

**Improvisational Interfaces for Visualization Construction
and Scalar Function Sketching**

by

William O. Chao

B.Sc., The University of British Columbia, 2006

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES
(Computer Science)

The University Of British Columbia
(Vancouver)

April 2012

© William O. Chao, 2012

Abstract

Presentations are an important aspect of daily communication in most organizations. As sketch, and gesture-capable interfaces such as tablets and smart boards become increasingly common, they open up new possibilities for interacting with presentations. This thesis explores two new interface prototypes to improve upon otherwise tedious presentation needs such as demonstrating models based on scalar functions, and visualization of data. We combine a spreadsheet style interface with sketching of scalar mathematical functions to develop and demonstrate intuitive mathematical models without the need of coding or complex equations. We also explore sketch and gesture based creation of data visualizations.

Table of Contents

- Abstract ii**
- Table of Contents iii**
- List of Tables v**
- List of Figures vi**
- List of Acronyms x**
- Preface xi**
- Acknowledgments xii**

- 1 Introduction 1**
 - 1.1 Motivation 1
 - 1.2 A Function Sketching Example 5
 - 1.3 A Visualization Sketching Example 6
 - 1.4 Contributions 6
 - 1.5 Organization 7

- 2 Related Work 10**
 - 2.1 Presentation Tools 10
 - 2.2 Sketch Authoring Systems 12
 - 2.3 Visual Programming Languages 16
 - 2.4 Spreadsheets 18
 - 2.5 Visualization Authoring Tools 19

3	Function Sketching	23
3.1	Work-Flow	27
3.2	Implementation	30
3.3	Results	32
3.4	Discussion	36
4	Visualization Sketching	38
4.1	Work-Flow	38
4.2	Implementation	40
4.3	Results	45
4.4	Discussion	47
5	Conclusions	50
5.1	Conclusion	50
5.2	Discussion and Future Work	50
	Bibliography	53

List of Tables

Table 1.1	A comparison of various presentation media	4
Table 3.1	A comparison between functions, their plots, and approximate sketches of them using our interface. Note that the third and sixth function sketches are computed from the two above them, and are not manually drawn.	26

List of Figures

Figure 1.1 An example of function sketching. The three steps of function sketching are setting up the scalar function models in the sketch spreadsheet, setting up the diagram, and interacting with it via sketching new scalar functions. 8

Figure 1.2 An example of visualization sketching. The three steps of function sketching are drawing the canvas actors, selecting the visual encoding of marks, and combining the marks in a compound visualization. 9

Figure 2.1 Part of the user interface seen in today’s presentation software. This class of software allows one to construct and organize virtual slides in a sequential order, and transition between them. 11

Figure 2.2 Zoomable user interfaces (ZUI) allow a more flexible placement of slide show elements. This enhances the presentation overall by introducing a spatial memory element. (Reproduced with permission from: [21]). 13

Figure 2.3 In Motion Doodles, features of a sketch are used to set parameters for classes of recognized bipedal characters, as well as their fully animated motions. (Reproduced with permission from: [55]) 14

Figure 2.4 In SILK, sketching and storyboarding combined with brushing-and-linking are used to rapidly prototype a user interface. (Reproduced with permission from: [29]) 15

Figure 2.5	In MathPad ² , mathematical models can be sketched and used to drive animations and other exploratory visualizations. (Reproduced with permission from: [30])	17
Figure 2.6	Today’s spreadsheets contain rows and columns of cells that can contain values and equations.	18
Figure 2.7	Spreadsheets have been augmented to include artifacts in their cells, such as full visualizations. (Reproduced with permission from: [14])	19
Figure 2.8	Protovis is a declarative Javascript toolkit that regards visualizations as a combined collection of marks, each with properties such as position and size specified by data. (Reproduced with permission from: [7])	20
Figure 3.1	A sample of white-board drawings one might find in classes or meetings that include plots of scalar functions. The plots are used to qualitatively demonstrate concepts, and multiple plots tend to have more than one plot that are connected by some functional relationship.	24
Figure 3.2	A cell actor is used to visually represent a function.	27
Figure 3.3	The spreadsheet mode arranges cell actors in a spreadsheet layout.	28
Figure 3.4	Functions are drawn in cells ss.0.0 and ss.0.1.	29
Figure 3.5	Cell ss.0.2 is set to be the sum of cells ss.0.0 and ss.0.1.	29
Figure 3.6	The function in ss.0.1 is redrawn and the result in ss.0.2 updates accordingly.	30
Figure 3.7	Any cell involved in an equation can be shown in the presentation modes and can be interacted with.	30
Figure 3.8	A simple addition of functions in spreadsheet mode.	33
Figure 3.9	A simple addition of functions in canvas mode.	33
Figure 3.10	A demonstration in canvas mode of normalized cross-correlation for signal detection.	34

Figure 3.11	A demonstration in canvas mode showing amplitude modulation. The center and right image shows the result when a new signal or carrier are drawn respectively.	34
Figure 3.12	A function sketch demonstration showing amplitude modulation in spreadsheet mode.	35
Figure 3.13	A demonstration in canvas mode showing that if you apply a kernel with normalized-cross-correlation repeatedly, you will eventually end up with a Gaussian. The reverse-polish-notation equations are shown in this figure. This demo is often shown in introductory computer vision classes.	35
Figure 3.14	A comparison between acceleration, velocity, and position. This demonstration is often used to illustrate integration. The equations used to calculate velocity and position for this function sketch are shown.	36
Figure 4.1	Drawing strokes and proto-actors.	39
Figure 4.2	To import data a proto-actor is converted into a data actor which is then used to open a spreadsheet. The spreadsheet data is displayed as a series of bar charts.	39
Figure 4.3	A gesture converts a proto-object into a visualization, and another gesture picks the visual mark.	40
Figure 4.4	The user can pick from several marks to visually encode information.	40
Figure 4.5	The information can be linked to a visual mark by brushing and linking, in other words drawing a stroke from the information to the visual mark.	40
Figure 4.6	Marks are combined in a compound visualization.	41
Figure 4.7	Elements of the visualization sketching system, demonstrated in its own interface.	41
Figure 4.8	A sample napkin visualization charting one week of page views to a website. By stacking bars on each other versus stacking them on the bottom of the visualization we create two common types of bar-based visualizations.	46

Figure 4.9	A mirrored, stacked area chart, a bar chart superimposed on an area char, and a stacked area chart created by combining area marks or bar marks in a compound visualization.	47
Figure 4.10	An unconventional visualization shows a comparison of total fuel usage between two monitored valves at a factory. Valve bars (seen in green and orange) are stacked on top of a baseline bar chart (seen in blue).	48

List of Acronyms

GUI	Graphical User Interface
ODO	On-(Drag)-Off mouse interaction
RPN	Reverse Polish Notation
WIMP	Window, Icon, Menu, Pointing device
WYSIWYG	What You See Is What You Get
ZUI	Zoomable User Interface

Preface

A portion of this thesis was presented as a poster at the 2010 IEEE Information Visualization Conference, under the title: “Poster: rapid pen-centric authoring of improvisational visualizations with NapkinVis” [11]. Two-page poster abstracts were included in the IEEE VisWeek Conference Electronic Proceedings. All research work and programming was wholly performed by William O. Chao, and the authoring of the aforementioned submission was reviewed and supervised by Dr. Michiel van de Panne and Dr. Tamara Munzner. Details of this work can be found in Chapter 4 of this thesis.

Acknowledgments

I first and foremost want to thank my supervisor, Dr. Michiel van de Panne, for his support and advice all throughout my thesis research. Discussions with him were always exciting, motivating, and informative, and I am happy to say that they got me through some of the toughest parts of this project. I would also like to thank Dr. Tamara Munzner for her guidance and for her very helpful suggestions for revisions throughout various points of writing this document. This thesis would not be what it is today without her. Another person I am grateful to is Dr. Ron Rensink. My discussions with him early in my studies, and his support during that time helped shape the future direction of my research. Additionally, I would like to thank NSERC, MITACS, and the Department of Computer Science for their financial support. These acknowledgements would certainly not be complete without expressing my gratitude to my family for always being there. Finally, I would like to give my heartfelt thanks to Lydia Liao, as having her by my side always motivated me to keep moving forward, and still continues to do so.

Chapter 1

Introduction

1.1 Motivation

Let us begin with a story about Jack and Jill. Jack is a high school physics teacher in his mid 30's who works in the Burnaby area. He loves showing students diagrams of mathematical functions. When he was in school, his teacher drew a cannon ball and an arc trailing behind it, which was enough to make sense of its projectile motion. Ever since then he was hooked. These days when he is asked questions, he would race to the chalk-board to doodle some curves to better answer them.

Zooming across the lower mainland, we find Jill who is a young and successful project lead at a major accounting firm in Vancouver. During her presentations, she is also asked questions. Jill's presentations are often based on data her team has collected about sales and audits, so she often finds herself describing the numbers, or if lucky, displaying a chart she had made well in advance just in case she were asked.

Both Jack and Jill have similar goals: they both need to be able to communicate some conceptual, numbers-based model in an improvisational setting. In Jack's case, the model can be constructed to fit the task at hand, in other words it can be artificial. In Jill's case, the model is dictated by the data that exists.

Ten years ago the story would have ended here as most presentation media back then were more or less static. In other words, you could not interact much with them. However these days technology such as tablets, smart boards, and other

novel interfaces like laser pointer input open up new possibilities of interacting in these settings.

These personae bring me to this thesis. Here, you will be seeing two closely related, sketch-based interfaces: one for creating and interacting with mathematical models, and the other for creating visualizations.

Thirty million is a sizable number. It is nearly the population of Canada and it is larger than the population of many countries in the world. It is also the number of presentations made each day as estimated by Microsoft...in 2001 [45]. Presentations are likely even more pervasive today. Even if a fraction of this number represents the improvisations of the Jacks and Jills of the world, this is still an interesting area that we can improve upon.

Our motivations for this thesis are to help improve everyday improvisations such as those we have mentioned earlier. Moreover, we aim to support rapidly emerging technologies in this area, especially those that help people interactively construct and show mathematical functions, as well as data. These first two motivations work towards the overall aim of one day supporting a workspace-approach of working in which, rather than having tasks sand-boxed in separate applications, functionality is available seamlessly in the work-flow. Finally, our last motivation for this thesis is to simply see if the system could be made.

We can summarize the problem we are approaching in the following question: “In an improvisational setting, how do we support the rapid authoring of active diagrams that visually communicate information?”

This question leads to a few domain constraints. First, we need to support an interaction method that can be easily applied in our domain. For this, we chose to build upon on-(drag)-off (ODO) interaction; examples of such being sketching, mousing, single-finger gestures, and the like. As presenters, we are often limited in how we can interact with presentation visuals when performing in front of a live audience. At best, we are given an electronic smart board to work with. More typically we have a means to point at items such as a laser pointer. In the best situations, these familiar interactions can be adapted to simple stroke, or gesture-like interactions by various means [13, 43]. This makes ODO interactions adaptable to how we already perform presentations. Fortunately once established, an interface built on ODO interactions can then be supplemented with other types of interactions, such

as multi-touch gestures, in order to provide a richer experience.

This does not mean that we will be eliminating the keyboard altogether, as this mode of interaction is still useful, and is often available in a virtual form [51] in modern devices. However, because virtual keyboards can be slow and visually obstructive, we aim to limit their usage.

The next domain constraint imposed is that authoring of any visual artifacts, such as charts, needs to be fast. In our case, we aim for under a minute, ideally around thirty seconds. The visuals created do not necessarily need to be polished, but they do need to be able to illustrate the key ideas the presenter is conveying.

This naturally leads to the next constraint: the interaction with any created artifact needs to be even faster than authoring. In other words, any visuals created should be responsive enough to demonstrate in a live setting.

Finally, there is the constraint that the visible interface should be minimal. This is because graphical user interface elements can be visually distracting to audiences in the presentation and improvisation setting.

There are two kinds of presentation artifacts, such as slides, that are generally encountered: passive and active. Passive artifacts stay the same and are used in scripted situations, such as when marketing a product, giving a talk, and so forth. These are usually created in advance, never change, and often cannot be changed, examples being videos or images. These have the benefit of being predictable and consistent; however, there are times when their immutable nature is a disadvantage.

Whether it is teaching new concepts in classes or demonstrating a new business idea, the process still often remains the same: using a series of static visuals with a minimalist interface in order to avoid distractions and preserve the flow of information. Yet this flow is often interrupted when the presenter wants to demonstrate a dynamic concept. An example could be a science teacher demonstrating the addition of sound waves in class using slides and then having to switch over to drawing on a black board, or an analyst asked to demonstrate a projection of data using different parameters than the ones prepared in advance. Being able to facilitate improvisational examples can be useful in settings such as these, with the added advantage that anyone sharing the slides can improvise their own examples.

Active artifacts are used in situations that allow or require improvisation, such as providing examples when answering questions, exploration, and more. Exam-

Table 1.1: A comparison of various presentation media

Name	Interactivity	Ease of Construction	Programmability
Dry-Erase/Chalk Board	low	easy	none
Linear Slides	low	easy	none
Pre-Recorded Video	low	easy	none
Powerpoint	low	easy-moderate	low
Spreadsheets	low	easy-moderate	low-moderate
Sketch Interfaces	low-high	easy-moderate	moderate
Physics-Based Systems	low-high	moderate	moderate
Custom Web Applications	low-high	moderate-hard	moderate-high
Custom C++ Program	low-high	hard	high
Java Applet Demo	low-high	hard	high

ples of active artifacts can include demonstrating a live computer program prototype, using presentation software with a dynamic slide order, showing animations that respond to equations, or even freezing a banana in liquid nitrogen in front of a chemistry class. In this thesis, we will refer to any active visual artifact that are created or interacted with in our interface as ‘actors’.

Table 1.1 compares common categories of presentation media available today. Each of these vary in terms of interactivity, ease of construction, as well as programmability. In general, there is often a trade-off between how dynamic something is, and how difficult it is to construct. On one end for instance, preparing a set of static slides is relatively easy to make without needing technical knowledge, however static slides are generally not interactive. On the other end, interactivity can be achieved by including a Java applet in the presentation to aid in demonstration and the sky is the limit as to what can be constructed with this approach. However, this option is only available to those who have sufficient technical background.

By combining ideas from sketch interfaces and spreadsheets, this thesis aims to provide an increased level of interactivity without significantly increasing the difficulty of authoring active diagrams in the presentation domain. The next two sections present examples of accomplishing this with our system.

1.2 A Function Sketching Example

A good compromise between ease of construction and programmability can be found in the spreadsheet interface. Spreadsheets allow for the creation of mathematical models that can be easily updated. They are dynamic; they have a well established relationship between cells and equations; they do not require coding to produce interesting results; and they are fast when compared to some other methods of generating models, such as typing in commands in a declarative programming language.

The process of function sketching is summarized in Figure 1.1. Creating a model and interacting with it using sketch involves a few steps: the creation of a spreadsheet actor on the main canvas, sketching functions in them, and linking these actors with equations.

Interaction with this system begins on a blank canvas. When drawing on the canvas sketched marks appear either as ink that remains static, like ink on paper; or form interactive tools called widgets or actors. In function sketching the canvas actors used are spreadsheet cell actors, where scalar functions can be sketched in. These actors can be created on the canvas and linked to equations individually, or the spreadsheet mode can be used as a shortcut to access to many cell actors at once.

Just like a spreadsheet, mathematical models involving cells can be created by linking cells with equations. Also, similar to a spreadsheet, any cell update pushes changes to other cells that are involved in the model. The result is a system that lets you quickly construct scalar-function based models and alter their inputs. Practical examples where this is useful include demonstrating the concept of filters in computer vision classes, showing construction and destruction of waves in an introduction physics class, drawing different waves to show how they would affect their resulting Fourier transform, modelling factors that could affect sales data over time and trying out hypothetical situations to see what they would do, to name a few. Surprisingly, there are almost no tools to our knowledge that allow any form of interactive sketching of a scalar functions as a means of exploring models or processes.

1.3 A Visualization Sketching Example

In the past, the authoring of visualizations was only available to those with programming knowledge. Others were limited to using pre-programmed wizards and templates. The advent of declarative scripting languages for authoring visualizations such as Protovis [7], opened up the possibility to provide GUI tools to modify aspects of visualizations that were previously available to only programmers. The visualization sketching aspect of this thesis presents an example of a sketch-based implementation of such a tool. Figure 1.2 summarizes the process of sketch-based authoring of visualizations.

Sketching visualizations requires the creation and linking of various actors, each specifying a subset of parameters for the underlying scripting language that will render the final visualization. This rendering is possible because the language we use understands that visualizations are collections of drawn marks that are systematically altered, positioned, and sized based on rules that can be governed by a small set of parameters [7, 59].

In the example of Figure 1.2 a familiar stacked bar chart is created. However the method of authoring visualizations in this thesis is not limited to only familiar types of charts. A variety of visualizations, familiar and specialized, can be constructed, consisting of many types of marks, stacked and positioned in multiple ways, each being able to respond to changes in the dimension of information that they are modelling [57].

1.4 Contributions

The contributions of this thesis include: (1) a sketch spreadsheet to construct and alter functional models, (2) a proof-of-concept WYSIWYG system to author visualizations that supports on-(drag)-off interaction, and (3) a working implementation of both components comprising of over 25k lines of code.

We combine ideas from several well-known interfaces and apply them to two types of information-based active diagrams. Our first proposition—constructing scalar function models in spreadsheets—borrows ideas from sketch interfaces, visual programming languages, and spreadsheets. Specifically, although spreadsheets have already been augmented to include information besides scalar values in

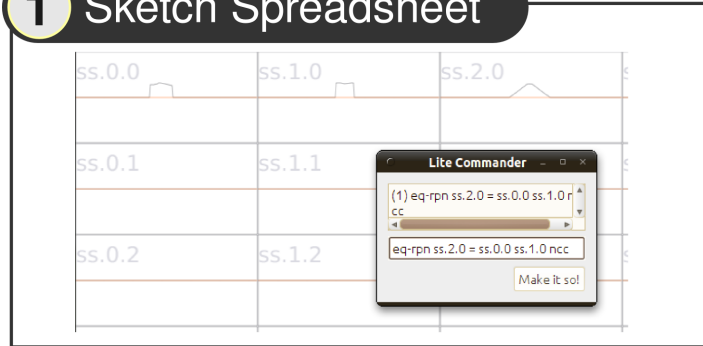
their cells, such as visualizations or even entire functions, we propose that sketching scalar functions in the cells can be a useful input method for construction, interaction, and improvisation. Our second proposition—constructing WYSIWYG visualizations with sketch—combines ideas from sketch-based interface construction, data-flow visual programming languages, and declarative visualization scripting languages, and demonstrates that it is possible to more finely construct visualizations without coding in contrast to previously available methods. These contributions are presented together in this thesis as they share a great deal in terms of interaction and applications.

1.5 Organization

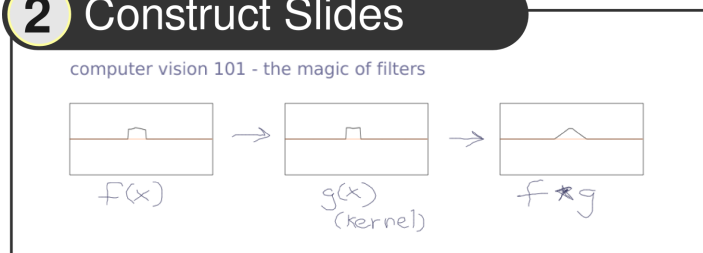
The remainder of this thesis is organized as follows. Chapter 2 covers related work. Our system for sketching functions is detailed in Chapter 3. Visualization sketching is described in Chapter 4. Finally the conclusions, limitations, and possible future directions of this thesis are summarized in Chapter 5.

Function Sketching

1 Sketch Spreadsheet



2 Construct Slides



3 Interact

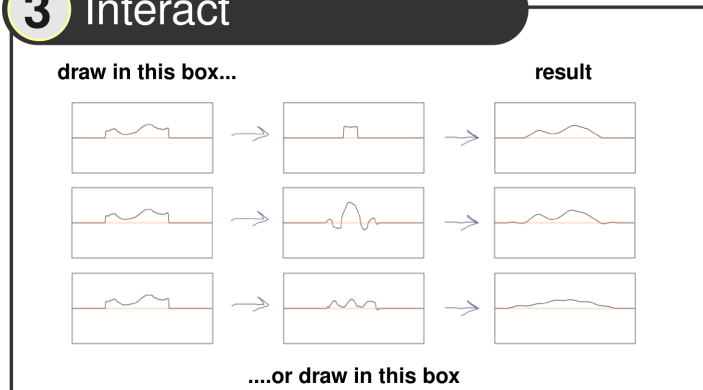


Figure 1.1: An example of function sketching. The three steps of function sketching are setting up the scalar function models in the sketch spreadsheet, setting up the diagram, and interacting with it via sketching new scalar functions.

Visualization Sketching

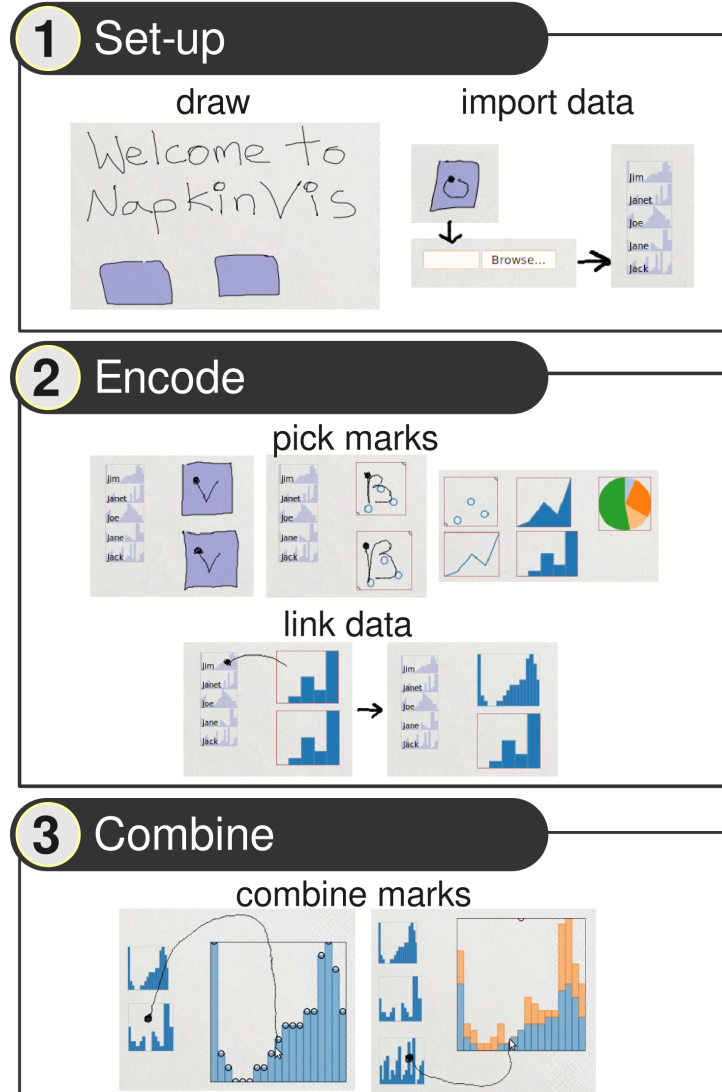


Figure 1.2: An example of visualization sketching. The three steps of function sketching are drawing the canvas actors, selecting the visual encoding of marks, and combining the marks in a compound visualization.

Chapter 2

Related Work

This thesis touches upon several areas of related work. Research in presentations reveal guidelines on how to implement systems for improvisational settings. Sketch authoring systems look at methods to create content from sketch input. Visual programming languages provide the means to encode program logic and information filters using a GUI, and often without requiring the use of a keyboard. Spreadsheets provide a visual layout for information and provide a means to quickly construct and interact with simple numerical models. Finally, visualization authoring tools provide the means to map information to visualizations.

In this chapter, we will briefly talk about some of the related areas of research, and expand on the ideas they contribute towards solving the problem statement mentioned in Chapter 1.

2.1 Presentation Tools

Presentation tools are designed to systematically show visual information to an audience and aid in the communication of ideas. They have come a long way from early technology such as manually drawn or printed visuals, to slide projection, to the digitized slide shows we commonly see today.

Currently, popular tools such as Microsoft PowerPoint or LibreOffice Impress are designed to present virtual slides one at a time. Figure 2.1 shows an example of part of an interface for this kind of software. The advantage of these programs

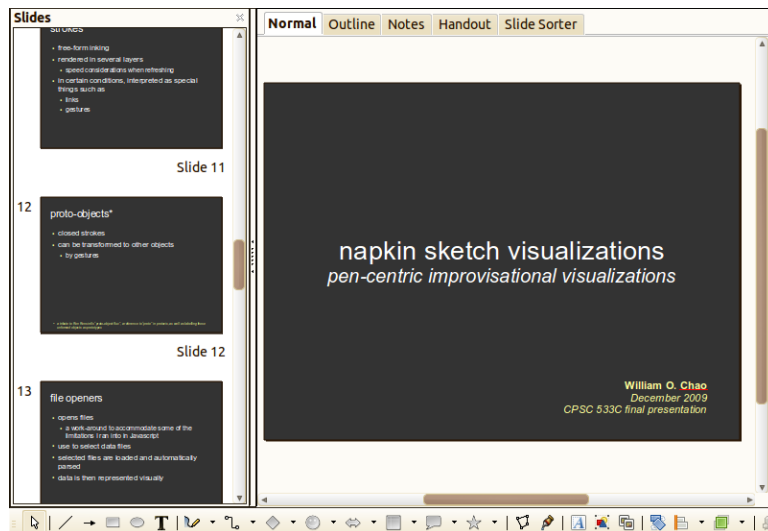


Figure 2.1: Part of the user interface seen in today’s presentation software. This class of software allows one to construct and organize virtual slides in a sequential order, and transition between them.

over traditional methods of presenting visuals, such as using chalk boards or slide projection, is that visual artifacts for presentations are easier to create and alter without having to re-print or re-create them entirely. However, the disadvantage of using these tools in an improvised setting is that the created visuals are often static, which makes them difficult to alter during a live presentation.

Additionally, the ability to interact with static visuals such as virtual slides is often limited. For instance, it would be a cumbersome task to demonstrate to a physics class how a ball would respond when applying different forces. A series of prepared animations would need to be made, or for those with programming knowledge, an application could be created and then embedded into virtual slides. Typically though, the range of interactions with presentation software is restricted to transitioning between slides, triggering an embedded multimedia object such as a video clip, or turning the screen on and off. These generally involve single button presses.

Pen and gesture interaction, although supported, is often used for making annotations or even for drawing static diagrams. This narrow spectrum of uses today

may be due to the lack of support for this mode of interaction in the presentation setting. Fortunately, the availability of pen-like interaction in presentations has been increasing, as evidenced by examples of works that have investigated additional means of supporting this style of input [13, 43]. Furthermore, in the marketplace today, software can be purchased that allows portable, gesture-capable devices such as tablets and smart phones to interact with presentations. Unlike single button clicks, which can typically be used to trigger events or toggle states, two-dimensional interactions, such as pen sketching and gestures open the door for a richer set of interactions with presentation media.

An area that has been explored is the rapid generation and exploration of visual artifacts. Some sketch-based contributions focused primarily on generating presentations [2, 34]. However these still maintained the traditional slide format, one that some have argued could use improvement [56]. Other works experimented with augmenting the traditional slide format to a more free-form canvas with enhancements such as incorporating zoomable user interfaces (ZUI) [6, 21]. An example of this interface can be seen in Figure 2.2. This idea not only changed the format of presentations, but enhanced them by adding spatial cues to information in the presentations. Other research, e.g., [63], added specialized, single-parameter animations for presentation purposes, and added an element of interactivity to them. The research also proposed additional guidelines for presentations and their animations such as building hierarchical slides, adding interactive controllers, doing one thing at a time, and making all movements meaningful, among other things. These changes and guidelines helped to bridge the gap between presentations and sketch authoring systems, albeit with specialized restrictions.

2.2 Sketch Authoring Systems

Sketching is often used in the ideation stage of design due to its speed and ease [27, 46, 47, 52]. The benefit of using digital sketches is that they can often be converted into rough, but functional content.

Sketch authoring systems often create content by mapping features of sketched, digitized ink to parameters used to create other types of digitized artifacts. Often these artifacts are time consuming to create using other methods, such as graphical

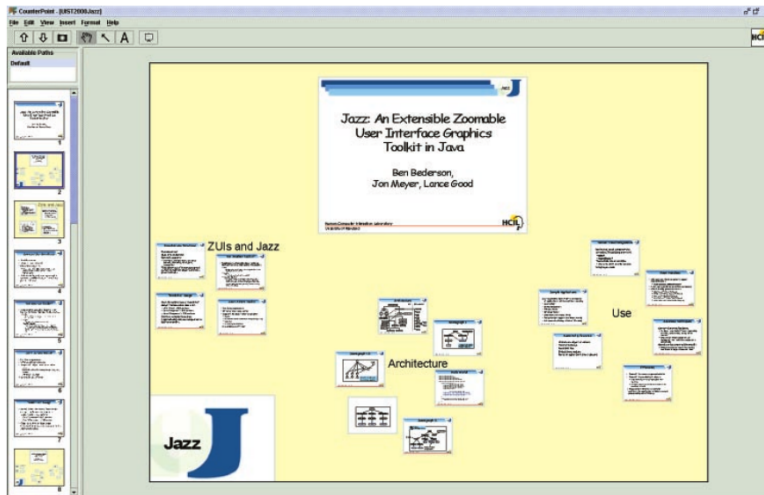


Figure 2.2: Zoomable user interfaces (ZUI) allow a more flexible placement of slide show elements. This enhances the presentation overall by introducing a spatial memory element. (Reproduced with permission from: [21]).

user interfaces [29], UML diagrams [12], three-dimensional models [3, 4, 61], or even full animations [55] as seen in Figure 2.3. This approach of mapping features of sketched curves to parameters provides a quick and natural means of specifying a set of parameters, which would otherwise need to be specified individually in sequence or inferred by other means. However, there is a trade-off between speed and precision when specifying parameters by features of sketches, due to the approximate nature of sketching compared to typing in a set of exact numbers. Therefore sketch is often seen in situations where the benefits of creating an approximately correct artifact outweighs the need for precision. There are several areas of research that build on top of sketch and sketch-like interactions for rapidly generating content.

Interaction by demonstration is one technique that has been explored in the past. Parameters are either inferred or specified by demonstrating desired behaviour. In some implementations this generally requires desired behaviour to be preprogrammed and mappable to demonstrated interactions [38, 39, 40], such as constructing a WIMP-based interface by demonstration. In others, demonstration is

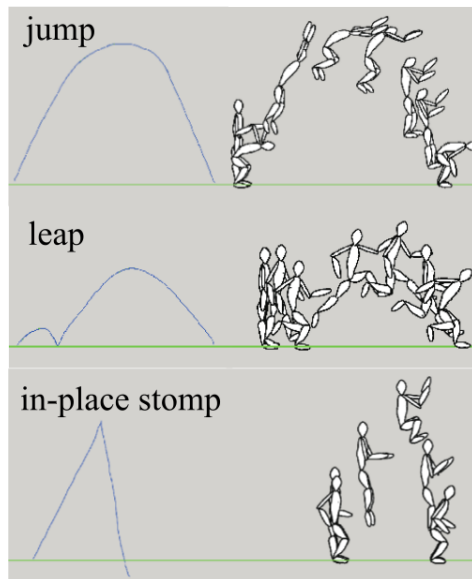


Figure 2.3: In Motion Doodles, features of a sketch are used to set parameters for classes of recognized bipedal characters, as well as their fully animated motions. (Reproduced with permission from: [55])

used to train the system how to respond to certain kinds of input [62], such as virtual characters responding to virtual puppets controlled in real-time. The strength of authoring by demonstration is that it is accessible to non-programmers and it provides a quick means to define classes of program behaviours that respond to specific triggers. It still however, remains an open problem to reliably have programs accurately guess the intentions of the user in every case. For instance, situations with an insufficient number of demonstrated examples could under-constrain the programming of the desired behaviour, leading to many possible behaviours that still match the examples given, in addition to opening the possibility that edge cases of desired behaviours may be entirely omitted.

Storyboarding is another approach to specifying desired behaviour [5, 29, 35] in created content. An example of this is seen in Figure 2.4. In this technique, several alternative configurations of the interface are sketched in separate frames. Individual components of the interface, along with parameters describing these components and their states are recognized and inferred from the sketch itself.

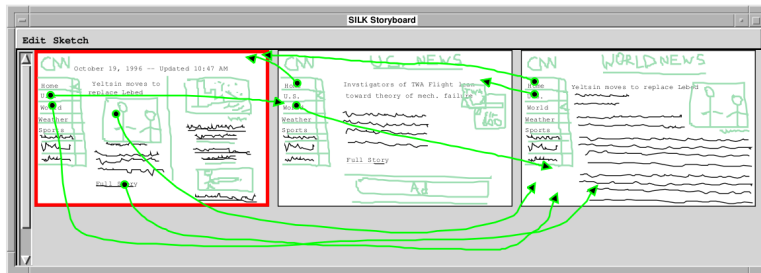


Figure 2.4: In SILK, sketching and storyboarding combined with brushing-and-linking are used to rapidly prototype a user interface. (Reproduced with permission from: [29])

The behaviour of the interface is specified by mapping sketched components to other frames of the storyboard. The end result behaves similarly to navigating web pages by clicking on links. An advantage of storyboarding is that it is accessible to non-programmers, as all that is required is sketching possible states of the system and drawing hyperlinks to transition between them. Additionally, storyboarding provides the assurance that the resulting behaviour of the drawn interface is predictable. However, storyboarding requires enough canvas space to sketch the possible frames of behaviour, and the resulting interface is constrained to the states that exist in the storyboard.

Kinetic sketching is a method for quickly creating content from sketches, primarily rough animations [17]. This is accomplished in part, by allowing sketched artifacts to be treated as objects that can be manipulated, and by recording the actions of those objects using direct manipulation, such as dragging a sketched boat to record its motion. In kinetic sketching, properties of an animation such as objects, trajectories, and transformations are specified during a recorded animation. This allows the specification of those parameters to occur at an appropriate point in the timeline of the animation. This process requires authoring to be specified iteratively, with each component of the animation defined one at a time, and then combined to produce the final product which can then be replayed when some event is triggered. The process of specifying parameters of an animation can be greatly sped up by pre-programming a library of motions and invoking them using parsed gestures embedded in a continuously drawn motion path, such as with

motion doodles [55]. Both of these methods provide a rapid and easy means for creating interesting animations. However, because the produced animations need to be defined previously, and because these animations play back the same way as they are defined, they are best suited for situations where novel behaviour or exploration of the behaviour of virtual models, such as in simple physics demonstrations, is not heavily required.

Another approach for specifying the behaviour of artifacts is to create mathematical models using sketch, and then use these models to govern the behaviour of the artifacts. One method that has been explored involves parsing equations written on a canvas, and then binding select variables of these equations to properties of drawn artifacts, such as the position of a sketched car, in order to produce animations [30, 31]. The resulting animations aid in visualizing the system of equations, as seen in Figure 2.5. Rather than recording animations that are immutable, the advantage of using a mathematical model is that the equations set up constraints that the components of the animations follow, allowing for variability in behaviour. Additionally, setting up a system of equations provides a quick means for specifying certain properties of animations such as the position of artifacts being animated. However, the approach of parsing written equations works best only if one is familiar with the mathematical equations needed to define the behaviour being illustrated.

In this thesis, we partially look at an approach for creating an underlying mathematical model for the canvas due to the potential utility afforded from having such a model. Instead of recording animations directly, using an underlying system of equations to constrain the behaviour of visual artifacts allows for ideas explored in visual programming languages to be useful for transforming and interacting with the underlying constraints themselves, and by extension the artifacts linked to them.

2.3 Visual Programming Languages

Visual programming languages provide the means to specify programs by primarily graphical, rather than textual means.

A type of visual programming language called data-flow visual programming

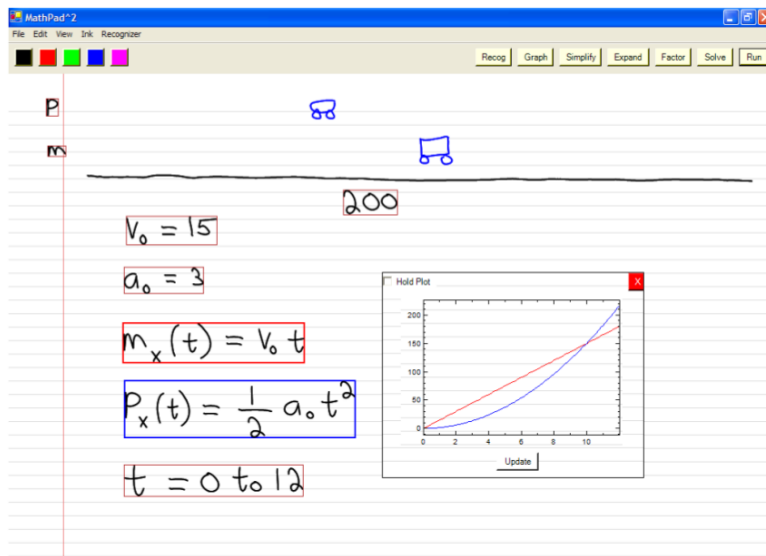


Figure 2.5: In MathPad², mathematical models can be sketched and used to drive animations and other exploratory visualizations. (Reproduced with permission from: [30])

languages acts on information as though it flows through a series of filters, with each filter transforming the information a particular way [26]. Traditionally these programming languages often visually appeared as node-link diagrams, with nodes representing either inputs, filters, or outputs and links specifying through which nodes transformed data should flow. These types of visual programming languages provide a means to both specify, as well as observe intermediate stages in a given series of transformations without the need to type out much code, if any at all.

Spreadsheets act similarly to data-flow visual programming languages by providing a visual means to show data and intermediate steps in a series of transformations. A difference is that they hide the filters that link these intermediate steps from view. A major advantage to note for spreadsheets is that you can easily specify the function to operate on a range of cells.

NOVA experiment descriptions												
Durations												
Exp #	Experiment	prep.	blank	display 1	blank	display 2	blank	display 3	blank	display 4	blank	
7	1000 8.20	150	10	20	10	20	10	20	10	200	10	
8	1000 8.30	150	10	20	10	30	10	20	10	200	10	
9	1001 9+cMask3	150	10	20	10	15	10	20	10	200	10	
10	1001 9+cMask4	150	10	20	10	15	10	20	10	200	10	
11	1002 10.1.20 master	150	10	20	10	20	10	20	10	200	10	
12	1002 10.2.20c master	150	5	20	10	20	10	20	10	200	10	
13	1003 11.2.15+	150	5	20	5	15	5	20	5	200	5	
14	1003 11.3.15+	150	5	20	5	15	5	20	5	200	5	
15	1004 12.20	150	5	20	5	20	5	20	5	200	5	
16	1004 12.30	150	5	20	5	30	5	20	5	200	5	
17	1005 12.20a	150	5	20	5	20	5	20	5	200	5	
18	1005 12.30a	150	5	20	5	30	5	20	5	200	5	
19	1006 12.20b	150	5	20	5	20	5	20	5	200	5	
20	1006 12.30b	150	5	20	5	30	5	20	5	200	5	
21	1007 12.20c	150	5	20	5	20	5	20	5	200	5	
22	1007 12.30c	150	5	20	5	30	5	20	5	200	5	
23	1008 14-3c	150	5	25	10	30	10	25	10	200	5	
24	1008 14-4c	150	5	25	10	30	10	25	10	200	5	
25	1009 15.30c	150	5	25	10	30	10	25	10	200	5	

Figure 2.6: Today’s spreadsheets contain rows and columns of cells that can contain values and equations.

2.4 Spreadsheets

Spreadsheets are collections of cells and formulas, with the cells are arranged in a tabular layout. By producing formulaic relationships between cells or groups of cells, simple applications can be constructed without the need to learn a formal programming language. Much of the popular spreadsheet programs such as Excel or Calc are used this way. Figure 2.6 shows an example of a spreadsheet interface.

Extending spreadsheets to drive animations and interactive graphics is an old, but useful idea [33, 58]. By omitting the tabular layout in the spreadsheet, systems like NoPumpG [33] were able to provide a free-form layout of cells and link them to properties of graphics, such as position. Thus, by altering the values contained in cells via sliders and other methods, they could alter the graphics being displayed. This resembles dynamic updates in graphs of today’s common spreadsheet programs, but in a way that is generalized to custom graphics. The spreadsheet paradigm has also been further extended to allow cells to contain not only single numbers, but also more complex objects such as visualizations [14]. An example of this is seen in Figure 2.7. By displaying many coordinated views of the same data and applying different filters or attributes to the cells, it is possible to visualize different aspects of the data simultaneously.

Combining the speed of creating mathematical models using spreadsheets with the ease of sketching approximations of scalar functions, the function sketching

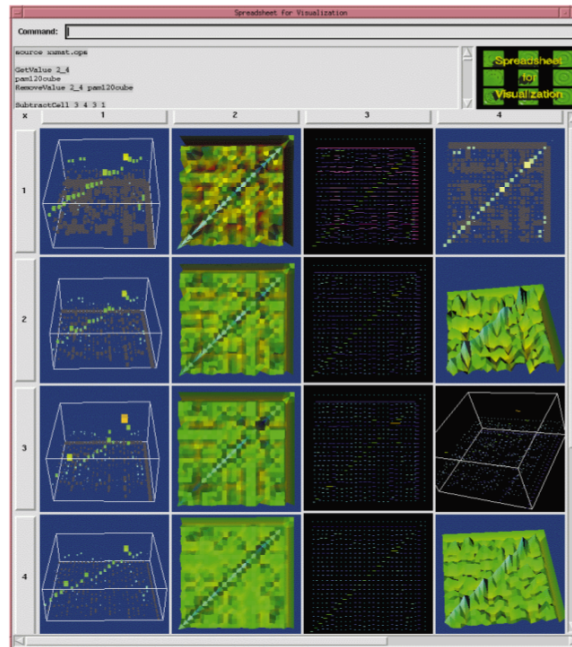


Figure 2.7: Spreadsheets have been augmented to include artifacts in their cells, such as full visualizations. (Reproduced with permission from: [14])

component of this thesis provides a tool that quickly authors and interacts with improvised demonstrations based on scalar functions.

2.5 Visualization Authoring Tools

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, which stems from the Polaris project [53], 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. Both the approach of wizards, as well as the approach of tools such as Tableau share the common goal of mapping visualizations to parameters that specify previously defined visualizations. These tools are best suited for situations where data can be best represented and communicated by well

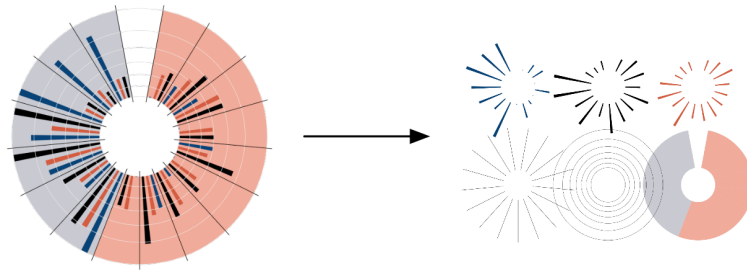


Figure 2.8: Protovis is a declarative Javascript toolkit that regards visualizations as a combined collection of marks, each with properties such as position and size specified by data. (Reproduced with permission from: [7])

known visualizations, such as bar charts. However, in situations where a non-traditional encoding, or a unique customization is desired due to the nature of the data, it would be impossible to use predefined visualizations in most cases.

An approach to producing custom visualizations without writing code is to use some sort of generalized animation software. Programs designed for kids such as eToys [28] or Phun [19] allow for the creation of virtual 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, thereby enabling user-directed timing if used in a presentation setting. By incorporating more pen-based interaction, some other works come even closer to the mark of sketch-oriented design [15, 48, 49]. The advantage of using more general purpose tools when compared to packaged visualizations, is that a finer-grained level of control can be achieved in the final visualization produced, particularly when defining the behaviour of individual elements. The major disadvantage of this approach, however, is that these programs often do not easily support the linking of data in many situations and with many types of data. This leads to the difficulty that any visual encoding of data dimensions, such as a column in a spreadsheet, would have to be done manually. In other words, it would not be obvious how to create a class of rules and automatically apply these rules over each dimension of data being visualized, thus making it unclear how to easily drive a visualization consisting of a group of visual

artifacts, each representing a dimension of data.

Fortunately, several toolkits exist that allow for the prototyping of visualizations, also at a finer-grained level compared to prepackaged visualizations. These toolkits follow the idea that visualizations can be thought of as layouts of grouped, parameterized visual marks [59]. Protovis [7] is a declarative Javascript toolkit which allows for a very rich set of visualizations to be specified and programmed in minutes. A visualization created in Protovis seen in Figure 2.8 demonstrates visualizations as combined groups of marks. Prefuse [25] and Flare 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 [20, 50], however we argue about the ease of significantly extending these visualizations due to the need to be able to first understand how they are implemented. There is also another flavour of toolkit also available to the designer. Processing [1] is another toolkit which lets designers focus on programming visual marks without worrying about any complicated underlying programming structures (for instance, in contrast to being required to initialize many non-obvious classes and pointers required by other languages, just for the purpose of being able to draw something on-screen). Unlike Protovis, this does not 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.

The advantage of using a toolkit to specify visualizations is that many tedious operations that are common across many types of visualizations such as representing data, linking data to marks, and so forth, are implemented in advance, potentially saving the programmer a great deal of time. Furthermore, the level of customization available to the user is nearly limitless, given enough time. The disadvantage however, is that knowledge of programming, and competency in the toolkit being used are both required. In many instances one or both requirements can provide a steep learning curve that needs to be overcome.

Combining Protovis and sketch, the visualization aspect of this thesis [11] cre-

ates an easily accessible tool with a flexibility of creating visualizations and a speed of interaction somewhere in between Tableau and Protovis, without the need to program code.

Chapter 3

Function Sketching

If you have ever attended a class such as high school science, economics, mathematics, or similar, there is a chance you might have encountered a lesson that involved learning how to describe some phenomenon with a system of scalar functions. In other words, the item being described could be represented as more than one mathematical function somehow interconnected, such as demonstrating wave interference by adding together two sine waves that are out of phase.

Chances are that words alone did not adequately convey a complete understanding for most people. For some, writing down a system of equations would be an approach that could have led to success, particularly if the functions that comprise the equations are already very familiar. For many others, pictures of the example functions and how the system affects them would have to be drawn to adequately convey an understanding of the system being explained.

Figure 3.1 shows an example of functions that were drawn on dry-erase boards to illustrate motion in one-dimension, amplitude modulation, the additivity of the Fourier transform, and profit/loss over time for a case study. In each of these cases several plots were drawn, and in many cases multiple plots contained more than one plot that was functionally related to another. For instance, in the motion example, the acceleration plot is the derivative of the velocity plot with respect to time.

Despite the relative speed and ease offered when communicating scalar-function based systems, manually drawing static functions on media, such as paper or the board, has several drawbacks. First, it is somewhat time consuming to draw many

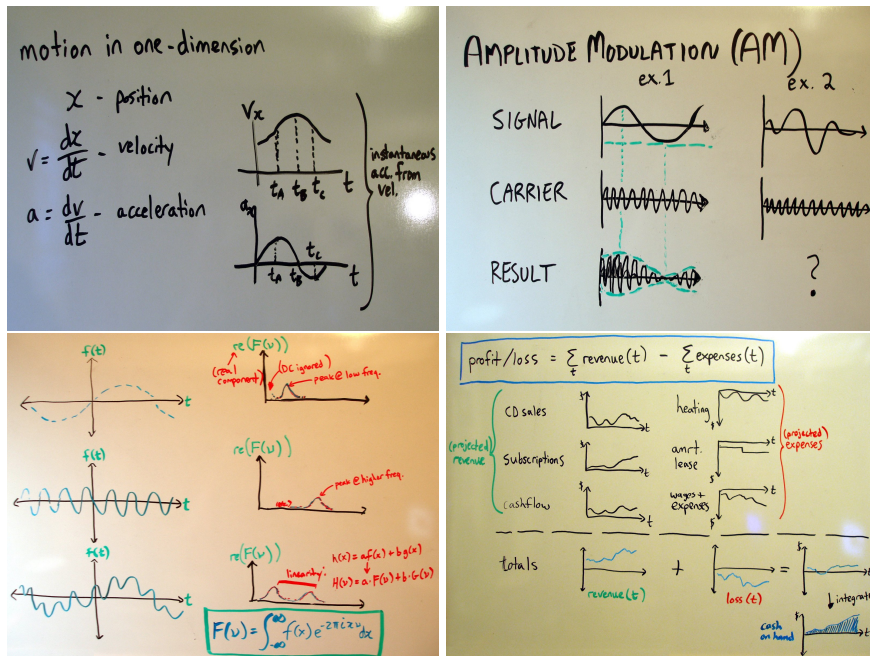


Figure 3.1: A sample of white-board drawings one might find in classes or meetings that include plots of scalar functions. The plots are used to qualitatively demonstrate concepts, and multiple plots tend to have more than one plot that are connected by some functional relationship.

functions, especially functions that relate to one another, as key parts of the various plots need to correspond to sections of other plots. Next, those with less-than-stellar drawing skills may not be able to reproduce the drawings of the functions they desire to demonstrate. Another drawback is that providing additional examples of the system require drawing every plot of the input functions, intermediate steps, and final products. This could be difficult if the system being demonstrated has many intermediate steps, such as a neural network. Finally, the people being communicated to, such as students, often cannot explore the ideas being conveyed using examples of their own when given manually-drawn, static plots. To help address these problems we utilize a spreadsheet-like interface.

For our purposes, spreadsheets afford several useful properties: Spreadsheets can let us perform complex calculations. We can visually construct a number of useful tools without knowing how to program. More interestingly, we can substi-

tute different data into a previously constructed spreadsheet and see how the results change without having to reconstruct all of the previous calculations from scratch.

The typical spreadsheet however, has the limitation that each cell contains only a single number, making operations on entire curves and functions quite cumbersome. This leads to the question, what if each cell represented a fully-typed scalar function? This leads to a new interpretation of the spreadsheet whereby the goal is to now construct full mathematical models when linking cells by equations.

This approach, however, introduces an obstacle that we quickly encounter: some curves that look very simple can actually have a fairly complex equation. While our system currently handles one-dimensional scalar functions, for the purpose of illustrating this point let us look at a familiar two-dimensional example. Take the following curve which has the equation:

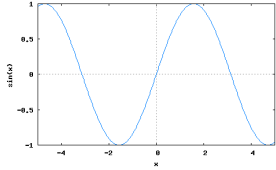
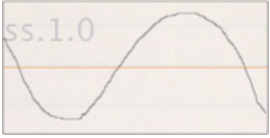
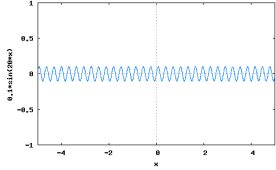
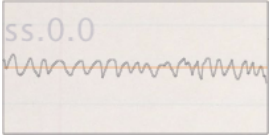
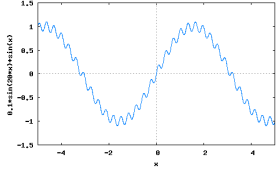
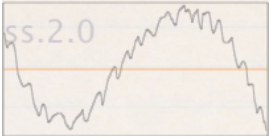
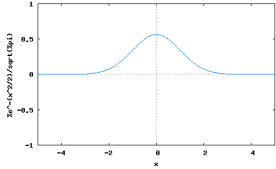
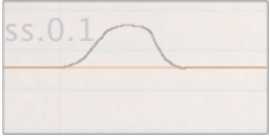
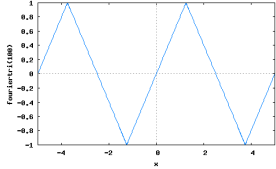
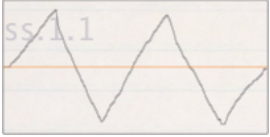
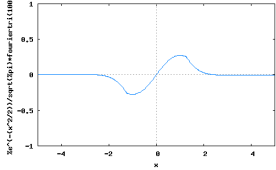
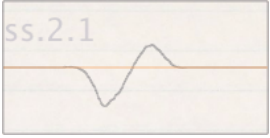
$$(x^2 + y^2 - 1)^3 - x^2y^3 = 0 \tag{3.1}$$

Although the above equation looks a bit complicated to some, it draws out a very familiar shape: a “heart” curve [54]. As an illustrative and (hopefully) fun exercise, we will leave it to the reader to convince yourself of this curve’s heart shape, and for those who do not already know how to quickly do this, take note of the effort required to generate a shape from an equation while attempting.

So rather than writing out the equation for every curve we wanted in a cell, what if we instead, drew out an approximation of the curves and used those approximations directly? One result is that we would be able to qualitatively see what effect a system of equations would have on the curves. In many settings, such as teaching the behaviour of functions in a classroom, the demonstration of qualitative effects would be adequate. Table 3.1 illustrates this by comparing plots of functions to their approximate sketches. In this table, the third and sixth rows are computed results. In other words, rather than being manually sketched, they are instead calculated from other sketched curves. We can see from this table that qualitatively speaking, the function sketches behave similarly to the actual function plots, with the added benefit that the memorization the functions in the left-most column is not required in order to produce the plots.

By combining the generality and versatility of a spreadsheet with the speed

Table 3.1: A comparison between functions, their plots, and approximate sketches of them using our interface. Note that the third and sixth function sketches are computed from the two above them, and are not manually drawn.

Formula	Plotted	Sketched
$\sin(x)$		
$\frac{1}{10} \sin(20x)$		
$\sin(x) + \frac{1}{10} \sin(20x)$		
$\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$		
$\frac{8 \sum_{n=1}^{\infty} \frac{(-1)^{n+1} \sin(2\pi f(2n-1)t)}{(2n-1)^2}}{\pi^2}$		
$\frac{8e^{-\frac{(x-\mu)^2}{2\sigma^2}} \sum_{n=1}^{\infty} \frac{(-1)^{n+1} \sin(2\pi f(2n-1)t)}{(2n-1)^2}}{\sqrt{2\pi^5\sigma^2}}$		

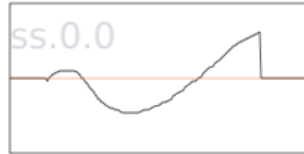


Figure 3.2: A cell actor is used to visually represent a function.

and ease of sketching, we provide a tool to generate models quickly enough to be used in a live setting. Creating a model is as simple as sketching the component functions, and linking them together with equations in a similar fashion to a spreadsheet. For instance, a simple example of this was seen in Section 1.2, where the sum of two cells is demonstrated. Any cell that makes up a model can be sketched on to redefine the function in the cell. Any cells that depend on the modified cell will also be updated accordingly.

3.1 Work-Flow

The general process of building a function-based model consists of defining functions and linking them by equations. The definition functions is facilitated by cell actors, which act as spreadsheet cells that can be sketched on. Cell actors can be linked by one or more spreadsheet equations. The input of equations has been implemented via keyboard entry in order to focus on canvas-actor interaction. However, equation entry could be implemented as purely sketching by incorporating a handwriting recognition library and a sketching area for equations.

An example of the spreadsheet cell actor is seen in Figure 3.2. Like a spreadsheet cell, values can be inserted, edited, or linked to a formula involving other cells. Unlike the typical spreadsheet cell, the cell actors in this canvas contain functions rather than atomic values.

A cell actor is generally placed on the canvas by two means: using the spreadsheet mode, whereby all altered cells are automatically placed on the canvas; or by creating a proto-actor and transforming it into a cell actor. The work-flow example in this section will use the spreadsheet mode to construct a functional model.

The user can toggle the spreadsheet mode whenever it is desired. In this mode,

ss.0.0	ss.1.0	ss.2.0	ss.3.0
ss.0.1	ss.1.1	ss.2.1	ss.3.1
ss.0.2	ss.1.2	ss.2.2	ss.3.2
ss.0.3	ss.1.3	ss.2.3	ss.3.3

Figure 3.3: The spreadsheet mode arranges cell actors in a spreadsheet layout.

several cell actors are placed into the spreadsheet automatically so the user does not need to create them manually. Cells are placed and named similar to a spreadsheet, as seen in Figure 3.3. Interaction in this mode mimics the familiar interaction of a spreadsheet.

Functions are drawn on to cells, mimicking how one would populate a spreadsheet with values. To make the spreadsheet more interesting these cells are then combined by equations. In this example, functions are drawn on two cells in the spreadsheet as shown in Figure 3.4.

To demonstrate a simple sum of functions, a cell is set to display the sum of two other cells. This is shown by the reverse-polish-notation (RPN) equation in Figure 3.5.

Once cells are linked in a model with equations, the cells making up the model can be modified and the cells containing the results will be updated. This can be seen in Figure 3.6.

Cells can be repositioned and interacted with in other canvas modes. In the presentation mode for instance, freely-arranged cells can be drawn in, dynamically updating other cells linked to it. Annotations, sketches, and other artifacts can

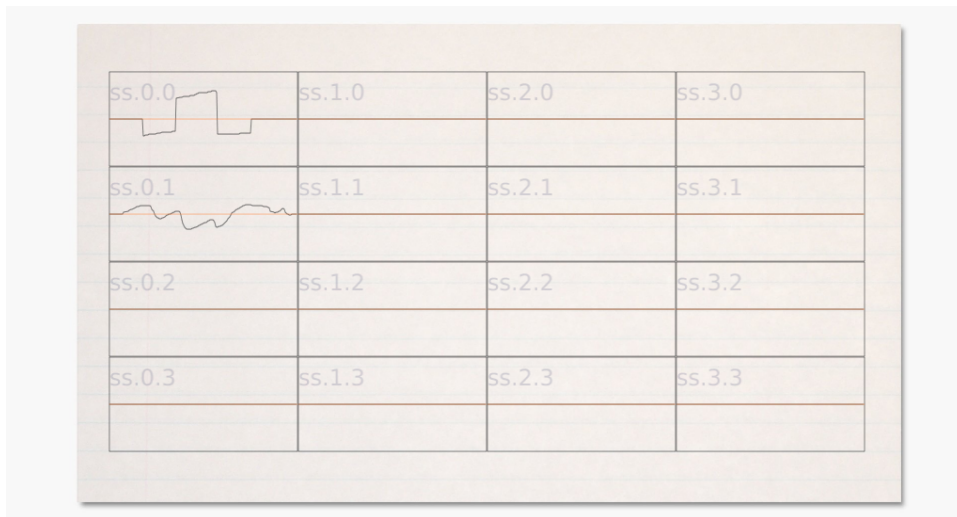


Figure 3.4: Functions are drawn in cells ss.0.0 and ss.0.1.

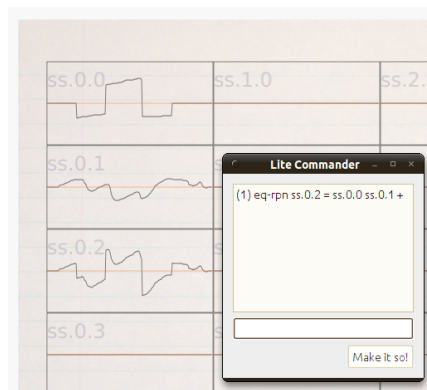


Figure 3.5: Cell ss.0.2 is set to be the sum of cells ss.0.0 and ss.0.1.

be placed around the active cells, providing clarification of the artifacts or simply improving the overall visual appeal. The final result can give the appearance of an interactive demonstration where novel examples can be improvised during a presentation, (such as with a prepared, embedded applet), as seen in Figure 3.7.

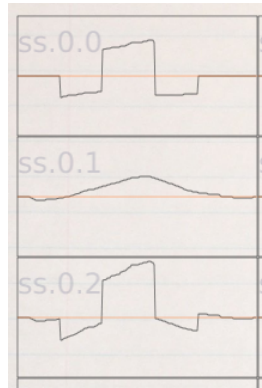


Figure 3.6: The function in ss.0.1 is redrawn and the result in ss.0.2 updates accordingly.

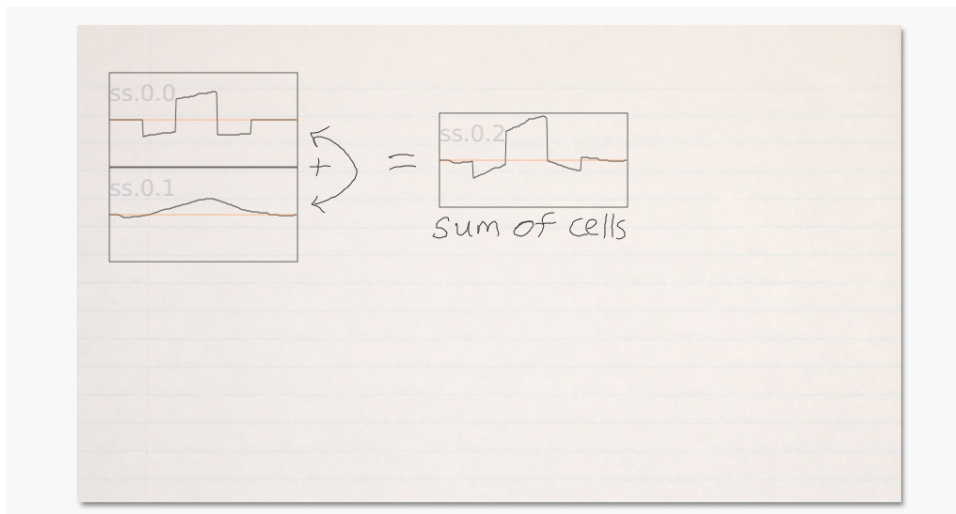


Figure 3.7: Any cell involved in an equation can be shown in the presentation modes and can be interacted with.

3.2 Implementation

The basic cell actor consists of a drawable area with a visible line graph to plot the stored function. The function appears as drawn, and the resolution of the discrete points of the function is limited to pixel resolution. The zero point is located in the middle of the drawing area, and the horizontal and vertical scale of all basic

cell actors match one another, allowing for easy qualitative comparisons between functions. Any visualization that can store data compatible with the basic cell actor can also act as a cell actor. These extended cell actors may include enhancements such as horizontal and vertical scaling, alternative representations of the function, multiple functions per actor, and more.

The scalar functions themselves are implemented as vectors of coordinate values, with step-wise, linear, or polynomial interpolation used to render the curves and determine the result of mathematical operations, depending on the specific situation. We found that this precision of interpolation provided an acceptable balance between performance and qualitative results of commonly used functional models.

There are several operations supported in the spreadsheet. These make up a proof-of-concept set, and are far from a complete set that would comprise a fully functioning spreadsheet. That being said, adding additional operations is an easy task by design, and can be done by writing classes that implement the operations interface we have defined internally.

The current allowable operations implemented are:

- addition
- subtraction
- multiplication
- scaling
- division
- convolution
- normalized cross-correlation
- Discrete Fourier Transform (without phase information)
- differentiation
- integration
- smoothing
- square root
- operations on constants

Equations that bind cells together are internally stored in a list of equations. All stored equations are in RPN (Reverse-Polish-Notation) form as calculations are performed trivially, and many other forms of representing equations can be reduced to RPN. For instance, currently equations can be directly entered in RPN format, or in traditional infix notation using a limited Shunting Yard algorithm [18] to convert the equations to RPN.

In addition to the list of equations stored, hash maps are used to index cell dependencies in an equation, in addition to equations dependencies of particular cells. This allows for constant time lookup of dependencies during any operation that affect multiple cells, such as updates, creating new equations, etc. All atomic cells with data but no dependencies are also noted. This is to prevent cells from updating if they depend on an undefined cell in an equation. When a cell is updated, any cells affected by this change are also updated. Additionally, when entering equations, invalid equations as well as equations with cycles are blocked from being processed.

This implementation results in a spreadsheet-like behaviour between drawn functions, with operations fast enough to update in real time.

3.3 Results

The sketchable spreadsheet described in this chapter can be used in both the spreadsheet mode, or in an more free-form mode, as seen in Figures 3.8 and 3.9. The spreadsheet mode allows for construction and manipulation of functions and equations in its cells, which is appropriate for exploring a new model, concept, or system of functions, among other things. The cells can then be arranged in a mode more appropriate for demonstration. This combined interface can be used to construct many kinds of useful demonstrations.

A sample demonstration illustrating normalized-cross-correlation is shown in Figure 3.10. In this example, the spreadsheet mode was used to apply normalized cross-correlation [8, 44] between two cells and display the result in a third cell. The cells were then re-positioned in canvas mode and some additional ink was drawn to quickly demonstrate what is roughly happening between the cells. The function or the kernel can now be drawn on to show the effect this has on the result.

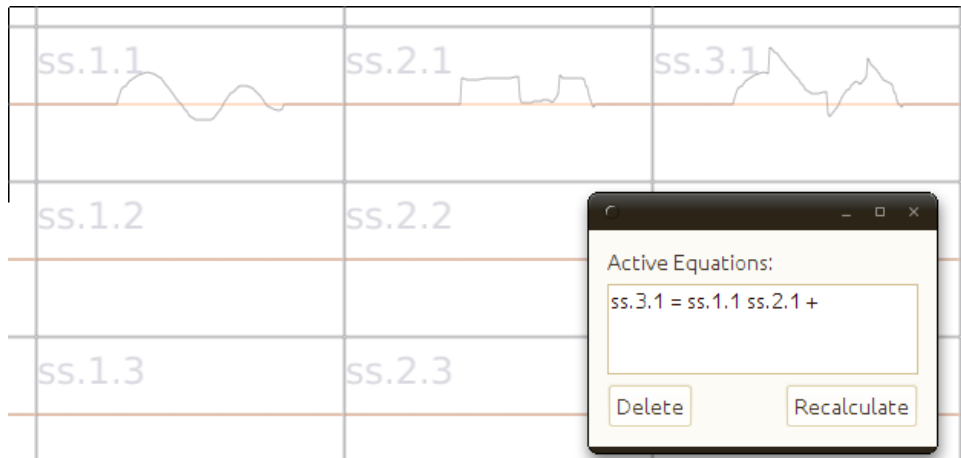


Figure 3.8: A simple addition of functions in spreadsheet mode.



Figure 3.9: A simple addition of functions in canvas mode.

Figure 3.11 illustrates a more involved demonstration showing a simplified explanation for amplitude modulation, a method used to encode signals onto a carrier wave as seen in older radios, where a sound signal is carried on a radio wave. In this setup, both the carrier wave as well as the signal can be drawn in to see how this affects the final wave.

The next demonstration, which is often introduced in computer vision classes demonstrates that if you apply normalized-cross-correlation (or convolution) often enough using the same kernel, you will eventually end up with a Gaussian (i.e. bell) curve. This is seen in Figure 3.13, in which a student or instructor can draw in both the function and the kernel, and the final result will be shown. Without some interactive demonstration, this concept is either accepted at face value by the students of the class, or attempted to be drawn on a dry-erase board to some

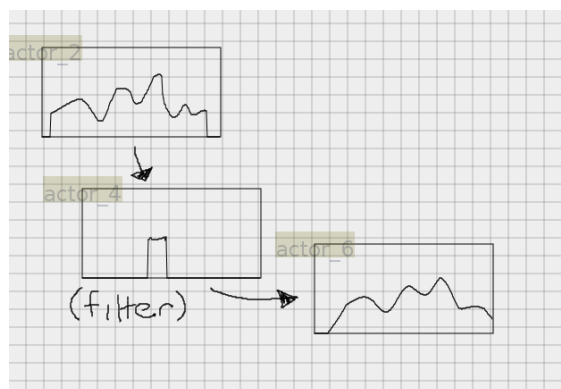


Figure 3.10: A demonstration in canvas mode of normalized cross-correlation for signal detection.

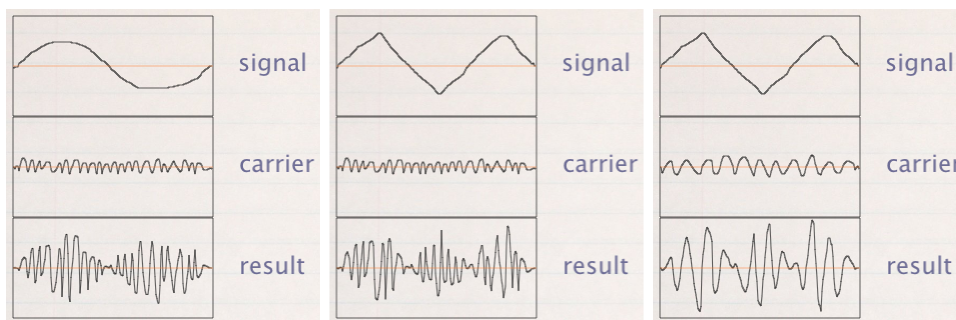


Figure 3.11: A demonstration in canvas mode showing amplitude modulation. The center and right image shows the result when a new signal or carrier are drawn respectively.

confusion.

The final example shown in this section illustrates the relationship between acceleration, velocity, and position. This can be seen in Figure 3.14. This concept is often used to demonstrate integration in beginner calculus courses. In an example class, graph paper was used to construct the curves, resulting in close to an hour just to produce a handful of examples. Using our interface, many examples can be constructed in seconds to explore this concept. Also note in this demonstration that scaling of functions is performed in the equations entered.

These are just a handful of the many kinds of demonstrations that can be con-

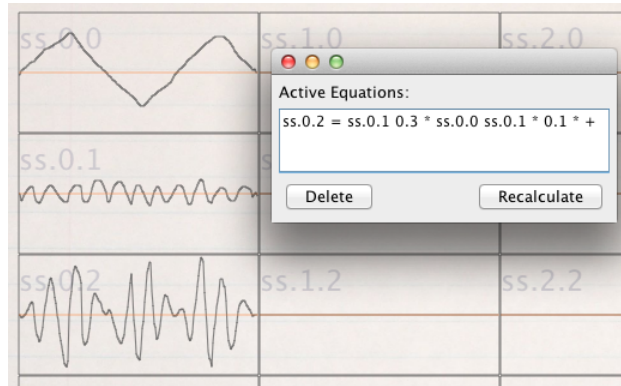


Figure 3.12: A function sketch demonstration showing amplitude modulation in spreadsheet mode.

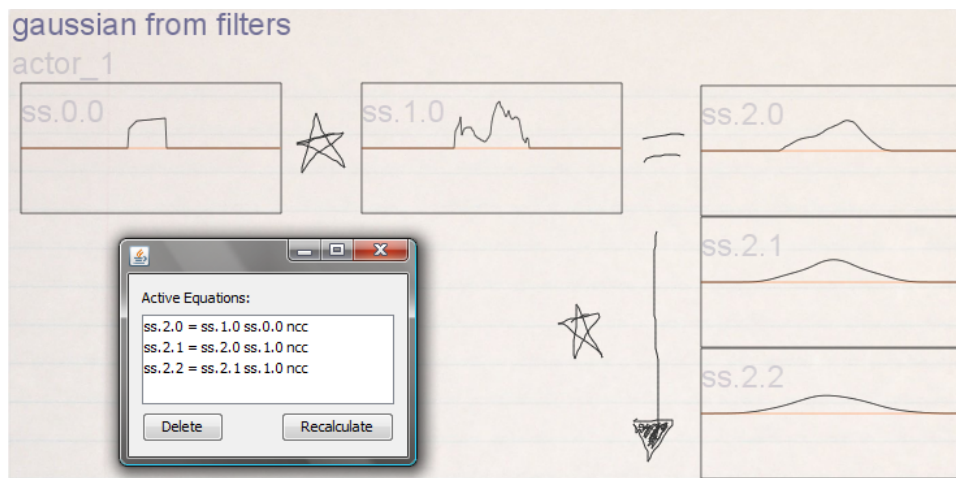


Figure 3.13: A demonstration in canvas mode showing that if you apply a kernel with normalized-cross-correlation repeatedly, you will eventually end up with a Gaussian. The reverse-polish-notation equations are shown in this figure. This demo is often shown in introductory computer vision classes.

structured and performed using the sketchable spreadsheet and canvas.

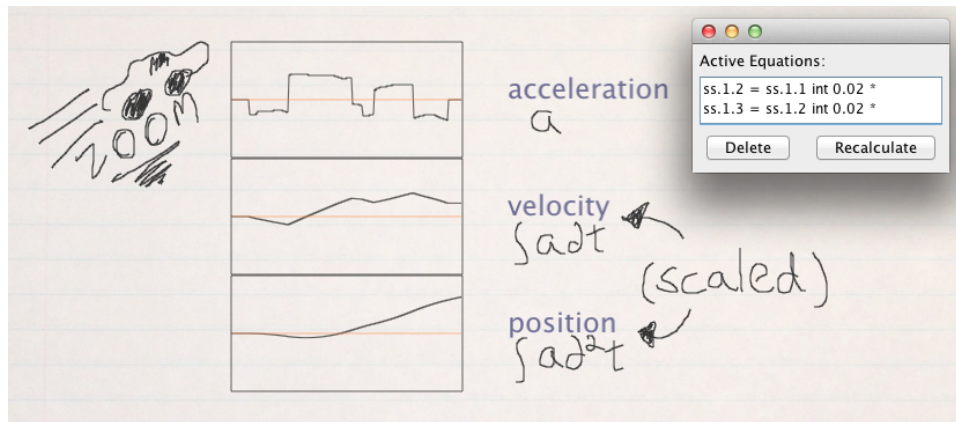


Figure 3.14: A comparison between acceleration, velocity, and position. This demonstration is often used to illustrate integration. The equations used to calculate velocity and position for this function sketch are shown.

3.4 Discussion

While the mechanism shown in this chapter provides a quick way to create qualitative demonstrations using ODO interaction, it should be noted that it is not intended for any functions that require precision in its definition, such as chaotic functions, the reason being two-fold. First, sketching functions by hand is inherently imprecise, and any resulting operations on these functions will propagate existing errors downstream. Second, interpolation within the sketched functions has been implemented with speed in mind, rather than extreme accuracy, in order to support interactivity. As a result, although the function shapes produced by our system can be used to qualitatively demonstrate a concept well within screen-resolution accuracy, this may not hold in situations that require an accuracy with more precision. For instance, in equations that require numerous applications of error-magnifying operations such as derivatives, noticeable inaccuracies will be visible in the resulting plots. To mitigate this error propagation somewhat, users can manually apply the smoothing operation to help stabilize the effects of these types of error-magnifying operations.

An additional point worth discussing is that the best practises for automatically scaling graphical representations of functions remains an issue that impacts

this interface. On one hand, a one-to-one scaling of functions allows for direct, qualitative comparisons between them. On the other hand, a one-to-one scaling can quickly make it difficult to represent intermediate results in systems of equations involving many different scales. Therefore, we have left automatic scaling as an open problem, noting that scaling can be manually performed in equations if needed by the user. This allows the user to have full control of the scaling behaviour, and allows for quick qualitative comparison between functions, however at the expense of increased authoring time.

Evaluation is often performed during the construction of a novel interface. However, formal, structured evaluation of this visual spreadsheet has been avoided as it could be considered counterproductive in this stage of development [22]. Currently comparable interfaces such as spreadsheets, functional programming languages, symbolic manipulation systems, and presentation software, among others, are at a mature state and a direct comparison could possibly only bring to the surface shortcomings due to a lack of implemented familiar features, such as custom formatting, context menus, wizards, and much more. Prior to evaluation, these features will need to be implemented either by programming this interface as a plug-in to a mature software suite such as LibreOffice, or by surveying the most commonly used subset of features and including as many as possible to allow testing of a very specific set of use cases. Additionally, the scripting back-end could be adapted to allow for Wizard-of-Oz testing of a few informative scenarios prior to this investment of resources. These lie outside the scope of this thesis and have therefore been left as future work. At a later stage they will be necessary to test the real-world, long-term viability of this interface.

Chapter 4

Visualization Sketching

Visualizations are important in presentations as they allow the presenter to quickly represent concepts being shown in various ways that may be easier to understand to an audience. Our implementation is a gesture-based WYSIWYG interface for building improvisational visualizations. We build on top of the Protovis visualization toolkit [7], while maintaining the on-(drag)-off interaction easily used in presentation settings. This allows users to quickly build various kinds of visualizations if needed without having to write code, use confusing wizards, or open bulky external programs, such as spreadsheets, which may detract from the flow of a presentation.

4.1 Work-Flow

The general process of sketching visualizations involves invoking the appropriate actors that will help with selecting a data source and parameters for the underlying scripting language, and linking these to an actor that knows how to render the final visualization. In this case the *data actor* and *single-mark visualization actor* perform the former tasks, and the *compound visualization actor* generates the final visualization.

The presenter begins by drawing on the canvas. All open strokes are inked as regular strokes while any closed strokes are shown as filled shapes called proto-actors, seen in Figure 4.1. These proto-actors are converted to various other actors by drawing gestures inside them. The following steps in this example are each

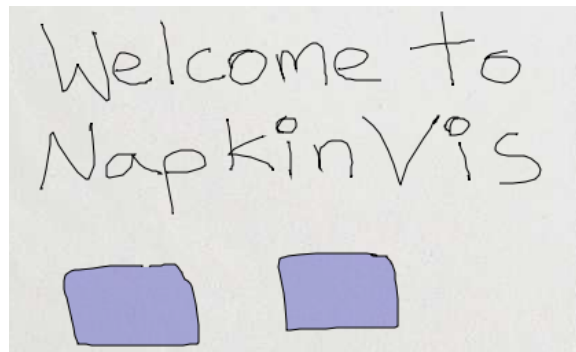


Figure 4.1: Drawing strokes and proto-actors.



Figure 4.2: To import data a proto-actor is converted into a data actor which is then used to open a spreadsheet. The spreadsheet data is displayed as a series of bar charts.

facilitated by a specialized actor.

To create something more interesting than a drawing the presenter imports data as seen in Figure 4.2.

The presenter then picks a mark to visually encode the data with. This is shown in Figures 4.3 and 4.4.

After picking the marks being used to encode the information, the user then links the information to the marks. This is demonstrated in Figure 4.5.

Once information is linked to the visual marks, these marks are then combined to form a visualization. This is shown in Figure 4.6.

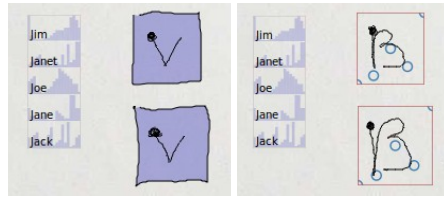


Figure 4.3: A gesture converts a proto-object into a visualization, and another gesture picks the visual mark.

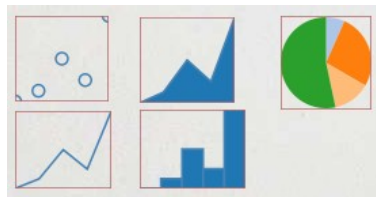


Figure 4.4: The user can pick from several marks to visually encode information.

4.2 Implementation

There are several elements that comprise our system and add various functionality to it. Each of these components allow the user to either quickly view simple visualizations, or to author more complex visualizations. A quick summary of these elements is seen in Figure 4.7.

The canvas is a fixed-size area that can be sketched on. In the current early implementation, it is simply a napkin-sized, sketchable area in a website which responds to mouse-down, mouse-up, and mouse-dragged events. The back end

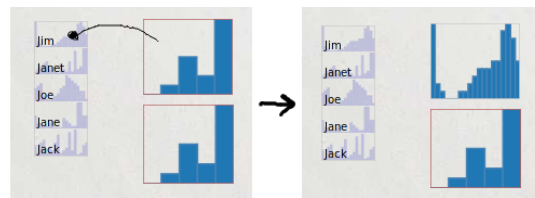


Figure 4.5: The information can be linked to a visual mark by brushing and linking, in other words drawing a stroke from the information to the visual mark.

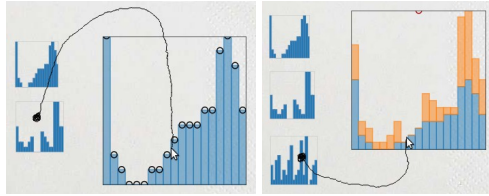


Figure 4.6: Marks are combined in a compound visualization.

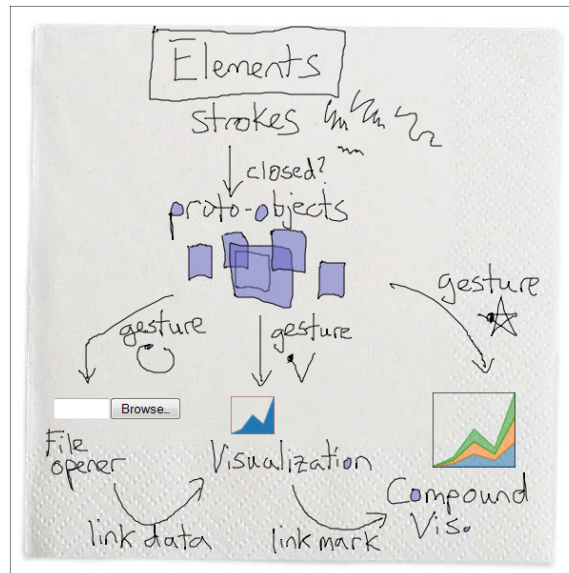


Figure 4.7: Elements of the visualization sketching system, demonstrated in its own interface.

renders the canvas and visual actors in several layers 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.

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-actors, gestures, links, and so forth.

When the user draws a closed stroke, the stroke becomes transformed into a proto-actor. Proto-actors can then be transformed into other elements by drawing a gesture inside of the proto-actor. In the context of a drawn web page with a

visualization, the proto-actor provides a means of specifying the location and size the user might want. For instance, if the user wanted to create a visualization location (25, 20) with a size of around 100x150, they could simply draw a box of that size with the top-left corner at (25, 20) creating a new proto-actor, then draw a gesture inside the proto-actor to create a visualization.

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 [32] to compare the stroke with predefined gestures. Although the gesture vocabulary can be customized to suit the user, we will use an example set of gestures in the following paragraphs for the sake of concreteness.

A file opener is a tool that allows the user to specify the path of a file to open. This tool can be invoked via an “O” gesture drawn in a proto-actor. Once a file is chosen, this widget is transformed into a data actor which gives access to the data in the file. The file opener was programmed for convenience of implementation and in future versions added support for sketching will be included to complement this tool.

For the scope of this project it is assumed that all information visualized is tabular data with an evenly-spaced independent variable. This can be provided by any compatible canvas actor such as the data actor.

The data actor provides the first real visualization of the information, 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, this tool may be all the user will need. Each row in the visualization represents a dimension in the parsed file. In essence, this actor performs like a scented widget to allow selection of a dimension of data [60].

Single-mark visualization actors are visualizations that contain a single mark for each data item, such as a line graph, bar chart, area graph and scatter plot. These can be invoked by drawing a ‘V’ gesture inside a proto-actor. The position and size of the visualization are determined by the respective properties of the proto-actor. Additional gestures can change the visualization’s mark after the visualization has been invoked. For instance, an ‘A’ gesture selects the area mark, used for area charts. The ‘W’ gesture selects wedges, used in making pie charts and similar visualizations with a radial layout. The ‘D’ gesture selects dots, commonly used

in scatter plots. A 'B' gesture causes bars to be selected as the desired visual encoding. Finally, a 'L' gesture selects lines as the desired visual encoding.

The single-mark visualization displays a default visualization to let the user know what it will look like. A brown border around the visualization indicates this behaviour. When any data is linked to the visualization the border changes color to indicate this.

In order to link information to the single-mark visualization, the user draws a stroke starting in any actor containing compatible data such as the data actor, to any point in the visualization itself.

When brushing and linking data from the data actor to the single-mark visualization, in order to specify which data dimension the user wants to visualize the starting point of the link must lie within the mini visualization of that dimension in the data actor.

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 draws a 'star' gesture inside a proto-actor, similar to other actors.

To then add a mark to the visualization, the user simply draws 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.

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 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.

On the front-end of the compound visualization actor one uses ODO interaction to combine marks to build visualizations. On the back-end, this actor maps

interactions on the front-end and data provided by sources mapped to marks to a generated Protovis-compatible script.

In order to do this, all current marks linked to this actor, their associated data, and anchors to other linked marks are kept track of. The remaining two pieces of information that needs to be specified for each group of marks are: which directional anchors should be used to position a group of marks relative to an existing group of marks, and which existing group of marks should a group of marks be placed on. Actually, these two requirements can be reduced to simply specifying which existing anchor in the compound visualization should a new mark be placed on, but conceptually it is easier to split the task into the two tasks of specifying which anchor, and which group of marks to place new marks on.

Since Protovis provides every mark in a visualization with a top, bottom, left, right, and a center anchor, the compound visualization actor needs to support interaction for specifying which existing anchor to place a group of marks on. For this, we have experimented with making the direction of entry to the actor when linking a mark mapped to the top, bottom, and similar anchors of existing marks in the visualization. Center anchors are ignored in this case, however it would be easy to include them.

In order to determine which existing mark to stack the new mark on, the compound visualization is partitioned into sections that one can drop a mark on to. If no marks exist, the entire actor has its own set of anchors that can be used to stack new marks. Each section represents an existing mark that has already been stacked on to a previous mark. For instance, if there exist two marks in a stacked bar chart, the bottom half of the compound visualization would correspond to the first mark stacked in this visualization, and the other half corresponding to the other mark. With three marks the compound visualization is partitioned into thirds. Because a visualization with many marks (more than the pixel density of a visualization) would be overly complicated, given the domain of this system, partitioning of the compound visualization should not enter into impossible sub-pixel partitions, practically speaking.

The necessary information needed to create a compound visualization can be stored in a table where each row consists of marks, their types, attributes, a pointer to data, a destination anchor, and various other customizable attributes. Construct-

ing the final visualization requires iterating through the rows of this table and generating a script that can be parsed by the declarative visualization language.

4.3 Results

There are a wide variety of visualizations that can be dreamed up and authored using the currently implemented subset of interactions. We will show a few examples in this section to help jump-start the imagination. The first example, seen in Figure 4.8 shows a visualization to show page views off a sample website. In this case, a quick-and-dirty set of graphics were created to compare the number of users who visited the survey page and the number of visitors on the front page during the week. There are two visualizations seen on this sketch, a stacked bar chart and a layered bar chart. These visualizations were both created by using the bar mark to visually encode the respective data, and they both differ by stacking. This visualization was created on a small form-factor smaller than the size of a napkin, in this case on a tablet's browser in a coffee shop where right-clicking was unavailable.

In order to create this visualization, spreadsheet data is imported using a data actor. Two single-mark visualizations are then created and bar marks were selected. Columns of data from the spreadsheet are brushed and linked to encode them both as bars. Two compound visualizations are created. In the top visualization, the bars are stacked on the bottom of the visualization to layer the visual elements. In the bottom visualization, the bars are instead stacked on one another. This visualization sketch, including all inked annotations and doodles took under thirty seconds to create.

To further illustrate visualizations that can be authored by visually encoding data and stacking marks, several created on the canvas are seen in Figure 4.9. The left shows the result of stacking area marks on both top and bottom anchors of existing marks. This provides the behaviour of a stacked area chart. To demonstrate that this behaviour is useful outside of the scope that a simple Excel wizard can provide, an example of a more complicated visualization that utilize the behaviour of stacking area marks is ThemeRiver [9, 23]. The middle visualization shown demonstrates superposition of a bar chart and an area chart. This is done by stack-

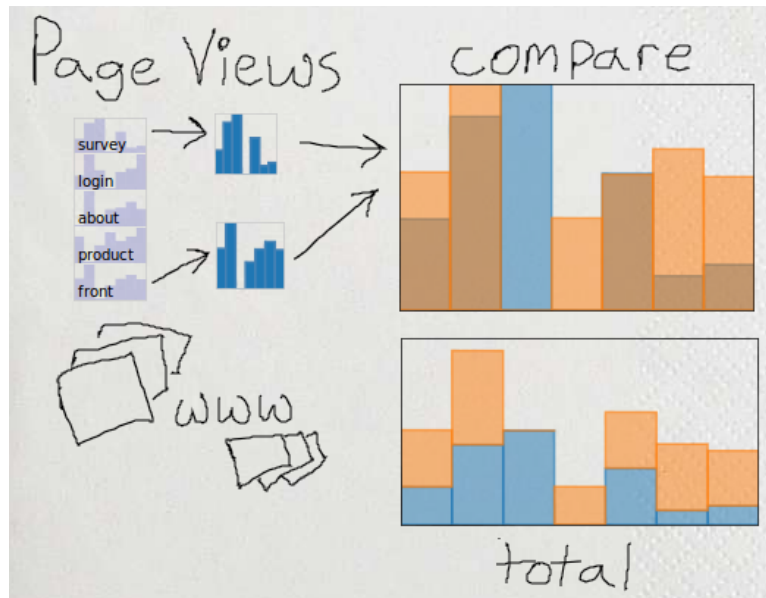


Figure 4.8: A sample napkin visualization charting one week of page views to a website. By stacking bars on each other versus stacking them on the bottom of the visualization we create two common types of bar-based visualizations.

ing marks on the same anchor. The example on the right demonstrates a stacked bar chart created.

These three visualizations were created using a similar procedure as the example with page view data. The difference is in the marks selected to visually encode the information, and the method of stacking the marks. For the left-most mirrored area chart, marks are stacked on top of one another, on both the top and the bottom. The middle mixed visualization demonstrates that visual marks can be combined. In this case, the area and bar marks are stacked on the bottom of the visualization. The stacked bar chart on the right is created the same way as the stacked bar chart in page view example. These three visualizations were created in under a minute total.

Finally, to help illustrate that providing some of the versatility of a toolkit to non-expert users can be useful, unconventional but potentially useful visualizations can also be created. In Figure 4.10, an unconventional stacked bar chart is created

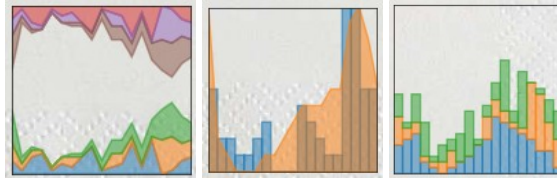


Figure 4.9: A mirrored, stacked area chart, a bar chart superimposed on an area char, and a stacked area chart created by combining area marks or bar marks in a compound visualization.

to compare the fuel usage of two valves in a facility. In this case, the result that is of interest is whether or not a valve’s flow of fuel, in addition to the baseline fuel usage, is above a threshold. In a spreadsheet, this would be done by either adding the baseline column to each individual valve’s column, then charting the result in a layered bar chart or two separate charts. In this visualization, the baseline fuel usage is stacked on the bottom of the chart, and the fuel usage for valves 1 and 2 are stacked on top of the baseline. This lets us see some interesting things. First, we see what the baseline fuel flow looks like over each day, something that would be hidden in the summed spreadsheet method. Second, we can see that on some days a single valve exceeds the threshold, whereas on others both do. We can see which valves they are, and which days this occurs quickly, without having to compare multiple charts. Finally, we can compare the trending fuel usage between valves and the baseline, among other things. Although this visualization is unconventional, it still provides some utility. This visualization sketch including all inked annotations was created in under thirty seconds.

All of these visualizations and many more can be created using only ODO interaction, which can be further extended in the future by redundantly mapping other conventional interactions like context menus.

4.4 Discussion

While the gamut of visualizations that can be created using canvas-only interactions is smaller when compared to when coding using a scripting language, this gap can be quickly narrowed by providing more interactions that map to addi-

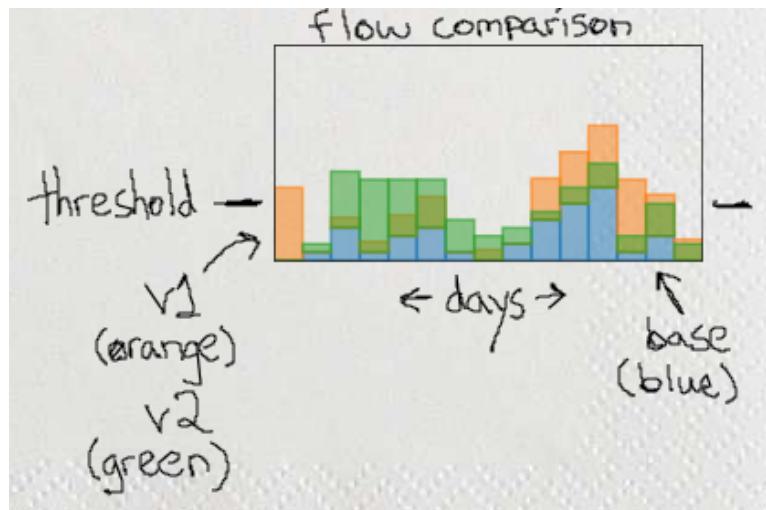


Figure 4.10: An unconventional visualization shows a comparison of total fuel usage between two monitored valves at a factory. Valve bars (seen in green and orange) are stacked on top of a baseline bar chart (seen in blue).

tional functionality of the language itself. Although this canvas is an early step, it demonstrates that just like WYSIWYG editors and visual programming languages, finer-grained authoring of visualizations can also be achieved using only ODO interactions, enabling the improvisation of them.

An open problem which is left as future work is the determination of a good set of default behaviour for visualization formatting. This includes issues such as ideal color of marks, shapes of marks, sizes of all visual artifacts, scaling along axes, inclusion of tick marks, labelling, and so forth. Although the capability to customize these properties can be included, because this interface may target non-experts, subtle perceptual issues such as the effect of aspect ratio on the perception of trends in graphs, may not be known [16, 24]. Thus, automatic handling of these subtleties should be provided by a more complete future version. While surveying this issue to completion is outside the scope of this thesis, the good news is that existing toolkits such as Prefuse or Protovis [7, 25] already provide a good set of default behaviours which can be built upon.

Another topic to discuss is the future evaluation of this interface. At this stage

of development, evaluation has been limited to several rounds of informal feedback from a group of expert users. While a structured, formal evaluation will be necessary to fully validate any claims of efficiency at a later stage, we would like to suggest that any evaluation at this state of development would only bring attention to familiar features that, for the purpose of properly focusing our resources, were intentionally excluded in the development of this interface [22]. This would therefore not be as helpful as evaluating a tool with these features included. For instance, a common question asked by early users regarded the notable absence of context menus, whereas a conscious decision was made to see how far we could go using gestures alone before extending our interface with well-established features.

There are several other aspects of this interface that fall into future work for evaluation of this interface. The first factor that will need to be determined, at least roughly, is the functional coverage that pen-based interaction can have over the original Protovis toolkit. In other words, how much of the scripting capability can we map to gestures and ink before requiring coding. This could be done by building on top of results previously found by works in visual programming languages. The next factor that should be investigated would be the ease of learning the concept of building visualizations by strategically encoding marks prior to mapping them to a data set. In other words, investigate how easily this concept of visual marks can be adopted and applied by non-expert users. Finally, familiar features in established programs should be included in a more mature product. Then a more comprehensive comparison between established tools and our interface can be conducted in order to test claims of increased usability, ease of learning, and efficiency, in addition to garnering of useful feedback.

Chapter 5

Conclusions

5.1 Conclusion

The variety of live situations that active diagrams can enhance are on the rise. For instance, presentations, something many people perform daily, can use interactive demonstrations to aid in conveying ideas more effectively when compared with static drawings. Because the kinds of human-computer interactions that can be performed in these situations can be limited, input methods that can be more widely generalized are useful to build on.

In this thesis a proof-of-concept canvas was shown that allows for on-(drag)-off interaction to be used to create and interact with function-based models and visualizations, bringing current systems a step closer to accommodating the improvisational use of them.

A visual mapping to a declarative scripting language for visualizations was used in order to provide finer-grained control in a proof-of-concept WYSIWYG visualization authoring tool. A sketch-capable spreadsheet was created to facilitate the creation of functional models.

5.2 Discussion and Future Work

There is an explosion of devices appearing as this thesis is being written that opens up a variety of interaction modalities. Tablets are making a come-back and catch-

ing on with the general public, multi-touch is the new craze, and even bodily gestures are being explored with stereoscopic, video-based game interfaces such as Microsoft's Kinect. Although it was tempting to try and work with some of these interfaces, we chose to try and interaction method that, in our opinion, was quite generalizable and hopefully easily extended to current technologies. That being said, it would be quite fun to try and map some of these new interaction methods to the functions of this canvas now that the framework has been laid out.

Our system assumes that it is worthwhile to support ODO (on-(drag)-off) interaction and focuses more on facilitating rough, qualitative, improvisational demonstrations rather than finalized and precise analysis. We believe that these can be widely generalized to more complicated systems. For instance, any device that accepts input from a tracking device such as a mouse, camera, tablet or a touch-screen can already support this type of interaction. Existing software technology such as virtual keyboards and sketch recognition can then build on this technology to provide even more interaction options.

While the system we presented can already be adapted for use in various settings, there still remains a number of tasks that are worth mentioning as future work.

A formal evaluation has been left as future work as one would be harmful at this stage of prototyping [22]. During informal feedback sessions with expert users, the issue of the lack of familiar features was a common theme that crept up. We suggest that, similar to Greenberg's suggestion, development was at too early a stage to be compared to current, well-established tools without highlighting very common, yet time-consuming features to implement. We feel that this re-implementation of common features is necessary, as without testing the interactions presented in this thesis integrated with existing interactions we are used to, it is hard to see how well our would integrate with them. It should be noted, however, that most feedback of our interface itself were quite positive. Any future version of the system could benefit greatly from being implemented as a plug-in to an established office software suite such as LibreOffice, as this would remove any distraction in work-flow caused by a lack of common features, and instead bring forward any genuine design flaws that should be improved upon in the interface itself. Fortunately, when that time comes, guidelines for testing interfaces such as

ours exist, (e.g., [10, 36, 37, 41]), and we will be drawing from them to guide our planned future evaluation.

It should be noted that the interactions explored in this thesis were also meant to be memorized by the presenter, thus giving the appearance of fluidity to an audience watching. Similar to keyboard shortcuts, these interactions are not meant to be learned on-the-fly and therefore just like shortcuts, a system would not be complete without a way to visually search for the function itself, such as menus or palettes. An interesting problem would be to determine if a subset of interactions and hints could be used that provide a good middle-ground between the two extremes while appearing to be fluid, or whether it would instead simply be worth it to implement both extremes and provide training for those interested in shortcuts.

In future versions, although the systems demonstrated in this thesis can be extended by reimplementing conventional interactions, such as context menus, to make interaction easier and more obvious, it should still be noted that with solely ODO interactions alone, there is definitely room for improvement. For instance, an issue that arises is that the system needs to be learned, rather than discovered by the user. More specifically, affordance of the different interactions is not obvious, and the visibility of how the different parts of the system interact could be further improved [42]. In other words, the user will not be able to easily figure out how to use these gestural interactions without some prior instruction. The flip-side to the hidden nature of ODO interaction as mentioned earlier, is that when learned, visual distractions do not get in the way of work-flow (such as attention-grabbing pop-up windows), and demonstrations remain fluid and focused as extraneous artifacts do not appear on screen.

Finally, while this project began with implementing a generalized presentation scripting language, it soon became apparent that the scope may be wider than anticipated. It would be interesting to revisit this endeavour as we feel that interesting systems can be made quickly when a good, well-established scripting back-end is created first. Such is the case with the visualization authoring sub-system of this project. A good portion of the canvas is scripted, however, so expanding this to the entire gamut of the canvas functions will be a fun next step. Once completed, the task of creating interesting presentation interactions will then be simplified to writing modules to generate scripting code from novel input devices.

Bibliography

- [1] Processing toolkit. <http://www.processing.org>. accessed on October 28, 2009. → pages 21
- [2] E. André and T. Rist. Generating coherent presentations employing textual and visual material. *Artificial Intelligence Review*, 9(2):147–165, 1995. → pages 12
- [3] S. Bae, R. Balakrishnan, and K. Singh. ILoveSketch: as-natural-as-possible sketching system for creating 3D curve models. In *Proc. ACM Symp. User Interface Software and Technology*, pages 151–160, 2008. → pages 13
- [4] S. Bae, R. Balakrishnan, and K. Singh. EverybodyLovesSketch: 3D sketching for a broader audience. In *Proc. ACM Symp. User Interface Software and Technology*, pages 59–68, 2009. → pages 13
- [5] B. Bailey, J. Konstan, and J. Carlis. DEMAIS: designing multimedia applications with interactive storyboards. In *Proc. ACM International Conference on Multimedia*, pages 241–250, 2001. → pages 14
- [6] B. Bederson and J. Hollan. Pad++: a zooming graphical interface for exploring alternate interface physics. In *Proc. ACM Symp. User Interface Software and Technology*, pages 17–26, 1994. → pages 12
- [7] M. Bostock and J. Heer. Protovis: a graphical toolkit for visualization. *IEEE Trans. on Visualization and Computer Graphics*, pages 1121–1128, 2009. → pages vii, 6, 20, 21, 38, 48
- [8] R. Bracewell. Pentagon notation for cross correlation. In *The Fourier Transform and Its Applications*, pages 46, 243. McGraw-Hill, 1965. → pages 32
- [9] L. Byron and M. Wattenberg. Stacked graphs geometry & aesthetics. *IEEE Trans. Visualization and Computer Graphics*, 14(6):1245–1252, 2008. → pages 45

- [10] S. Carpendale. Evaluating information visualizations. In *Information Visualization*, volume 4950 of *Lecture Notes in Computer Science*, pages 19–45. Springer, 2008. → pages 52
- [11] W. Chao, T. Munzner, and M. van de Panne. Poster: rapid pen-centric authoring of improvisational visualizations with NapkinVis. In *Two-Page Abstract Incl. in Electronic Proc. IEEE Conf. Information Visualization (InfoVis)*, 2010. → pages xi, 21
- [12] Q. Chen, J. Grundy, and J. Hosking. An e-whiteboard application to support early design-stage sketching of UML diagrams. In *Proc. IEEE Symp. Human Centric Computing Languages and Environment*, pages 219–226, 2004. → pages 13
- [13] K. Cheng and K. Pulo. Direct interaction with large-scale display systems using infrared laser tracking devices. In *Proc. the Asia-Pacific Symp. Information Visualisation*, pages 67–74. Australian Computer Society, Inc., 2003. → pages 2, 12
- [14] E. Chi, J. Riedl, P. Barry, and J. Konstan. Principles for information visualization spreadsheets. *IEEE Computer Graphics and Applications*, 18(4):38, 1998. → pages vii, 18, 19
- [15] M. Chuah, S. Roth, and S. Kerpedjiev. Sketching, searching, and customizing visualizations: a content-based approach to design retrieval. *Intelligent Multimedia Information Retrieval*, pages 83–111, 1997. → pages 20
- [16] W. Cleveland. *Visualizing data*. Hobart Press, 1993. → pages 48
- [17] R. Davis, B. Colwell, and J. Landay. K-sketch: a ‘kinetic’ sketch pad for novice animators. In *Proc. SIGCHI Conf. Human Factors in Computing Systems*, pages 413–422. ACM, 2008. → pages 15
- [18] E. Dijkstra. An ALGOL 60 translator for the X1. *ALGOL Bulletin Supplement*, 10:21–32, 1961. → pages 32
- [19] E. Ernerfeldt. Phun 2D physics sandbox. Master’s thesis, Umeå University, 2008. → pages 20
- [20] J. Fekete. The infovis toolkit. In *IEEE Symp. Information Visualization (InfoVis)*, pages 167–174, 2005. → pages 21

- [21] L. Good and B. Bederson. Zoomable user interfaces as a medium for slide show presentations. *Information Visualization*, 1(1):35–49, 2002. → pages vi, 12, 13
- [22] S. Greenberg and B. Buxton. Usability evaluation considered harmful (some of the time). In *Proc. SIGCHI Conf. Human Factors in Computing Systems*, pages 111–120. ACM, 2008. → pages 37, 49, 51
- [23] S. Havre, B. Hetzler, and L. Nowell. ThemeRiver: visualizing theme changes over time. In *IEEE Symp. Information Visualization (InfoVis)*, pages 115–123, 2000. → pages 45
- [24] J. Heer and M. Agrawala. Multi-scale banking to 45 degrees. *IEEE Trans. Visualization and Computer Graphics*, pages 701–708, 2006. → pages 48
- [25] J. Heer, S. Card, and J. Landay. Prefuse: a toolkit for interactive information visualization. In *Proc. SIGCHI Conf. Human Factors in Computing Systems*, pages 421–430. ACM, 2005. → pages 21, 48
- [26] D. Hils. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages & Computing*, 3(1): 69–101, 1992. → pages 17
- [27] B. Jonson. Design ideation: the conceptual sketch in the digital age. *Design Studies*, 26(6):613–624, 2005. → pages 12
- [28] A. Kay. Squeak etoys, children & learning. <http://www.squeakland.org/resources/articles>, 2005. accessed on January 16, 2010. → pages 20
- [29] J. Landay and B. Myers. Interactive sketching for the early stages of user interface design. In *Proc. SIGCHI Conf. Human Factors in Computing Systems*, pages 43–50, 1995. → pages vi, 13, 14, 15
- [30] J. Laviola Jr. *Mathematical sketching: a new approach to creating and exploring dynamic illustrations*. PhD thesis, Brown University, 2005. → pages vii, 16, 17
- [31] J. LaViola Jr and R. Zeleznik. MathPad²: a system for the creation and exploration of mathematical sketches. *ACM Trans. Graphics*, 23(3): 432–440, 2004. → pages 16
- [32] V. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics Doklady*, volume 10, pages 707–710, 1966. → pages 42

- [33] C. Lewis. NoPumpG: creating interactive graphics with spreadsheet machinery. In *Visual Programming Environments: Paradigms and Systems*, pages 526–546. IEEE, 1987. → pages 18
- [34] Y. Li, J. Landay, Z. Guan, X. Ren, and G. Dai. Sketching informal presentations. In *Proc. International Conf. Multimodal Interfaces*, page 241. ACM, 2003. → pages 12
- [35] J. Lin, M. Newman, J. Hong, and J. Landay. DENIM: finding a tighter fit between tools and practice for web site design. In *Proc. SIGCHI Conf. Human Factors in Computing Systems*, pages 510–517. ACM, 2000. → pages 14
- [36] T. Munzner. Process and pitfalls in writing information visualization research papers. In *Information Visualization*, volume 4950 of *Lecture Notes in Computer Science*, pages 134–153. Springer, 2008. → pages 52
- [37] T. Munzner. A nested process model for visualization design and validation. *IEEE Trans. Visualization and Computer Graphics*, 15(6):921–928, 2009. → pages 52
- [38] B. Myers. Creating interaction techniques by demonstration. *Computer Graphics and Applications*, 7(9):51–60, 1987. → pages 13
- [39] B. Myers. Scripting graphical applications by demonstration. In *Proc. SIGCHI Conf. Human Factors in Computing Systems*, pages 534–541. ACM, 1998. → pages 13
- [40] B. Myers, J. Goldstein, and M. Goldberg. Creating charts by demonstration. In *Proc. SIGCHI Conf. Human Factors in Computing Systems: Celebrating Interdependence*, pages 106–111. ACM, 1994. → pages 13
- [41] J. Nielsen. Usability inspection methods. In *Conf. Companion SIGCHI*, pages 413–414. ACM, 1994. → pages 52
- [42] D. Norman. *The design of everyday things*. Doubleday Business, 1990. → pages 52
- [43] D. Olsen Jr and T. Nielsen. Laser pointer interaction. In *Proc. SIGCHI Conf. Human Factors in Computing Systems*, pages 17–22. ACM, 2001. → pages 2, 12
- [44] A. Papoulis. *The Fourier integral and its applications*, pages 244–245, 252–253. McGraw-Hill, 1962. → pages 32

- [45] I. Parker. Absolute powerpoint: can a software package edit our thoughts. *The New Yorker*, 77(13):76–87, 2001. → pages 2
- [46] R. Patel, B. Plimmer, J. Grundy, and R. Ihaka. Ink features for diagram recognition. In *Proc. Eurographics Workshop Sketch-based Interfaces and Modeling*, pages 131–138. ACM, 2007. → pages 12
- [47] B. Plimmer and M. Apperley. Computer-aided sketching to capture preliminary design. *Proc. AUIC*, 7, 2002. → pages 12
- [48] S. Roth, J. Kolojechick, J. Mattis, and J. Goldstein. Interactive graphic design using automatic presentation knowledge. In *Proc. SIGCHI: Celebrating Interdependence*, pages 112–117. ACM, 1994. → pages 20
- [49] S. Roth, J. Kolojechick, J. Mattis, and M. Chuah. Sagetools: an intelligent environment for sketching, browsing, and customizing data-graphics. In *Proc. SIGCHI Conf. Companion Human Factors Computing Systems*, pages 409–410. ACM, 1995. → pages 20
- [50] W. Schroeder, K. Martin, and W. Lorensen. The design and implementation of an object-oriented toolkit for 3d graphics and visualization. In *Proc. Conf. Visualization*. IEEE Computer Society, 1996. → pages 21
- [51] T. Speeter. Intelligent work surfaces, June 16 1993. EP Patent 0,546,704. → pages 3
- [52] P. Stappers and J. Hennessey. Toward electronic napkins and beer mats: computer support for visual ideation skills. *Visual Representations and Interpretations*, pages 220–225, 1999. → pages 12
- [53] C. Stolte, D. Tang, and P. Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Trans. Visualization and Computer Graphics*, pages 52–65, 2002. → pages 19
- [54] G. Taubin. An accurate algorithm for rasterizing algebraic curves. In *Proc. ACM Symp. Solid Modeling and Applications*, pages 221–230, 1993. → pages 25
- [55] M. Thorne, D. Burke, and M. van de Panne. Motion doodles: an interface for sketching character motion. In *Proc. ACM SIGGRAPH*, pages 442–429, 2004. → pages vi, 13, 14, 16
- [56] E. Tufte. *The cognitive style of PowerPoint*. Graphics Press, 2003. → pages 12

- [57] H. Wickham and H. Hofmann. Product plots. *IEEE Trans. Visualization and Computer Graphics*, 17(12):2223–2230, 2011. → pages 6
- [58] N. Wilde and C. Lewis. Spreadsheet-based interactive graphics: from prototype to tool. In *Proc. SIGCHI Conference Human Factors in Computing Systems: Empowering People*, pages 153–160. ACM, 1990. → pages 18
- [59] L. Wilkinson. *The grammar of graphics*. Springer, 2005. → pages 6, 21
- [60] W. Willett, J. Heer, and M. Agrawala. Scented widgets: improving navigation cues with embedded visualizations. *IEEE Trans. Visualization and Computer Graphics*, pages 1129–1136, 2007. → pages 42
- [61] C. Yang. *Sketch-based modeling of parameterized objects*. PhD thesis, University of British Columbia, 2006. → pages 13
- [62] J. Young, T. Igarashi, and E. Sharlin. Puppet master: designing reactive character behavior by demonstration. In *Proc. ACM SIGGRAPH/Eurographics Symp. Computer Animation*, pages 183–191. Eurographics Association, 2008. → pages 14
- [63] D. Zongker and D. Salesin. On creating animated presentations. In *Proc. ACM SIGGRAPH/Eurographics Symp. Computer Animation*, pages 298–308. Eurographics Association, 2003. → pages 12