NapkinVis: Rapid Pen-Centric Authoring of Improvisational Visualizations

William O. Chao*

University of British Columbia

Abstract

This paper designs and implements a web-based, pen-centric front end for the Protovis toolkit, allowing users to quickly create visualizations for improvisational purposes. The design of this system is constrained to the scope of visualizations that you would be able to sketch on a paper napkin. Even within this limited canvas size, it is shown that by creating visual interactions for authoring visualizations as a combination of separable marks, one can produce a wide variety of visualizations in a matter of seconds without needing to write a single line of code.

Keywords: Design

Index Terms: Visualization, User Interface, Toolkits

1 Introduction

Can you think of a time when you wished you could quickly sketch a visualization to illustrate a point? Was it while teaching a graphics class? Perhaps it was while collaborating with a group on a project? Or possibly it was to look at something quickly while brainstorming a new idea, and not interrupting your flow of work in the process? This paper is aimed at exploring fast creation of visualizations for improvisational purposes, whether it is quickly demonstrating data live in a presentation, visualizing to move a conversation in a collaborative process, informally toying around with visualization ideas to see how they'd look, or even teaching information visualization principles in a classroom setting.

Anecdotally, many exceptional ideas have begun as simple sketches and doodles on paper napkins, whether it'd be at an inpromptu meeting with a colleage at a restaurant, creatively contemplating different thoughts in a coffee shop, or simply coming up with an epiphany outside the lab. This type of interaction was the inspiration for providing a number of creative constraints to this project by asking ourselves the following question: "if you could somehow quickly visualize something on a paper napkin sketch, how would you do so, and what could you visualize?" Fortunately, in addition to paper napkin sketches, restrictions for making visualizations quickly on small interfaces using only a pen-based interaction can also be applied to existing devices such as tablets, smart phones, interactive tables, pico-projector based interfaces, and other emerging technologies, so we feel that they serve as good guidelines for interactions that can already prove to be useful today.

So given these creative constraints, how can we approach authoring visualizations quickly?

Dept. of Computer Science, UBC
201-2366 Main Mall, Vancouver, B.C., V6T 1Z4
**email: wochao@cs.ubc.ca*

2 Related Work

There are several existing tools which are aimed at simplifying the production of visualizations. Excel and similar spreadsheet programs take a wizard-based approach to automatically generate visualizations based on predefined templates. Tableau [4] takes this a step further and allows for quick visual encodings of data dimensions to visualizations by drag and drop to predefined areas on the screen. At a more general pen-based active animations level, programs designed for kids such as eToys [2] or Phun [3] allow the creation of robots and simulations, which can often act as interesting visualizations if constructed cleverly. The former acts as a visual programming language while the latter is a pen-enabled physics simulation, both allowing for triggering of events, enabling user directed timing if used in a presentation setting. Other systems come even closer to the mark of sketch-oriented design [1]. At an even more general level, several toolkits exist to allow for the prototyping of visualizations. Protovis [5] provides a declarative Javascript toolkit which allows for a very rich set of visualizations to be specified and programmed in minutes. Prefuse [6] and Flare [7] are equivalent toolkits which allows users to specify data bindings, visual encodings, rendering, and control of visualizations to enable production of a very wide variety of visualizations. Unfortunately, the learning curve of Prefuse and Flare is very steep. Other toolkits with predefined visualizations also exist, however some argue about the ease of extending these visualizations [8, 9, 10]. Another flavour of toolkit is also available to the designer. Processing [11] is another toolkit which lets designers focus on programming visual marks without worrying about any complicated underlying programming structures (such as initializing many abstract classes). Unlike protovis, this doesn't provide shortcut marks with well defined anchors which can be bound to data or other marks, which leaves the logic of certain visualizations such as stacked area graphs too difficult for the everyday user. From this, tools become more general and more difficult to effectively design visualizations, such as ActionScript, Flash, OpenGL, etc.

This project aims to ultimately create an easily accessible tool with a flexibility of creating visualizations and a speed of interaction somewhere in between Tableau and Protovis.

3 Overview

NapkinVis is a web-based application which allows the user to quickly generate simple visualizations on a canvas of limited space, and in a time frame of under a minute. This is done by providing the user with a sketch-based means of encoding data on to single marks such as wedges or bars, and then combining the marks to form compound visualizations. NapkinVis is intended as a tool to quickly (and possibly sloppily) visualize data on-the-fly, rather than as a final authoring tool for more formal formats such

as when writing reports. As a result, the created visualizations may not be as precise as visualizations specified by hand, but a good approximation can be created rather rapidly to give a good qualitative feel of the data.

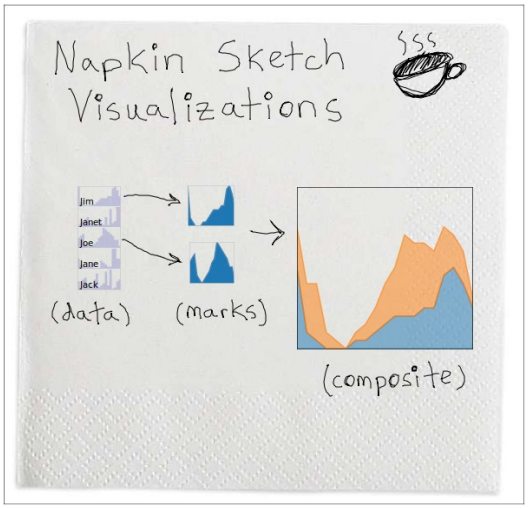

Figure 1. Sample screenshot NapkinVis with some sample visualizations and sketches

3.1 Work-flow

The process of creating visualizations in NapkinVis follows a work-flow that is centred along an idea suggested by Bostock and Heer [5] that visualizations can be thought of as a combination of individual marks, as seen in figure 2. This organizing idea is central to the design of Protovis. By adopting this idea, and by providing interactions that support the process of forging visualizations one mark at a time before combining them to form a visualization, NapkinVis can act as an effective pen-centric front-end to the Protovis toolkit as a result. The benefit of this is that the user can realize some of the flexibility of protovis without needing to write a single line of code.

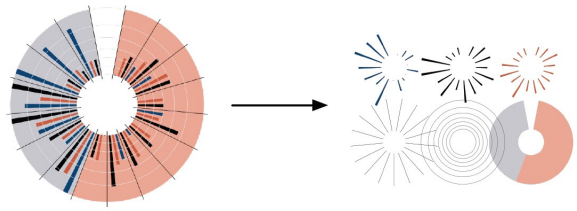

Figure 2. From Bostock and Heer (2009), visualizations can be thought of as composing of separable marks

The basic work-flow of NapkinVis begins with the user drawing ink strokes on a virtual canvas as they would when doodling or making quick sketches on a paper napkin. When the user draws a closed stroke, it transforms into a proto-object that can be further transformed into a variety of other actors on the canvas. These actors allow the user to open files containing data to visualize, create single-mark visualizations, or create compound visualizations which allows marks to be combined into new visualizations. A concrete example of this last point would be how a user can take area charts representing sales numbers of two employees, stack them on top of one another, and end up with a stacked area chart where the height represents the sum total of the sales of the two employees.

3.2 Elements of NapkinVis

There are several elements that comprise NapkinVis and add various functionality to the system. Each of these components allow the user to fluidly interact with NapkinVis in order to either quickly view simple visualizations, or to author more complex visualizations. A quick summary of these elements is seen in figure 3.

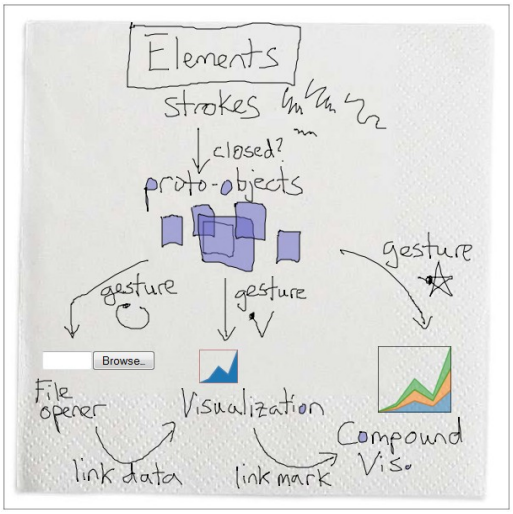

Figure 3. Elements of NapkinVis, demonstrated in the NapkinVis interface

The following sections will elaborate more on each element found in NapkinVis.

3.2.1 Main Canvas

To emulate the feel of sketching on a napkin-sized interface, the canvas is a fixed-size area that can be sketched on. Sketching is done by holding down the right-click button and dragging with a mouse, or simply drawing with a stylus pen. The back end renders several layers of canvas and visual actors to optimize for speed during various required re-draws, however to the user this appears as a single canvas that the user can simply doodle on.

3.2.2 Strokes

To simulate sketching on a napkin-sized canvas, strokes provide a means for general inking on the canvas. A stroke is a single set of points drawn by the user on the canvas between a mouse-down and a mouse-up event. This translates to a stroke simply being a single line or curve the user has drawn. Depending on context, strokes can be interpreted as simple inking, proto-objects (objects that can be transformed into other things such as visualizations), gestures, links, and so forth.

3.2.3 Proto-Objects

When the user draws a closed stroke, the stroke becomes transformed into a proto-object. Proto-objects can then be transformed into other elements by drawing a gesture inside of the proto-object. The proto-object provides a means of specifying the location and size of an object that a user might want. For instance, if the user wanted to create a visualization at location (25, 20) with a size of 100x150, they could simply draw a box of that size

with the top-left corner at (25, 20) creating a new proto-object, then draw a gesture inside the proto-object to create a visualization.

3.2.4 Gestures

Basic gestures are interpreted from strokes by converting the stroke information into a string of directions (from a character set of 9 directions), and then using the Levenshtein distance to compare the stroke with predefined gestures. Because gestures have been well defined outside of the scope of this project, we won't go much further in to them.

In addition, as the gesture vocabulary can be customized to suit the user, the gestures we used will be omitted from this paper. Thus when it is mentioned that a gesture was used, you may assume this means we used a different gesture for different actions or actor types.

3.2.5 File Opener

A file opener is simply a tool that allows the user to specify the path of a file to open. This tool can be invoked via a gesture drawn in a proto-object. Once a file is chosen, this widget is transformed into a data actor which gives access to the data in the file.

In future versions the file opener will be changed from a button to a more sketch-based solution such as text recognition, in order to preserve the doodling 'feel' of sketching on a napkin.

3.2.6 Data

It is assumed for the scope of this project, that all data imported is tabular data with an evenly-spaced independent variable. What this means is that every column represents a different data dimension, and plotting the values on a line chart will not produce any undesirable artifacts (vs. attempting to plot a set of x and y values using only the y values and evenly spacing out x).

3.2.7 Data Actor

As seen in the very left of figure 1, the data actor provides the first real visualization of the data, and is created by opening a data file using the file opener. Often times, if the user is simply curious about what their data 'looks' like as part of their work-flow, this tool may be all the user will need. Each row represents a data dimension in the parsed file.

3.2.8 Single-Mark Visualization

These are visualizations that contain a single mark, such as a canonical line graph, bar chart, area graph and scatter plot. These can be invoked by drawing a gesture inside a proto-object. The position and size of the visualization are determined by the respective properties of the proto-object. To select a mark after the visualization has been invoked, an additional gesture can be drawn inside the visualization.

The single-mark visualization is initially populated with toy data so the user can determine the mark used by the visualization. To indicate this toy data, the initial border of this visualization is colored brown. When actual data is linked to the single-mark visualization the border changes to a light gray color.

To link data to the single-mark visualization, the user simply draws a stroke starting in the data actor to any point in the visualization itself. To specify which data dimension the user wants to plot, the starting point must lie within the mini visualization of that dimension in the data actor.

3.2.9 Compound Visualization

The compound visualization is a tool to compose visualizations consisting of more than one mark, such as stacked area graphs, stacked bar charts, and much more. In order to create a compound visualization the user would simply draw a gesture inside a proto-object. To then add a mark to the visualization, the user would simply draw a line from the single-mark visualization to the compound visualization. The direction of entry determines the direction the mark will be stacked on previously placed marks. The location of the stroke's end point within the compound visualization determines which previous mark the new mark will be stacked on.

3.3 Summary of Interaction

Each subsequent interaction take users through a more and more detailed look of their data. In the early stages, the user can simply see a quick, automatically generated visualization of their data. Often times this may be all the user needs. Following this, the single-mark visualization allows an even closer look at individual data dimensions, offering a variety of visual marks the user can use, and through parameter specification such as size (and in effect, aspect ratio) the user can see different aspects of the same data dimension. The final stage allows the user to view a dimension's relationship to data in other dimensions by composing visualizations containing more than one mark.

4 Implementation

NapkinVis was created entirely in Javascript, CSS, and HTML, and it was programmed in such a way that the entire web application will run properly as a local file. Rather than drawing visualizations from scratch, the Protovis toolkit was used to build and render visualizations. Simple optimizations in speed were implemented and this system was tested on a netbook running a relatively slow 1.6 GHz Intel Atom processor. Testing of functionality was done in version 3.5.5 of the Firefox web browser running under Ubuntu 9.10, Windows XP, and Windows Vista, however no formal benchmarking was performed yet as this project is still in the proof-of-concept phase. Basically, if interaction was fast enough to be able to sketch comfortably, development continued without any additional focus on performance optimizations.

4.1 External Ideas

There were several ideas used in NapkinVis inspired by previous works. This section will outline some of the ideas used.

4.1.1 Scented Widgets

In order to provide a quick overview of the data the user has imported, the idea of scented widgets [12] was adopted. Scented widgets are essentially standard widgets with additional information about some aspect of the underlying data superimposed or rendered nearby the widget. With this in mind, the data actor can then be thought of as a scented list of data dimensions.

4.1.2 Stroke and Object Promotion

The idea of creating proto-objects that can be promoted to objects with additional properties, such as a visualization, was actually an idea adopted from examples in video games. For instance, in the game Super Mario (www.nintendo.com), the main character can gain certain characteristics by interacting with certain objects. A mushroom may increase the character's size, or a flower might allow the character to throw fire. This behavior can be seen as an example of how to trigger class inheritance visually. In the case of this project, the class inheritance is triggered by gestures. This idea is utilized much further in my previous sketch-based work outside the scope of this class project.

4.1.3 Visualization as a Composite of Marks

Although several interaction types were experimented with, it quickly became clear that it is quite difficult to attempt to specify properties of different marks inside of a compound visualization. The idea to break down the visualization to individual marks, and specify properties of each mark individually was thus adopted from the Protovis paper. This has so far proven to be a scalable way of thinking about constructing visualizations. In Protovis, this is done via scripting in a declarative language. In NapkinVis, this is done visually, so it is quite transparent to the user what is going on.

Originally a drag-and-drop methodology was experimented with, however the disadvantages of having the individual marks disappear as they merge with the compound visualization became apparent. It is much more difficult to get an idea of what is going on when these marks disappear. In addition, the user loses the ability to further author attributes of the individual marks.

4.1.4 Levenshtein Distance for Gesture Recognition

The algorithm for gesture recognition was adopted from the well-known method of comparing differences between character strings. This work was simply ported from commonly found pseudo-code into a Javascript implementation.

4.2 External Libraries

In addition to external ideas adopted into this project, significant work figuring out how to render the individual visualizations was saved by utilizing the Protovis toolkit.

4.2.1 Protovis Toolkit

Each visualization component essentially generated an HTML container (known as a “div” element) to render each Protovis visualization in, and then dynamically generated code to be executed by Protovis in order to create the actual visualization.

4.3 Implemented Components

There were many components implemented in order to bring NapkinVis to its current state. The following sections list some of the more crucial key components and considerations that went into them.

4.3.1 Drawing Canvas

The drawing canvas is created in several layers. The two primary layers contain all drawn strokes. The top-most layer contains only the current stroke being drawn. This is done so that feedback from drawing can be seen immediately. When the current stroke is rendered, it is placed in the further canvas. The other canvas is only updated when it is changed, unlike the top-most which is updated at the insertion of new points. The two primary layers are implemented using the 'canvas' tag, which currently isn't supported in all web browsers. To improve compatibility, SVG will be used in the future.

Visualizations are rendered differently. Containers specified by the 'div' tag are created for each visualization, and protovis is instructed to draw its visualization in its own specially created div tag. This allows for the localization of each visualization in the web page, and individual updates of visualizations without the need to re-draw everything else that is rendered.

These considerations allow the interactions and visualizations to be drawn in real-time, even on a slow netbook.

4.3.2 Gestures

In order to implement gestures, strokes were converted into a string of directions and these were then compared to strings representing the gesture templates using the Levenshtein distance. This distance does a rather good job at differentiating fairly different strokes from one another, however there doesn't appear to be a good 'fitness' parameter to measure whether a stroke should be accepted or rejected to begin with. In future versions, normalized cross correlation between normalized strokes and templates could provide a metric to help with this issue.

4.3.3 File Opener and File Parser

In order to open files, a workaround was needed as Javascript does not seem to provide a direct way to access local files. Basically, in order to open text files the file chooser embedded in the canvas determines the path of the desired file. This file is then loaded into a browser frame. A callback then executes a parser function on the contents of the frame once the text file has been loaded. Hopefully future versions can provide some sort of more direct method of accessing data.

Interestingly, it has been suggested that rather than try to open a file, the user is simply given a text field to copy and paste data in, which ends up being much simpler to parse using Javascript. This could potentially be an option in future iterations of this program once the constraint of 'paper napkin' proportions is relaxed a bit.

Once the data is loaded and parsed, the data is separated into columns and then visualized to the user.

4.3.4 Visualization for data actor

The visualization for the data actor is a simple one where boxes with text and a bar chart represent the titles and columns of data in the original spreadsheet. This visualization was programmed directly using Protovis. Data is scaled in such a way that the largest value of the data is 1, and it is shifted in such a way that the smallest point is 0. Although this will misrepresent data, for this prototype it appeared to be the best way to scale the data in order to maximize the small amount of space available to render the visualizations. In future versions a better scaling scheme will have to be employed so the user does not get a false sense of the qualitative aspects of their data.

4.3.5 Canonical Visualizations for Marks

In order to display the data encoded by the single-mark visualizations, a minimalistic visualization was coded for each of the marks. This was rendered using Protovis. The simple style was chosen to convey the feeling that these visualizations were still components that could be plugged into something. However, they still contained enough information that the user could make qualitative judgements on the data by using these visualizations, such as where minimums, maximums, different types of slopes, etc., occur within the data. In future implementations, other aspects such as detail on demand should be included to provide the user with a richer experience at this stage of the visualization process. Similarly to the data actor's visualization, the displayed data was scaled and sometimes translated in such a way that it maximized the utility of inking that could be done in the small amount of space provided to visualize.

4.3.6 Hit Detection

Although each Protovis component provides functions to handle callbacks for things like clicking and mouse movements, it was found to be much easier to simply keep track of the bounding rectangles of everything that you could possibly want to interact with, and simply check collision at every mouse movement. When an object was 'hit', it would be kept track of as being in focus, and only collisions with the item in focus was checked for. When the

item was no longer in focus, every possible item that could be interacted with was checked again. This helped keep processing hit detection relatively quick, and it allowed a much simpler way to access the items on the canvas that required specific zones to be selected later on in the development cycle.

It should be noted that this method seems to be similar to what other windowing systems do behind the scenes, so although it's important, it may not be novel.

4.3.7 Compound Visualization

A compound visualization allows users to combine marks together in order to create a wide variety of visualizations. In Protovis, a scene graph is essentially constructed where marks can inherit properties from other marks, and marks can be added to anchors of other marks. Complete detail can be found in the Protovis paper [5]. The key point is that this relationship from mark to mark in Protovis is utilized in NapkinVis. For every mark put into the compound visualization, the data linked to the mark, the type of mark, and a reference to a parent mark's anchor is stored in an array, essentially abstracting the scene graph into arrays of nodes and links. This allows the user to reference any mark stored in the scene graph with ease, as an array is one-dimensional in structure.

In order to specify an existing mark, the compound visualization is divided into equal segments, each representing a stored mark. Selecting a mark is as simple as dragging the mouse to a particular coordinate position within the compound visualization. For instance, in a vertically-oriented visualization like a stacked bar chart, if there were 2 bar marks in the chart (such as “sales of James and sales of Janet for every day in the month of January”), the compound visualization would be partitioned into 3 equal segments: two for the marks, and one for the root panel. The user can select a mark (visualized by dots placed on the mark's specified anchor) by moving the mouse to the first third of the visualization, the second third, or the final third. The type of anchor is selected by the direction of entry when drawing a stroke. So if the user would like to make a stacked bar chart, the user would draw a line between the bar visualization and the portion of the graph that specifies the mark the user would like to stack the new bar on top of. If the user instead, wanted to obtain a layered bar chart, the user would place all bar marks on the root panel.

To specify vertically oriented anchors on marks, the direction of entry would be from the top-down, or from the bottom-up. Top-down specified marks stack subsequent marks on top of other marks. Bottom-up places the marks underneath the previous marks. To specify a horizontal orientation, the user would enter left-to-right, or right-to-left. Currently this interaction works well for area charts as stacking is intuitive with these marks, however some thought will need to be put into other mark types. For instance, when working with bar charts and entering from the left, would we want to specify a horizontal bar chart, or do we want a vertical bar chart with the marks of elements staggered beside one another?

5 Results

This section will go through a sample scenario of using NapkinVis to create a visualization, and will show some sample visualizations that can also be created with this tool.

5.1 Towards Single-Mark Visualizations

The user begins by drawing a closed stroke in order to create a proto-object. This proto-object is then converted into a file loader by drawing a gesture inside of the proto-object as seen in figure 4.

Figure 4. Specifying a file opener from a proto-object by drawing a gesture within the proto-object

The user then uses the file opener to select a data file to open. When this is completed the application loads the file, parses the text, splits the data into columns (as it assumes that data dimensions are stored in columns), and automatically generates labels for the user. A visual representation of the data contained in the file is then produced for the user in the form of a data actor. This data actor shows the data for each dimension in different rows, with values displayed as normalized bar charts to maximize ink used to display the data.

The user may then wish to further visualize the data. In order to specify a single-mark visualization, the user can draw additional proto-objects, and then convert them into visualizations using a gesture. When the visualization is created, the user can then pick which mark will be used to represent the data. Before data is linked to the visualization, toy data is represented, and the border of the visualization is colored brown to represent this fact. When data is linked to the single-mark visualization, this border color changes to a light gray.

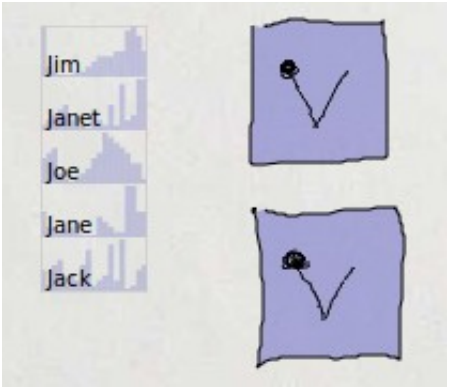

Figure 5. Left: When data files are opened, individual dimensions are visualized with a scented list widget. Right: Specifying single-mark visualizations from a proto-object by drawing a gesture within the proto-object.

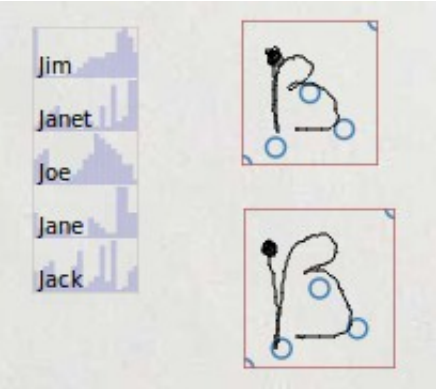

Figure 6. By drawing a gesture inside of a single-mark visualization, the mark type can be chosen

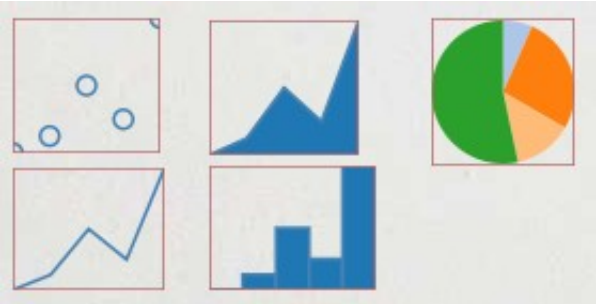

Figure 7. Several types of marks can be selected: dots, areas, wedges, lines, and bars

There are several marks that the user can chose from in order to visualize the data in the single-mark visualization. By drawing different gestures, the user can choose between encoding data with dots in a scatter-plot, using areas in an area graph, utilizing wedges in a pie-chart, having a line graph represent the data, or using bars to draw data in a bar chart.

5.2 Compound Visualizations: Creating an Example Stacked-Bar Chart

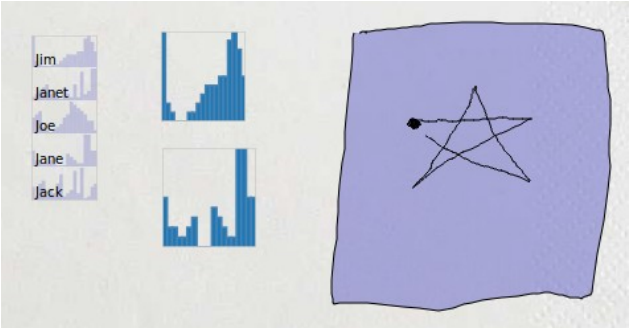

Figure 8. From a proto-object, a compound visualization can be specified

In addition to viewing the characteristics of a single dimension of data, the user may then want to compare dimensions against one another. By utilizing compound visualizations the user can then compose visualizations consisting of multiple marks, in a wide variety of combinations.

To create a compound visualization, the user first draws a proto-object, then specifies a compound visualization by drawing a gesture within the proto-object.

The newly created compound visualization only consists of a panel. This panel has a black outline in order to differentiate it from other components that NapkinVis supports. A panel contains anchors for the top, bottom, left, right, and center. In order to simplify things, the system currently supports the use of all but the center anchor for stackable marks such as bars and areas. Wedge-based visualizations are placed on the center anchor to begin with.

To create a stacked bar chart, the user first creates single-mark visualizations consisting of bars. From these, the user can then place them into the compound visualization.

The direction of entry is important in selecting anchors. In this example, a stacked bar chart oriented from the bottom-upwards is desired, so the user will link all marks in a top-down direction. This is shown in figure 9.

It should be noted that every new mark added to the compound visualization is encoded in an automatically chosen color from a palette based on the category 10 palette in Protovis.

5.3 Compound Visualizations: Extending the Stacked Bar Chart

In figure 10, the user decides they also want to add bars to the top of the diagram oriented downwards, so a link is created by entering the compound graph in an upward direction. The link end position is chosen in such a way that the root panel is selected, indicated by the brown circle on the top of the compound diagram.

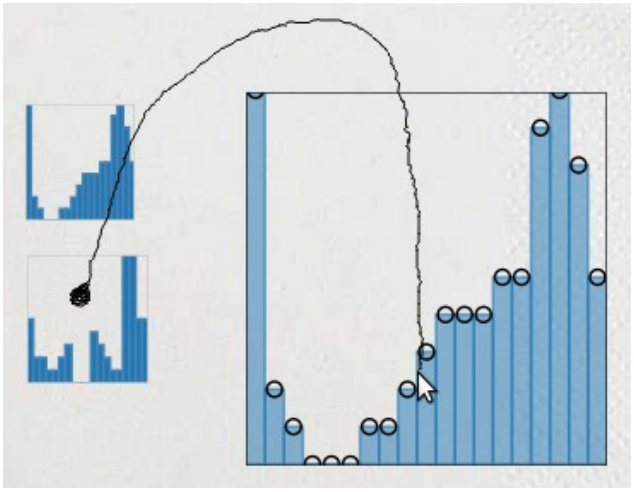

Figure 9. By drawing lines from the single-mark visualization to the compound visualization, marks can be placed on existing anchors. When approaching from the top, marks are stacked on top of other marks

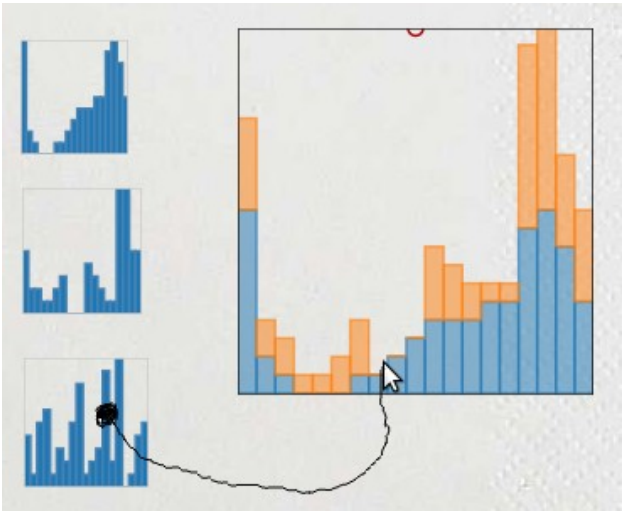

Figure 10. When linking from the bottom, marks are placed under already existing anchors

The result of placing the bars on top, oriented downwards can be seen in figure 11. Because this project aims to give the user a flexibility that approaches that of a toolkit, it quickly becomes apparent that some design choices allowed by the system may not be particularly wise visual encodings. However, we will continue with one more example to hint at the flexibility of expression in this system.

Different marks can also be combined in a compound visualization. In figure 11, the user decides to create an area-based mark, then add that to the compound visualization. The result is then seen in figure 12.

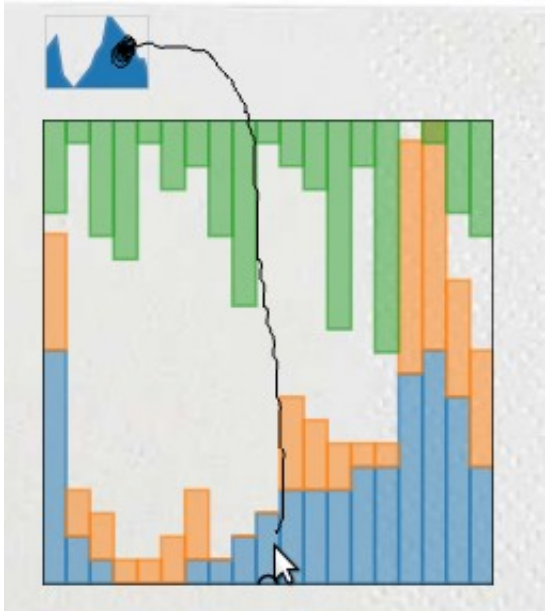

Figure 11. Various types of marks can be combined. In this figure an area mark is being applied to a visualization composed of only bars to this point

Because NapkinVis is can be used as a prototyping tool, marks are intentionally rendered transparent in order to show to the user that occlusion has occurred rather than stacking. In figure 12, it is clear that the area mark occludes several bar marks that were placed before it, however we can also see that the bar marks are stacked on top of one another. In the top right corner of figure 11 however, we can see that some occlusion does indeed occur between one of the green bars situated at the top of the panel, and one of the orange bars situated in the stacked bar chart on the bottom.

Following the authoring of the visualization, the user can then re-use the visualization should they decide to do so by changing the data being bound to the individual marks, or by changing the type of marks themselves.

By using a similar process, it becomes clear that there are many possible combinations in which one can arrange marks in a compound visualization. In the next section, a few more simple examples will be demonstrated.

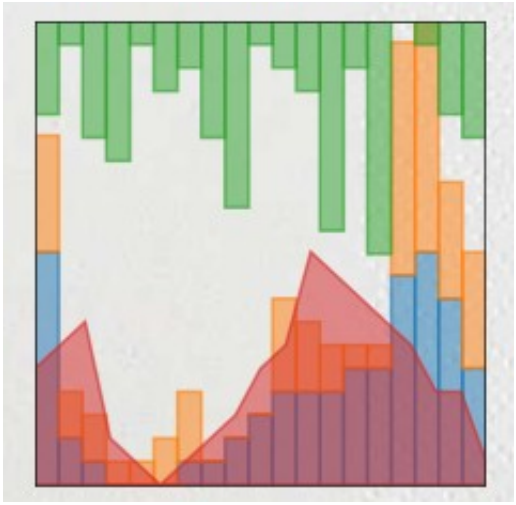

Figure 12. A compound visualization composed of bar and area marks

5.4 Some Sample Visualizations

Following are a few samples of visualizations that can be created from the interactions provided by NapkinVis. These are only a few samples out of the wide combination of possibilities that are available to the user.

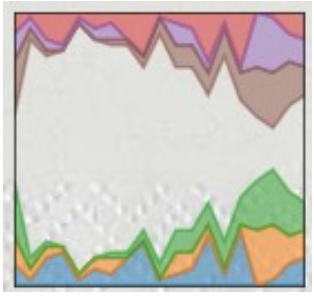

Figure 13. Area marks are stacked on top of one another and below one another to produce a mirrored effect in this stacked area graph

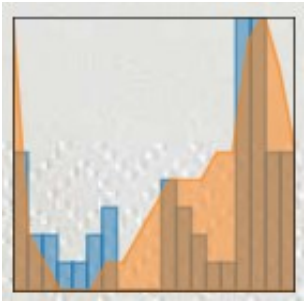

Figure 14. In this visualization a bar chart is combined with an area graph

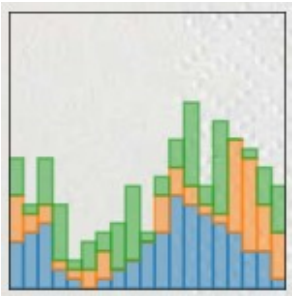

Figure 15. A stacked bar chart is created by placing three different bar visualizations on top of one another

6 Discussion and Future Work

6.1.1 Strengths

Even with trivial case of empty visualizations, one can quickly see that specifying the width, height, x-coordinate, and y-coordinate for 10 visualizations using Protovis alone would take minutes as the programmer would need to not only measure the locations desired on their canvas (a web page, for instance), and then type in the parameters for 10 different visualizations (assigning each to a different variable for future use, or dealing with many lines of code), but this would require some trial and error as visual feedback is only given when the programmer refreshes the page. With NapkinVis, visual feedback of drawing rectangles is given for free, and rather than caring about exact locations, the user can draw visualizations at the locations they feel like placing them for the purposes of quickly looking at some data. The benefits only multiply when we start moving from empty visualizations, to ones that actually show things!

NapkinVis allows the production of a surprisingly wide variety of visualizations, with a great number of which being useful. For instance, a simple stock chart showing highs and lows in a trading day about an average price can be generated by combining a line graph representing the average price, with two bar charts that represent the positive and negative deviations of the trading day. Although the range of combinations has not yet been explored, the preliminary results suggest that this tool is taking a step in the right direction.

The fact that the program runs fluidly on a netbook in an interpreted language is also encouraging as it seems to suggest that it may be possible to port the software to other low powered devices such as smart phones, which could open up an interesting alternative way to analyze data on the go.

6.1.2 Weaknesses

There are a number of missing basic features in order to keep the time scale realistic for a class project. For instance, basic features such as “undo”, “move”, “resize”, “copy”, “paste”, and many other functionalities are missing as they were not necessary in creating this proof-of-concept tool, however from a usability standpoint these would be critical features to include in later versions.

NapkinVis also has a few weaknesses however which are inherent to its current design.

With regards to the types of compound visualizations that can be made, the number of marks that can be combined comfortably into a visualization is limited to the number of vertical or horizontal pixels in that visualization. The practical resolution is far less. This may not be a problem with most visualizations, however for data with many dimensions, a new interaction scheme will most certainly need to be devised. Perhaps a large area with “last added mark”, or standard layouts can be employed to help tackle this issue.

This tool also lacks the precision of control Protovis as by not exposing many mark attributes to the user. While restricting the user's freedom with many automatic decisions is practical for the everyday user who may simply want a quick look at data before moving on, this will definitely be an unnecessary wall for power users and users wishing to author novel visualizations using this system.

Free-form placement of marks is also missing, and may be useful for further customization of compound visualizations.

NapkinVis also currently lacks basic visualization components such as axes, tick marks, labels, etc. In order for this system to extend its use beyond visualizations suited for very rough, qualitative analysis of data, these features will need to be included.

6.1.3 Future Work

When designing the interaction for NapkinVis, it became clear that there are some parameters that are inherently ambiguous to define. There are also more parameters than one can comfortably map to sketch-based interactions such as gestures without the system becoming too difficult to use. It has been suggested to offload some of these parameters from memory to interaction methods involving search, such as visual palettes, menus, etc. As the benefits of being able to specify these parameters exist, these avenues of design will certainly be explored in the future.

In addition, when composing single-mark visualizations, it became apparent that for different types of marks, different parameters may be more desirable to specify than in others. For instance, in bar charts, one may want to specify the thickness and offset of bars, whereas in wedge-based visualizations, being able to specify the outer radius and inner radius would be more beneficial. In future versions, these aspects will need to be considered and good default behaviors will need to be chosen.

One of the most powerful aspects of Protovis, the ability to pass anonymous functions as parameters is completely missing from NapkinVis. It would be very interesting to see if it's possible to include such a feature with the limited set of interactions that a visual system can provide versus a scripting language.

Other basic functionality which is important to any visual system are currently being worked on, too. For instance, interactions for zooming and filtering are being experimented with. It is unfortunate that the ability to parse text was not completed for this project, as it could have opened up an entirely new avenue to be explored. This is yet another open problem for this system.

NapkinVis has taken its first steps towards being able to flexibly create quick visualizations, as well as simple visualizations combining many marks. It still remains an open problem as to what kinds of attributes will need to be defined in order to give this tool the same flexibility as a visualization toolkit. Imagine being able to create novel visualizations on-the-fly without having to write a single line of code. I remain hopeful that this is something that can indeed be achieved one day.

7 Conclusions

In this paper we demonstrated the creation of a system to quickly author visualizations using pen-based interactions. This system, NapkinVis, used a work-flow that required the user to first author individual marks, and then combine them in a compound visualization. It was also able to utilize the concept of 'anchors' from Protovis to help guide the placement of marks. As a result, this system was able to quickly generate basic visualizations as well as provide functionality to compose a wide variety of new visualizations, in under a minute each in most cases.

Acknowledgements

I would like to thank Tamara Munzner for her helpful suggestions throughout the various stages of this project.

References

[1] M. Chuah et al. Sketching, Searching, and Customizing Visualizations: a Content-based Approach to Design Retrieval. *Intelligent Multimedia Information Retrieval*, 83-111, 1997.

[2] eToys, www.squeakland.org (accessed on October 28, 2009)

[3] E. Ernerfeldt. Master's Thesis: Phun. *Umea University*, 2008.

[4] Stolte, C. et al. Polaris: A System for Query, Analysis, and Visualization of Multidimensional Relational Databases. *IEEE Transactions on Visualization and Computer Graphics*, 8:1, January-March 2002

[5] M. Bostock and J. Heer, Protovis: A Graphical Toolkit for Visualization. October 2009

[6] J. Heer et al. Prefuse: a toolkit for interactive information visualization. *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2005.

[7] Flare, flare.prefuse.org (accessed on October 28, 2009)

[8] J. Fekete. The InfoVis Toolkit. *IEEE Symposium on Information Visualization*, 2004

[9] W. Schroeder et al. The Visualization Toolkit: An Object-Oriented Approach To 3D Graphics, *Kitware*, 2002

[10] JavaScript Infovis Toolkit, thejit.org (accessed on October 28, 2009)

[11] Processing, processing.org (accessed on October 28, 2009)

[12] Willett et al. Scented Widgets: Improving Navigation Cues with Embedded Visualizations. 2007.