRAPP/IVAPP* (2012), pp. 768–771. 20

[KPL12] KUHAIL M. A., PANDAZO K., LAUESEN S.: Customizable time-oriented visualizations. In *ISVC (2)* (2012), pp. 668–677. 20

[KPX13] KOSTAS PANTAZOS MOHAMMAD A. KUHAIL S. L., XU S.: Constructing visualizations with a development environment. In *Submitted to: VDA* (2013). 19

[LKP*13] LAUESEN S., KUHAIL M. A., PANDAZOS K., XU S., ANDERSEN M. B.: A drag-drop-formula tool for custom visualization. 20

[Nor86] NORMAN D. A.: *User Centered System Design: New Perspectives on Human-computer Interaction*. CRC Press, 1986. 21

[PGB02] PLAISANT C., GROSJEAN J., BEDERSON B. B.: Spacetree: Supporting exploration in large node link tree, design evolution and empirical evaluation. In *INFOVIS* (2002), pp. 57–64. 19

[PL12] PANTAZOS K., LAUESEN S.: Constructing visualizations with infovis tools - an evaluation from a user perspective. In *GRAPP/IVAPP* (2012), pp. 731–736. 19

[Pro] PROCESSING:. http://processing.org/. [Online; accessed Aug-2012]. 20

[SEH00] SUTCLIFFE A. G., ENNIS M., HU J.: Evaluating the effectiveness of visual user interfaces for information retrieval. *Int. J. Hum.-Comput. Stud. 53*, 5 (2000), 741–763. 19

[SJ07] SEARS A., JACKO J. A.: *The Human-Computer Interaction Handbook: Fundamentals, Evolving Technologies and Emerging Applications*. CRC Press, 2007. 20

[Wea04] WEAVER C.: Building highly-coordinated visualizations in improvise. In *INFOVIS* (2004), pp. 159–166. 20