Interfaces for Creating Quantitative Conceptual Diagrams

by

Robin S. Stewart

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2008

Author . . . .
Department of Electrical Engineering and Computer Science
May 23, 2008

Certified by . . . .
David R. Karger
Professor of Electrical Engineering and Computer Science
on behalf of: mc schraefel
Research Affiliate, Computer Science and Artificial Intelligence Laboratory
Thesis Supervisor

Accepted by . . . .
Terry P. Orlando
Chairman, Department Committee on Graduate Students

Interfaces for Creating Quantitative Conceptual Diagrams

by

Robin S. Stewart

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2008, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

Modern chart-making, illustration, and mathematical tools poorly support the use of conceptual components in quantitative graphs such as Economics diagrams. The substantial time those tools require to achieve the desired results leads many people to sketch their graphs with pencil and paper instead of using a computer. In this thesis, I address the challenge of designing a software user interface that not only includes all features necessary to create a wide range of quantitative conceptual diagrams, but also is dramatically more efficient to use than existing programs. My design takes several important interaction techniques that previous applications used separately and comprehensively integrates them in order to create new, flexible capabilities. I have implemented this design as a desktop application called Graph Sketcher, and I present results of studies which show that my interface halves the time required to complete several common graph creation tasks. I also show that the 700 students, teachers, professionals, and hobbyists worldwide who choose to use Graph Sketcher in their everyday work find the interface intuitive, enjoyable, and empowering for generating many different types of graphs.

Thesis Supervisor: mc schraefel
Title: Research Affiliate, Computer Science and Artificial Intelligence Laboratory

Acknowledgments

When I first noticed, more than five years ago, that there was no good software for sketching Economics graphs, I never imagined that this small observation would eventually grow into a major software product and the basis for an entire masters thesis.

Over those five years, there are a lot of people to thank; I will name a few of them here. Lucie Schmidt drew beautiful Economics diagrams on the board on my first day at Williams College. Ted Smith forced me to run cross-country in high school, which led me to become a habitual jogger, which gave me time to dream about graph sketching interfaces as I ran through the hills of Williamstown. Duane Bailey taught me that the best way to improve code is to delete it; that garbled code is also called "job security"; and that software engineers should sign their work. Ashleigh Theberge showed me how to use multiple highlighters to understand scientific papers. Aaron Hillegass wrote the book *Cocoa Programming for Mac OS X*, from which I learned how to make Mac applications and decided that a graph sketching program might be fun to build. Karl Naden, Evan Miller, Katie Belmont, Leah Weintraub, Miriam Lawrence, and many other friends were there to help me test Graph Sketcher, play with it, and solve geometry problems during the formative — and extremely cold — January of 2004.

Emily Fertig convinced me to study abroad in New Zealand for a semester, which gave me the time to read lots of books about human-computer interaction (HCI). Kris Kirby agreed to advise my subsequent independent study on graph sketching interfaces, "strongly encouraged" me to add best-fit lines, and was a wonderful mentor over the years. Bill Lenhart taught an excellent four-student lecture class on computer graphics and helped me derive curved line algorithms. Several other professors who will go unnamed told me simply "yep, it's possible to solve that."

Regina Barzilay admitted me to MIT on the assumption that I would do natural language processing, and Victor Zue convinced me that I would survive. Brainstorming meetings with Michael Bernstein and Max Van Kleek helped me realize that I was more interested in interfaces than in probability theory. David Karger provided supplemental funding for my second year on the assumption that I would do research related to semi-structured information retrieval, and he has been everything I could ask for in an advisor. Sara Su and Jaakko Lehtinen helped me derive yet more curve algorithms for Graph Sketcher. And when I decided to drop everything else and write my thesis on graph sketching, mc schraefel stepped in with abundant enthusiasm, ideas, advice, and support, despite her physical location in the UK.

Throughout all this, Graph Sketcher's wonderful beta testers and many other users have continued to give me feedback, find bugs, and suggest further improvements. The software would never have come this far if they had not repeatedly convinced me that it was actually useful.

Finally, the less specific but more important thanks to all the people I love: mom, dad, sista, grandma "B," grandpa "Abe," crazy techno dancers, Haystack homies, and the Williams alums who have helped keep me sane over the past two years.

This material is based upon work supported under a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

Contents

List of Figures

Chapter 1

Introduction

1.1 Visual Presentation of Quantitative Concepts

Information graphics are useful both for analyzing data sets and for visually presenting quantitative ideas. The relational graph, which plots the relationship between two variables by making use of two spatial dimensions, first came into wide use in the late 1700's, most often with time as one of the variables [38]. Since then, teachers of economics, math, physics, chemistry, and other subjects have found the two-dimensional relational abstraction to be an excellent method not only for plotting data but also for describing quantitative concepts. For example, most introductory economics students are shown (and are often asked to draw) conceptual graphs that look something like Figure 1-1 [19]. Although most such diagrams are at least motivated by real data, they do not necessarily plot any data points.

Despite this rich history of conceptual graphing, modern chart creation tools make the explicit or implicit assumption that all charts are based around data sets. Not surprisingly, this assumption has resulted in chart-making interfaces which make it very difficult, if not impossible, to create purely conceptual graphs. Equally worrisome, these interfaces poorly support the large spectrum of graphs which include both data and conceptual components. For example, Figure 1-2, which was published in an article on climate policy, shows the actual carbon emissions during the past 50 years along with a range of emissions scenarios for the next 50 years [34]. The diagram conveys several interesting quantitative concepts: an extrapolation of past trends (the "current predicted path"); a goal scenario (a "flat path" that "avoids doubling"); and the notion of a "stabilization triangle" which represents the carbon that must not be emitted in order to achieve the flat path. Chart making programs easily plot the historical emissions data, but I will show that they break down when it comes to adding and modifying the conceptual components.

For many graphs, the best existing solution is to use complex, general-purpose illustration programs which require considerable time and expertise to learn and use. Since people have limited time, most do one of two things instead. First, they use a hand-drawn sketch if it is acceptable for the situation at hand. For example, most students in introductory economics courses draw graphs by hand on the problem

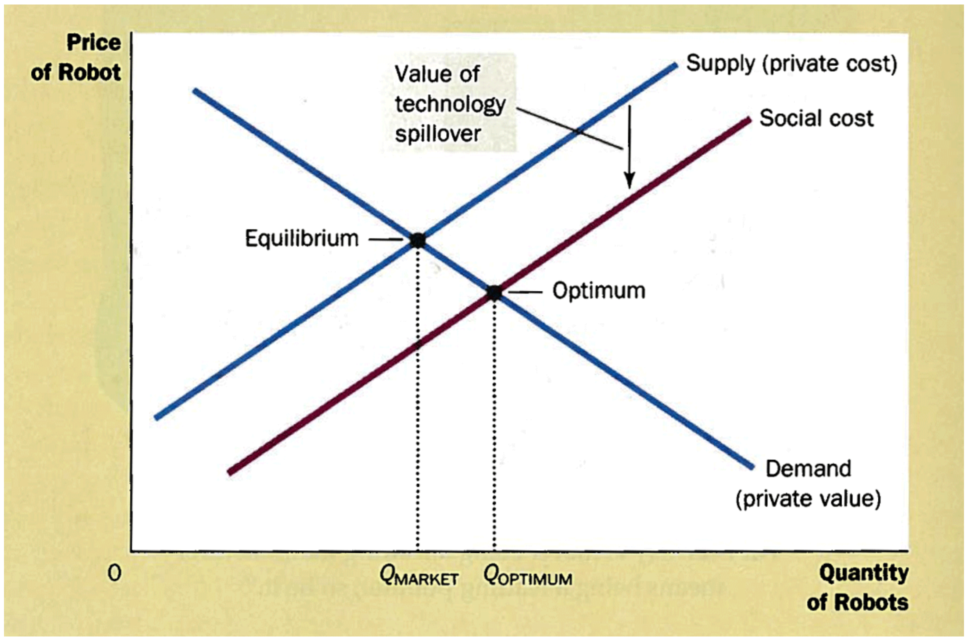


Figure 1-1: A typical diagram from an economics textbook [19] consists solely of conceptual lines and labels.

sets they turn in (Chapter 2.2.1). If a hand-sketched diagram is not sufficiently professional, transmittable, or precise, people might skip the graphic altogether and simply rely on verbal descriptions — which are comparatively easy to create with the highly refined word processing capabilities of modern computers. These verbal descriptions are useful, but the fact remains that many concepts are best described graphically — such as the "stabilization triangle" of Figure 1-2.

Because of the difficulty of creating professional-quality quantitative conceptual diagrams, the use of these diagrams is limited to textbooks and other professional publications. If these diagrams were substantially easier to create — as easy as verbal descriptions, for example — their use could be expanded into everyday mediums such as lecture notes, essays, problem sets, and blogs.

1.2 Contributions

This thesis addresses the challenge of designing a user interface that not only provides all features necessary to create graphs like Figures 1-1 and 1-2, but also is dramatically more efficient to use than existing programs. I show that neither current charting tools, illustration tools, nor mathematical tools satisfactorily support the creation of this class of information graphics. To improve this situation, I motivate and develop a graphing interface that treats conceptual components as data objects which can be created and modified visually in a manner that mimics sketching by hand. I have implemented this interface as a Mac OS X application called Graph Sketcher which

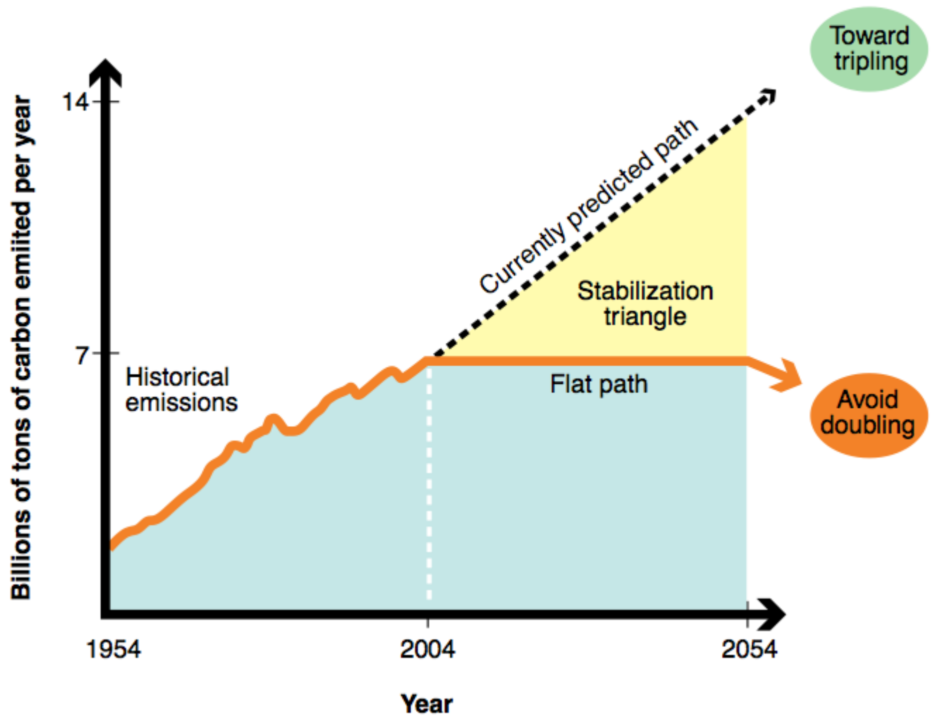


Figure 1-2: This graph from an article on climate policy [34] combines plotted data points with quantitative conceptual elements.

is being used by over 700 students, teachers, professionals, and hobbyists worldwide. Graph Sketcher has enabled these users to quickly create a wide range of quantitative conceptual diagrams which they otherwise perhaps never would have made.

I primarily used the domain of Economics to motivate and inform my design. There are several reasons behind this choice. First, conceptual graphs are very common in Economics textbooks, problem sets, and research publications. Second, these graphs use a wide range of diagrammatic components, including straight and curved lines, data sets, complex filled areas, and contextual labels. Third, I have personally taken several Economics classes, so I am familiar with the subject matter of these graphs as well as the experience of creating them (in class notes and problem sets). Last and perhaps most important, while quantitative conceptual diagrams are found in many fields, narrowing the scope of inquiry helps make the problem tractable and ensures that the design will be highly useful in at least one domain.

1.3 Organization of This Thesis

The rest of this thesis is organized as follows. In Chapter 2, I look at usage patterns and properties of quantitative conceptual diagrams in order to motivate an improved interface design. In Chapter 3, I overview the related research concerning graphical interfaces for chart making, information visualization, contextual annotations, geometric constraints, and pen-based input. In Chapter 4, I describe my interface approach in more detail and show that it is substantially more efficient than existing programs. In Chapter 5, I explore novel interaction techniques made possible by Graph Sketcher's design and I study its curve creation and manipulation interfaces in detail. In Chapter 6, I evaluate the success of the interface beyond economics diagrams by investigating how it is used longitudinally in the context of everyday work. Finally, in Chapter 7, I offer conclusions and directions for future work.

Chapter 2

Motivation

2.1 Quantitative Conceptual Diagrams

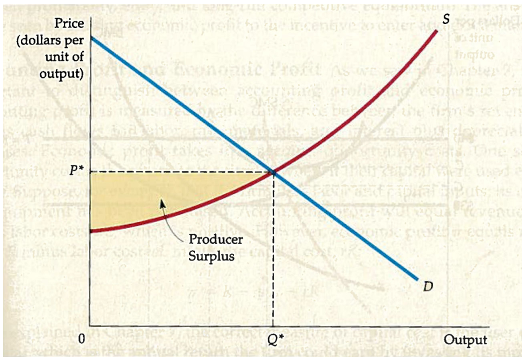
I consider a diagram to be *quantitative* if it involves a continuous scale, usually represented by an axis. The term is distinct from an *ordinal* set (where order matters but there is no scale) or a *nominal* set (simply an unordered collection). These can be combined; for example, a bar chart might show the quantitative height of a nominal set of people. Quantitative diagrams are also to be distinguished from *qualitative* diagrams, which might illustrate a concept with drawings of different phases, such as the growth of a caterpillar into a butterfly. A diagram can be both quantitative and qualitative, for example if the caterpillar pictures are arranged along a timeline scale.

In quantitative diagrams, I distinguish *conceptual* components from concrete *data sets*. The most common type of chart simply visualizes a set of data points, which might come from a scientific experiment, a sales database, thin air, etc. *Quantitative conceptual* components have quantitative meaning but are more complex or more abstract than a single data point. For example, a diagonal line on a 2D graph could be a quantitative conceptual object that represents an increasing trend over time. A shaded area could be a quantitative conceptual object that represents the total net gain over some time period. Diagrams often include both data sets and conceptual components, as seen in the carbon emissions diagram in Figure 1-2.

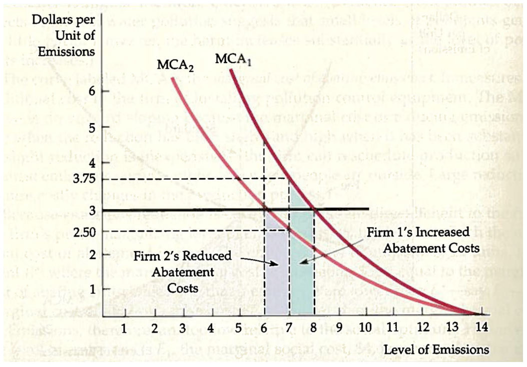
Quantitative conceptual diagrams are used to teach almost every quantitative discipline. They are especially common in economics, where they're used to visually explain a plethora of quantitative theories, such as the relationships between quantity and price of a good (see Figure 2-1). They're also used to describe concepts in chemistry, biology, physics, and engineering. Often, a plotted data set forms the core of the graph, and conceptual marks are overlaid to indicate the generalizable concepts.

2.1.1 Analysis of economics diagrams

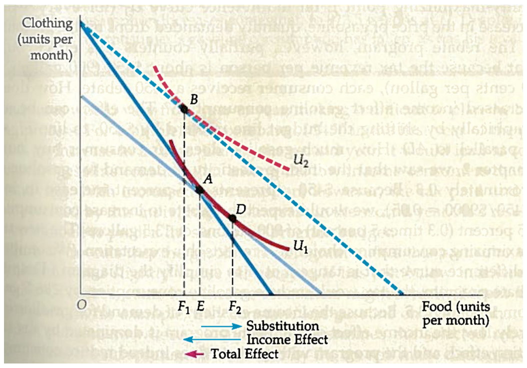
Before diving in to a thorough study of quantitative conceptual diagrams, it would be prudent to verify that they are indeed important and widely used in the world. To verify this in one domain, and to better understand what components these diagrams



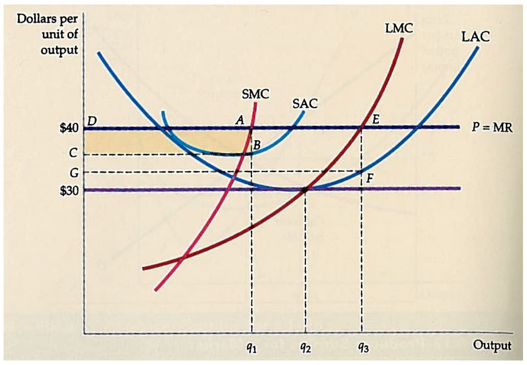
(a)



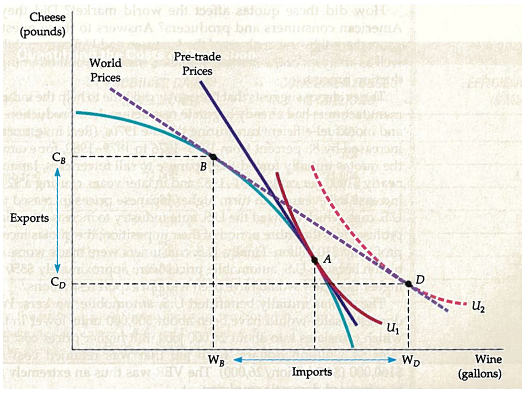
(b)



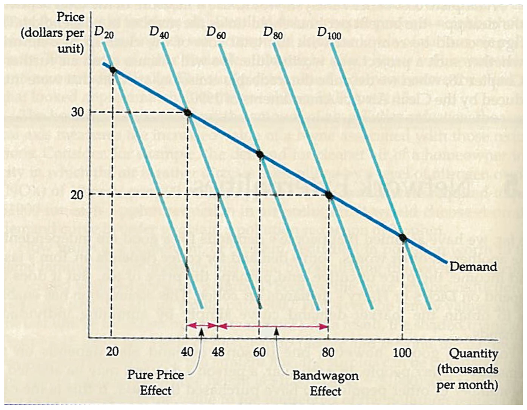
(c)



(d)



(e)



(f)

Figure 2-1: A sampling of quantitative conceptual economics diagrams from a Microeconomics textbook [25]. The goal of this thesis is to better support the creation of such diagrams.

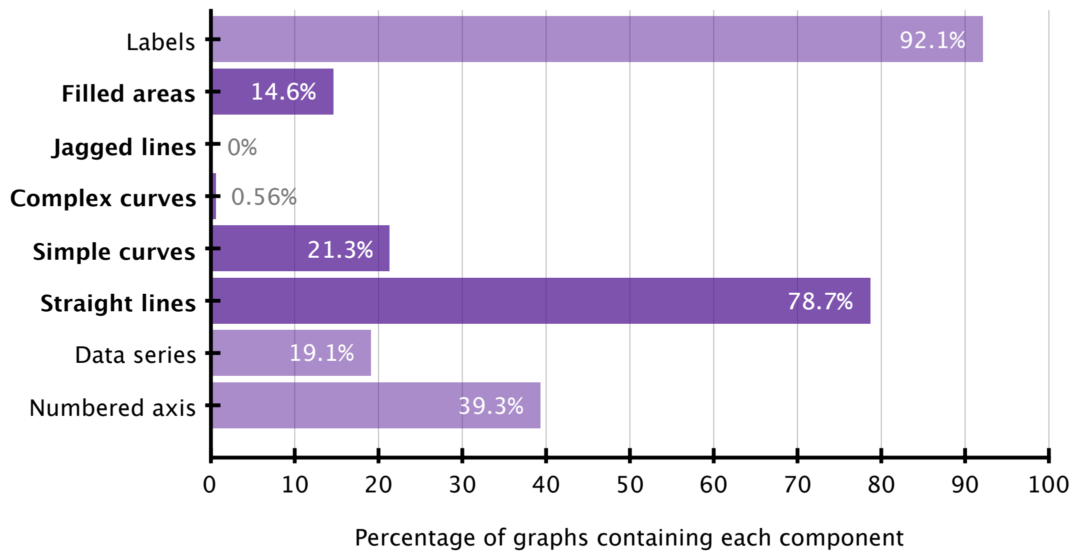


Figure 2-2: The prevalence of various components in 178 graphs from an introductory economics textbook [19]. The conceptual components are printed in bold.

typically contain, I analyzed all of the diagrams in a typical introductory economics textbook [19]. Out of 190 diagrams total, 178 (94%) were two-dimensional graphs with axes. The properties of these 178 graphs are summarized in Figure 2-2. For this analysis I did not count lines and labels found in the axes, because all of the graphs included both axes and axis labels.

Consistent with the definitions above, *conceptual components* are filled areas or lines that do not represent data series (the conceptual types are printed in bold in Figure 2-2). By this definition, fully 82% of the economics graphs contain at least one conceptual component, and in fact 80.9% of graphs have *only* conceptual components. By far the most common component is the straight line, which appears on 78.7% of graphs. Simple curves appear in 21.3% of graphs, and only one curve required more than one inflection point (curves are discussed in detail in Chapter 5.3). I found filled areas in 14.6% of graphs, representing concepts such as "consumer surplus" and "total cost." Finally, there were a significant number of data series plots in the textbook diagrams (19.1%), only two of which also contained conceptual components.

Importantly, 20.2% of the diagrams had numbered axes and solely contained conceptual components (such as Figures 2-1b and 2-1f). These are quantitative conceptual diagrams in the deepest sense, because the lines and filled areas refer to real, scaled quantitative units. In the 60.7% of diagrams which were conceptual and without a unit scale, the lines and filled areas are still quantitative insofar as their meaning depends on a generic scale. For example, the relative slope of different lines on the graph or relative areas of different regions is quantitatively meaningful.

This analysis shows that quantitative conceptual graphs are indeed a critical part of introductory economics: the standard textbook I analyzed contains diagrams on

23% of its pages and 82% of those diagrams have conceptual components. Thus, making it easier to create these diagrams would benefit at the very least the large number of people who take or teach introductory economics, and probably many more people in other quantitative fields such as physics, chemistry, biology, and math. However, it is still not clear whether it is even *possible* to make it significantly easier to create these diagrams, so in the next section I look for indications that there is indeed room for substantial improvement.

2.2 Computer-Generated vs. Drawing By Hand

The second motivation to study quantitative conceptual diagrams is the observation that they are commonly drawn by hand rather than using computer software. On the one hand, if the diagrams have to look professional, such as in textbooks, lecture slides, and scientific papers, then people do use computers to make them. But when the diagrams are used in less formal places such as class notes and problem sets, they are commonly drawn by hand. This observation caught my attention because one of the promises of computer technology is to be more efficient than pencil and paper. In many classrooms, laptops are already replacing paper notebooks for taking notes; so why not for sketching graphs? If people are choosing to make graphs by hand, then clearly the existing computer interfaces are insufficient.

2.2.1 Survey of economics professors

To verify my initial, anecdotal observations that most students draw their economics diagrams by hand rather than using a computer, I emailed 128 economics professors at Stanford University, Tufts University, University of Wisconsin-Madison, and University of Pennsylvania. I asked a single question: "Approximately what percentage of your students, if any, turn in computer-generated diagrams on their problem sets instead of drawing them by hand?" I received 24 responses (a 19% response rate), five of which stated that their classes did not require diagrams on problem sets. The remaining 19 answers ranged from zero to 100% of students turning in computer-generated diagrams. The mean estimate was 30.5%, and the median was 20%. A histogram of all 19 estimates appears in Figure 2-3.

The survey results are strongly skewed towards drawing economics diagrams by hand, particularly in introductory classes. Only one professor reported that 100% of students use the computer, and the majority of professors estimated that number at less than 30%. Some professors expressed pity for their students making computer diagrams: "In my most recent class (a graduate class) probably more than half. Not sure how they do it - some combination of powerpoint, latex [typesetting software], other software I don't even know about..." Another remarked, "Frankly, I find producing computer graphics much too time consuming and fiddly myself, so I do not expect them from students." By contrast, most college students write their short essays, reading responses, etc. using the computer, presumably because it is faster to type and easier to read digital text than hand writing. Indeed, one economics professor

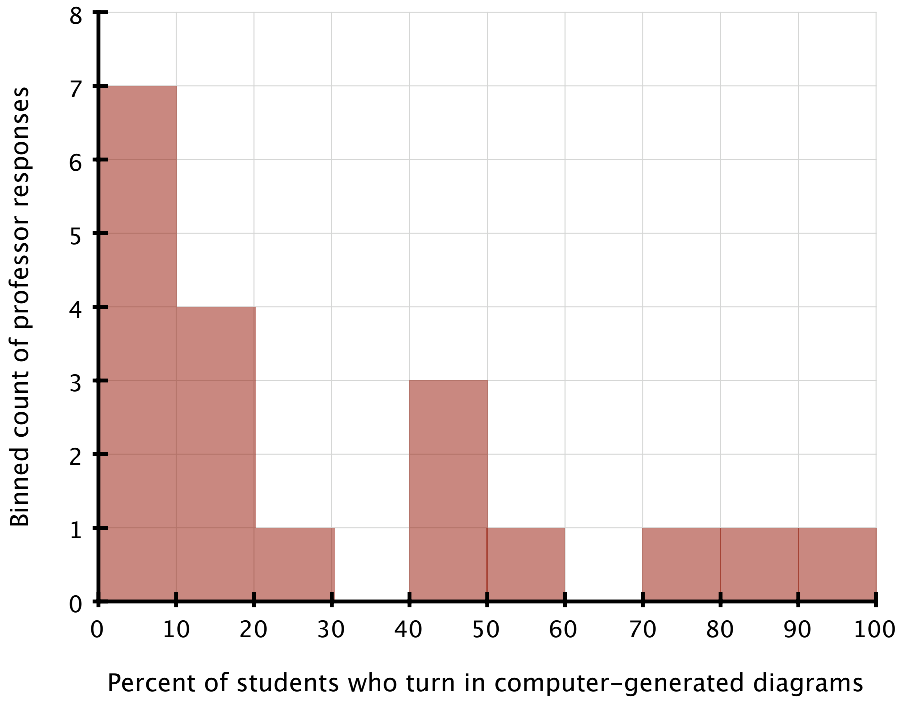


Figure 2-3: Histogram summarizing 19 Economics professors' estimates of the percentage of their students who turn in computer-generated diagrams on problem sets.

in the survey remarked that half of the students who wrote up their problem sets with a word processor still drew in the graphs by hand (I, too, did this in my introductory economics course).

These responses suggest that most students have not found a satisfactory way to create conceptual economics diagrams using existing software interfaces. One explanation for this could be that satisfactory software does exist but most students have just not found it. This explanation is unlikely, because no one mentioned any tools that are not widely available, nor did my extensive research of existing tools find any eligible programs. Instead, I conclude that the existing software is only satisfactory for a subset of students. Perhaps these students are expert users of existing charting or illustration software, or perhaps their professors strongly encourage the use of computer-generated diagrams even if it is very time-consuming. It seems likely that a much larger proportion of students would voluntarily make diagrams by computer if it saved them time — or even if the time penalty for getting professional-looking results was merely not as large. With this in mind, I turn to analyzing *why* existing software interfaces do not satisfactorily support the creation of these diagrams.

2.3 Properties of Quantitative Conceptual Diagrams

In this section I look at two quantitative conceptual diagrams in detail (Figures 2-4 and 2-5) to better understand their requirements.

Figure 2-4 is another typical diagram from an economics textbook [25]. It is not based on a data set, so it would be difficult to create with typical charting tools. The axes do not have explicit units, so it makes little sense to define the lines as equations (as is done in mathematical software) or as a set of numbers (as is done in spreadsheet and database software [5]). Instead, the quickest way to define the lines is to input them visually — the method used in typical illustration programs (e.g. Adobe Illustrator [14]). The labels and the yellow filled area are also most easily defined using visual input. Thus, software that supports this type of diagram should allow visual input of lines, labels, and filled areas.

However, the diagram has more structure than a generic visual collection of lines and labels: the various components are strongly related and connected to each other. For example, the dashed vertical and horizontal lines start at curved line intersections and end at the axes; labels like "MBC^1" appear just at the ends of lines; and the yellow filled area is in the region between lines. Thus, software that supports this type of diagram should help establish relationships between components, and help to maintain them when various aspects of the diagram are modified.

The diagram from climate policy that we saw in the introduction (and is reprinted below as Figure 2-5) contains many of the same components as Figure 2-4 — i.e. lines, labels, filled areas, axes — but in a more numeric context. The axes have units attached, and regular data points (the "historical emissions") are plotted accordingly alongside conceptual components (such as the "flat path"). The positions of the plotted data and conceptual components alike are given meaning by the units on the axes. For example, the "flat path" is not merely a line on a page but a line from (2004, 7 billion) to (2054, 7 billion). The shaded areas are similarly bounded by these quantitative measures. And the axis labels (such as "1954") obviously need to appear at the quantitative positions they correspond to. Thus, software that supports this type of diagram should have the ability to anchor all graphical components (not just data points) in the unit system given by the axes.

In summary, I claim that three capabilities in particular are necessary to fully support quantitative conceptual diagrams.

1. Visual input of lines, labels, and filled areas.

2. Detection and maintenance of geometric relationships between components.

3. Positioning of all components (not just data points) relative to axis units.

Of course, there are many additional needs, such as axis creation and manipulation, data set storage and viewing, copy/paste and undo functionality, and the formatting of text, lines, and fills. However, these are widely supported by most software (including my implementation) and thus fall outside the focus of this thesis.

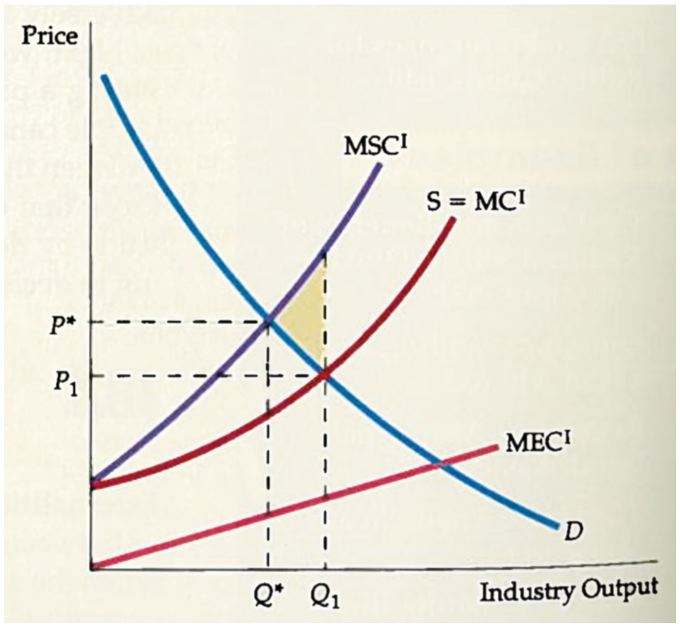


Figure 2-4: This economics diagram [25] demonstrates (a) the need for visual input of lines, labels, and filled areas, and (b) the presence of geometric relationships between the various components.

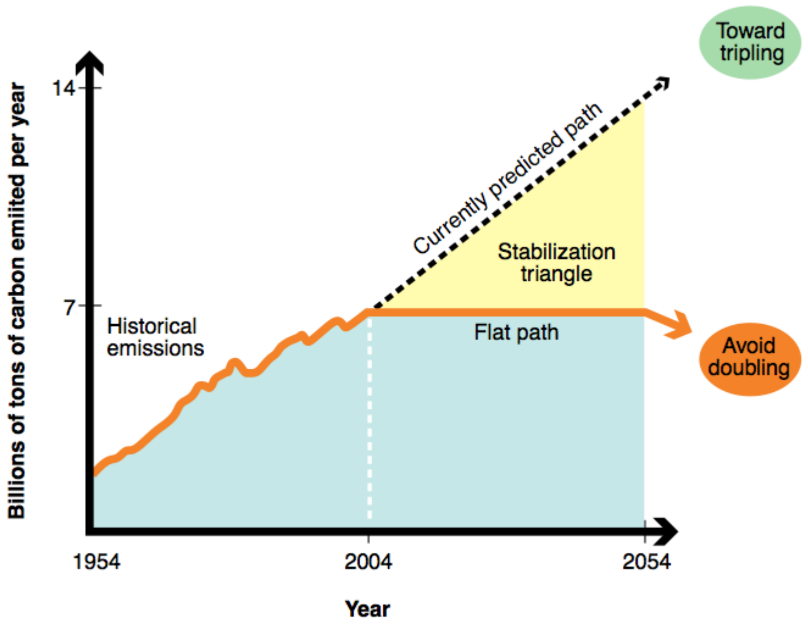


Figure 2-5: This climate policy diagram [34] demonstrates that both conceptual components and data points are positioned relative to the quantitative unit system displayed by the axes.

2.4 Shortcomings of Existing Tools

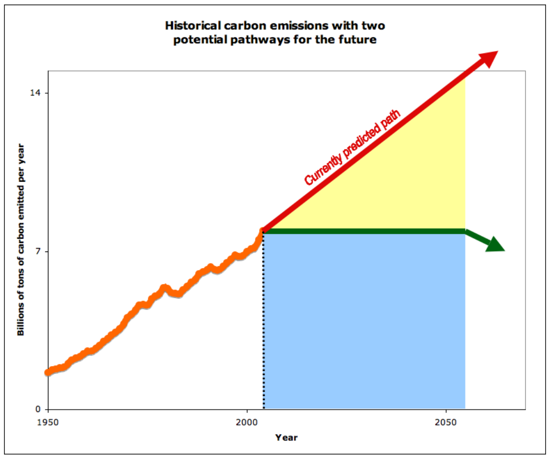
Existing tools often support one or two of the capabilities outlined in Section 2.3, but not all three. In particular, there is a disconnect between illustration programs which support visual (cursor-based) input of graphical components, and math- or data-focused tools which position components relative to a quantitative scale but do not support visual input. A few of the tools in both categories also detect and maintain geometric relationships to varying degrees.

Because of this tool dichotomy, the traditional method of creating a diagram like Figure 2-5 is to graph the data in a quantitative visualization program and then export the result to an illustration program to add the conceptual components. This pipelined approach suffers from the problem that if any data is added or modified, or the axis ranges need to be changed — in other words, if the underlying visualization has to be changed in any way — the export/import pipeline has to be started over again.

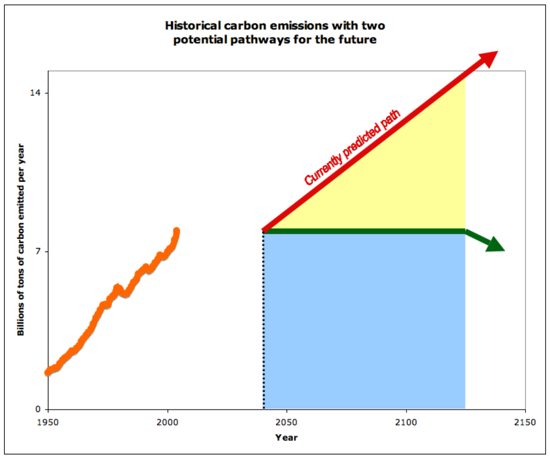
Microsoft Excel [5] solves the pipeline problem by providing both data visualization and illustration tools within the same program. However, it fails to actually integrate these two sets of functionality. In particular, illustration components added to a chart are not treated like data — they simply float on the chart surface independently of the axes and data sets. This means that when the axis ranges are changed, the annotation objects stay where they are, thus losing their meaning (Figure 2-6). In addition, it is difficult or impossible to establish relationships between illustration objects and data points, such as connecting a line to the end of a data set or filling an area bordered by a data series (both of which are needed for Figure 2-5).

2.5 Proposed Improvements

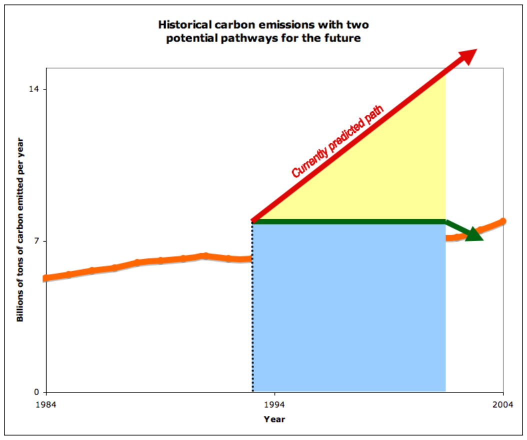
In this chapter, I have described several indicators which suggest that quantitative conceptual diagrams are important yet poorly supported by existing software. I have also put forth the hypothesis that three interface capabilities in particular are most relevant: visual input, positioning relative to axis units, and detection and maintenance of geometric relationships. This leads me to propose that an interface which integrates all three of these capabilities would be markedly more efficient for creating and modifying quantitative conceptual diagrams. In Chapter 3, I describe related research which is relevant to the task of designing such an interface. In Chapter 4, I describe my implementation and show that it indeed provides a substantial improvement over existing programs.



(a) Independently floating conceptual components work so long as the visualization does not change.



(b) If we adjust the x-axis to look further into the future, the data points get adjusted but the conceptual components don't.



(c) If we zoom in on a 20-year interval, the conceptual objects become completely meaningless.

Figure 2-6: An example of what can happen when conceptual components are not positioned relative to axis units.

Chapter 3

Related Work

To my knowledge, no prior work has specifically addressed the design of interfaces for creating quantitative conceptual diagrams. However, the interface techniques I use draw on a large body of related research.

3.1 Chart Creation Interfaces

The most commonly used chart-making programs include Microsoft Excel [5], Adobe Illustrator [14], OpenOffice.org [23], and Apple iWork [15]. Excel, OpenOffice, and iWork are designed to allow novice users to easily create charts that visualize existing data sets. They also include a wide range of visual annotation tools for adding lines, labels, clip art, and other objects to graphs. Illustrator is more flexible but requires a higher level of expertise to use. Because it is a general-purpose illustration program, it can be used to create any imaginable information graphic, but it does not provide many features to make chart creation more efficient. I already discussed some of the limitations of these programs in Chapter 2.4, and I will analyze them more quantitatively in Chapter 4.3.

Several research prototypes have explored alternative interfaces for creating business charts (mostly bar charts, x-y plots, and pie charts [40]). None of these prototypes can create conceptual Economics diagrams such as Figure 2-4, but the systems are still interesting as related work. I describe three examples here.

"Gold" is a system for creating business charts by demonstration [22]. The user puts data in a spreadsheet, and then starts to draw a chart using vector drawing tools such as rectangles. The system uses heuristics to guess which properties of the graphical objects map to which data fields, and automatically completes the chart by cleaning up the already-drawn rectangles and adding the rest according to the data. Graph Sketcher's approach is different: all data is immediately plotted on the 2D surface, and the user can then simply choose how they want it displayed. One advantage of this approach is that a default display can be produced without any user input beyond importing the data. It also completely avoids the difficulty Gold faces in distinguishing between new data series and one-time annotations.

Mackinlay developed a system for automatically generating information graphics from a given set of database relations [18]. It tries to optimize the presentation for both graphical standards and human perceptual ability. A disadvantage is that it requires the underlying data to be formally specified as quantitative, ordinal, or nominal (unordered set). This approach could be used in Graph Sketcher to provide an initial chart design, but the user still needs to be able to customize the design and add conceptual annotations.

SageBrush [30] in a sense brings both of the previous approaches together. The user specifies a graph design by dragging "graphemes" (e.g. points and bars) into the workspace and attaching data columns to grapheme properties such as position, color, and width. A single grapheme represents an entire data series, so the output graphic looks much different from the specification graphic. Anything that is left unspecified gets determined by the automatic presentation engine. SageBrush also lets the user easily reuse graphic designs by dragging new data columns into the grapheme property widgets. However, once again it does not support conceptual components or visual manipulation of the data.

3.2 Interactive Information Visualization

Some aspects of Graph Sketcher's interface were pioneered by interactive information visualization tools. For example, many of those tools (such as Spotfire [37]) let the user visually select data by clicking or dragging out a rectangular region. The selected data can sometimes be zoomed in on or have its formatting changed, but cannot be repositioned since the data is assumed to be fixed. Fathom [27], a visual tool for teaching statistics, includes a richer than usual set of direct-manipulation graphing abilities, such as the ability to drag a data set onto an axis and rescale axes by dragging them. It also lets the user directly manipulate certain lines while it updates the displayed line function. However, the lines must be straight and "infinite" in length and can only be adjusted from one end at a time. More complex functions can be adjusted in real time only via sliders that control variables in the function.

These information visualization tools are generally designed for data exploration and analysis. By contrast, Graph Sketcher is primarily designed for presenting data and concepts that have already been analyzed — though it can also be used to explore and analyze simple data sets.

3.3 Visual Annotations

As discussed in Chapter 2.4, Excel and other charting programs generally put user-defined visual annotations on a layer that floats independently above the graph surface. The advantage of this approach is that the visual annotation tools can be used across all types of underlying content (plots, pie charts, images, etc.). Indeed, ScreenCrayons [6] lets users annotate any window appearing in the operating system. The disadvantage with this approach is that the annotations quickly become useless if the underlying document is dynamic, for example if the text size is changed, the columns are resized, or of course, if the content is re-written.

The "sense.us" system for collaborative information visualization by Heer et al. [12] ties annotations to specific states of the visualization such that they only appear when the visualization has been drilled down accordingly. However, given a drill-down state, the annotations are still positioned independently of the chart. Because of this, if new census data were to be added, the visualization would shift the old data to the left and thereby cause old annotations to become misaligned. Also, if viewers were allowed to interact with the visualization more flexibly — for example, rescaling the axes to zoom in on arbitrary time windows of interest — the screen-based annotations would quickly lose their meaning (as was demonstrated in Figure 2-6).

WaveMetrics' Igor Pro [39] takes the first small step towards solving the visualization-rescaling problem by letting users anchor text notes to individual data points. But it does not address the problem for lines, shaded areas, or even labels that are not attached to specific data points.

Substantial work has been done to improve the situation for annotations on text documents. Golovchinsky and Denoue [10] anchored hand-drawn annotations to character positions so that each annotation was dynamically rendered at a size and location relative to the bounding box of the characters of interest. This allows the font size to be changed or the columns to be re-flowed while still maintaining the semantics of the annotation marks. Our interface is the first we know of to comprehensively provide this capability for information graphics.

3.4 Snapping and Constraint-based Interfaces

Earlier work in constraint-based interfaces has explored the idea of snapping and anchoring visual objects to each other. The basic idea of snapping to previously-drawn points goes back to the pioneering work of Ivan Sutherland's "Sketchpad" [36]. Bier & Stone [3] developed "snap-dragging" as a method for defining constraints among objects and created a system designed to resemble drawing with compass and straightedge. Gleicher and Witkin [8, 9] took the next step with Briar, an ingenious drawing program which used snapping to define a wide range of constraints, including relative angles, distances, and co-location. The fundamental idea is that the set of circumstances that cause a snap also inform the type of constraint to establish. For example, if a point is snapped along an existing line, the constraint to establish is that the point should be coincident with that line. Briar also includes "alignment objects" which allow users to set up constraints such as a constant distance from a point or a particular orientation in space.

Most of these ideas have not migrated into commercial illustration software, perhaps because geometric constraints are not useful to most graphic designers. However, they have been adopted in applications where geometric constraints play a key role, such as computer-aided design and simulation. The constraint-based approach has also been applied to interactively laying out graphs of the data-structure and org-chart variety [31]; OmniGraffle [11] further includes the notion of "connector magnets" for maintaining lines between boxes.

Graph Sketcher is the first to extend these snapping constraint-based techniques out of the physical 2D or 3D space and into the more abstract realm of relational graphs that map arbitrary quantitative units to each dimension. (It's interesting to note that a similar progression occurred with quantitative information graphics themselves, which began as cartographic maps, progressed to time series plots, and eventually introduced relations between non-spatial/temporal variables.) Graph Sketcher also goes beyond simple geometric constraints with tools for making annotations based on data — such as shaded areas that automatically follow curved lines and data series.

3.5 Pen-based Interfaces

An arguably more "natural" way to sketch graphs is with a pen interface, which strengthens the metaphor of drawing on paper. On the other hand, a pen is a poor metaphor for directly manipulating already-drawn objects. Baudel [2], Michalik et al. [20], and Mohammad and Nishida [21] have explored pen-based curve manipulation via the addition of new strokes. Igarashi et al. defined "interactive beautification" [13], which automatically applies constraints to each pen stroke drawn on a page, helping to maintain connectedness, parallelism, alignment, and other geometric properties.

Most pen-based interfaces have focused on recognizing gestures (via machine learning) so that the pen strokes can be treated as objects with greater meaning. For example, Ouyang and Davis [24] automatically convert strokes into chemical diagrams. Lower-level stroke recognition techniques include those developed by Sezgin, Stahovich, and Davis [32, 35] and Agar and Novins [1]. Pen interface research has also focused on removing tool modes in favor of automatically interpreting the intention of a gesture. The downside of fewer modes is a longer learning curve, a greater potential for ambiguity, or both. In Graph Sketcher, the decision to use four modes instead of one (see Chapter 4.1) was motivated by an early user study.

The closest I have seen to a pen-based interface for creating relational graphs is MathPad2 [16], a tablet system for exploring math and physics concepts. A simple graph can be created via a suitably angled pen stroke, and the axes can be configured by hand-writing new minimum or maximum values. However, this system does not support conceptual annotations — it can only make simple function graphs.

Graph Sketcher's interface is applicable to both pen interfaces and traditional pointing devices. To date, however, it has only been tested with mice and trackpads. Expanding this research to pen or multi-touch interfaces remains future work.

Chapter 4

Realization

The core of Graph Sketcher's interface is not a fundamentally new technology but rather a novel synthesis of many of the techniques described in the previous chapters. Just as Briar [8, 9] integrated snapping with constraints to create a substantially more usable interface, Graph Sketcher integrates the three capabilities that were described in Chapter 2.3 as the most important for creating quantitative conceptual diagrams. The first, visual input, is common in illustration and computer-aided design (CAD) software. The second, positioning relative to axis units, is a fundamental part of information visualization and is sometimes even combined with visual input in geographic information systems (GIS). The third, snapping to geometric relationships, is used in CAD, physical simulation, and visual layout software.

The intention of comprehensively integrating these capabilities — which no software has done before — is to design a graph-making interface that is both intuitive and maximally efficient for the user. In this chapter I describe the resulting interface and its implementation. I then perform task analyses across Graph Sketcher and three preexisting tools to show that Graph Sketcher's interface is approximately twice as efficient as the competition.

4.1 Graph Sketcher's Interface

Graph Sketcher uses four tools to support creation and manipulation of graph components (Figure 4-1). The first tool ("Modify") allows the user to select and directly manipulate all objects on the graph. The latter three tools enable quick visual creation of each basic component type: lines, filled areas, and text labels.

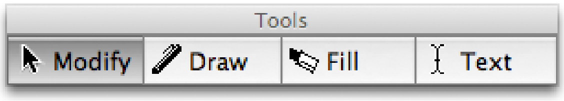
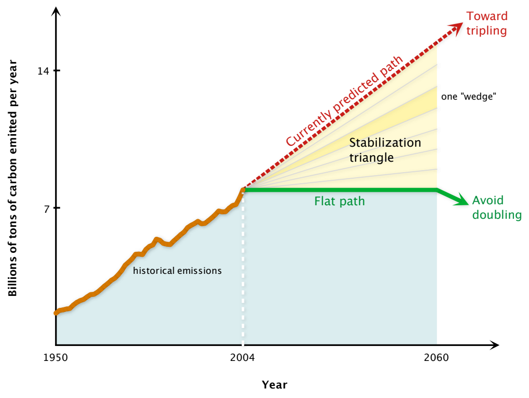
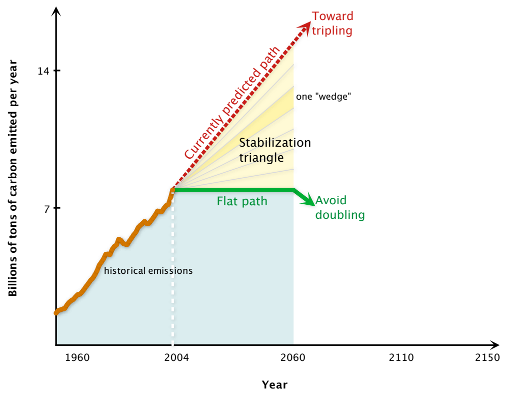


Figure 4-1: Graph Sketcher's tools for creating and modifying graph components.



(a) 1950–2060



(b) 1950–2150

Figure 4-2: In Graph Sketcher, all visual components are anchored to the axis scale, just like data. So when an axis range is updated (here to look further into the future), all components are adjusted accordingly.

4.1.1 Visual input of quantitative conceptual objects

Graph Sketcher's visual input interface is much like a typical illustration program, but the visually-input objects are treated like data in plotting software. With the appropriate tool selected, users can create lines by dragging across the graph surface, filled areas by clicking on each corner of the area in turn, and text labels by clicking at the cursor location. But unlike illustration programs, in Graph Sketcher all of these components are positioned relative to the unit scale of the axes. This means that when the axis ranges are changed, the conceptual lines, fills, and labels all shift accordingly (Figure 4-2).

4.1.2 Precise positioning via snapping

To make it possible to position these quantitative components quickly and precisely, all of the tools *snap* to various visual and geometric aspects of the graphic [3, 9]. Currently, components snap to existing:

- points
- lines
- line intersections
- axes
- grid lines

and lines also snap to horizontal, vertical, and 45 degrees. These snap-constraints are a subset of those supported in research prototypes such as Briar [8, 9] and

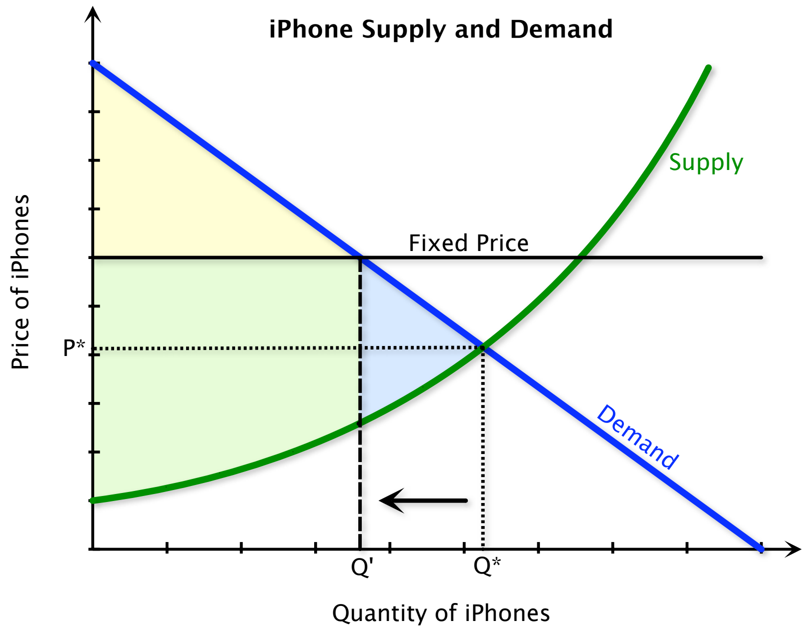


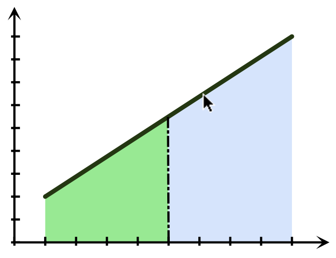
Figure 4-3: Graph Sketcher's snapping features make it much faster to create this economics diagram.

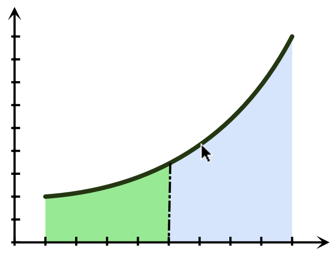
interactive beautification [13], which were chosen because they seemed to be the easiest constraints for users to understand.

These snapping behaviors make it more efficient to position nearly every element in Figure 4-3. Line endpoints snap to the axis, and even to individual tick marks. One end of each dotted line is snapped to an intersection, while the other end is snapped to horizontal/vertical and an axis. The corners and edges of filled areas are snapped to lines, intersections, and axes. And labels are snapped to lines, including the "Demand" label which was automatically rotated as a result. These snapping behaviors occur both as the annotations are being created (using the draw, fill and text tools) and as they are being adjusted (using the modify tool). Also, the snaps are displayed in real time so the user can see whether a snap will occur before lifting the mouse button.

4.1.3 Maintaining geometric relationships

As in Briar [8, 9], most snapped objects also maintain their snap-constraints when the object they were snapped to is moved. In Figure 4-4, for instance, the user manipulated only the curvature of the main line, while the other annotations stayed attached. Currently, only one constraint per position is maintained during manipulations, despite the fact that many constraints might apply to a given snap. For example, a line cannot be constrained during a manipulation as both vertical and attached to another line (in this case, only attachment to the line will be


(a)


(b)

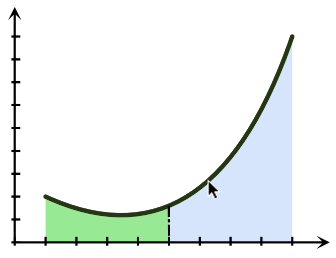

(c)

Figure 4-4: Maintaining geometric relationships while a line is manipulated.

maintained). However, it is easy to regain the extra constraints if necessary by simply snapping the annotations back into position. Automatically maintaining multiple constraints during manipulations would require the use of a much more complex constraint-solving engine such as the one described in [8].

4.1.4 Other functionality

For the program to be useful in realistic settings, Graph Sketcher also includes a range of supporting functionality such as axis creation, grids, grouping, locking, data set management, copy/paste, undo, and formatting of color, size, font, transparency, shadow, etc. of graph elements. Such functionality is widely supported and thus does not need to be described here in detail. However, these features are crucial if the program is to be voluntarily used in realistic settings outside the lab. Since one of the goals of this thesis is to evaluate the interface design in realistic settings, implementation of these supporting features along with the novel features is vital.

4.2 Implementation

In this section I describe the implementation details of Graph Sketcher's core functionality: displaying and manipulating visual objects that are positioned relative to the axis units.

4.2.1 Bidirectional mapping

Since all annotation types are anchored to the visualization, the heart of my implementation is a two-way mapping between data coordinates and screen/pixel coordinates. All information visualization systems map from data to the screen, but few map from the screen back to the data. This operation is more common in geographic systems which allow the user to visually add new roads and other items. Data coordinates are used in the data model, while screen units are crucial for human interactions such as selecting and dragging. In Graph Sketcher, the following components use data coordinates:

- axis ranges
- positions of points, lines, fills, and text labels

while the following are maintained in screen units:

- cursor "sensitivity" value for selecting and snapping
- line widths, font sizes, and styles
- amount of white space
- positions of axis titles.

All graphics are strictly vector-based. The diagram image itself is produced in the usual way by mapping from data to screen coordinates. Simple visual data entry works by mapping in the reverse direction. The positions of lines and fills are defined by their endpoints and corner locations, respectively.

4.2.2 Direct manipulation

Most direct-manipulation operations are implemented by the following sequence:

1. convert all relevant positions to screen coordinates
2. run the direct-manipulation algorithm
3. convert the results back to data coordinates for the data model

The "direct-manipulation algorithms" range from detecting that the cursor is near a line, to snapping a line to an intersection, to dragging a group of objects across the graph surface. It is essential that these direct-manipulation algorithms are performed in screen coordinates because they depend on parameters (such as the cursor sensitivity distance) which are defined in screen units. In order to show the manipulation in real time, step 3 is usually executed continually so that all aspects of the visualization are kept up to date. Step 1 is only executed once for a given manipulation, both for efficiency and to avoid the escalating margins of error that would result from repeatedly composing the inverse mappings.

I use a variation of this sequence to support various forms of direct-manipulation of the axes to modify the range of the graph. Step 1 is executed once to establish the connection between the initial axis range and the initial screen location of the cursor. Step 3 does not use the standard mapping function, but rather calculates new values for the axis range which are sent to the model and visualized in real time.

4.3 Verification: Comparative Task Analysis

To more quantitatively motivate Graph Sketcher's interface for creating and modifying conceptual diagrams, I performed a task analysis across several existing software programs. The other programs were chosen because they have a wide enough feature set to support a range of quantitative graphs and are designed for novice to moderately advanced users. Besides Graph Sketcher, I analyzed: Excel (part of Microsoft Office Mac 2008), Numbers (part of Apple's iWork 2008 suite), and Adobe Illustrator (in Adobe's Creative Suite 3). The tasks were chosen to focus on the types of graphs which I claim these existing programs' interfaces have neglected. Task 1 requires creating a simple economics diagram (Figure 4-6). Task 2 requires adjusting the graph range to make space for more items (Figure 4-8).

The analysis measures the number of "steps" required to carry out each task, to estimate the time needed by a user who is skilled at each interface. I approximate the keystroke-level model [4] by counting a step as any action that requires mouse targeting. If the target size is very small (within a few pixels) then I count the targeting action as two steps to approximate the extra time needed. (Fitt's law predicts that pointing time increases logarithmically as the target size decreases; a target size of 2 pixels unsnapped vs. 12 pixels snapped — the default in Graph Sketcher — predicts an increase of approximately a factor of 1.8.) So to draw a straight line with automatically-snapped endpoints in Graph Sketcher just requires positioning the mouse at each endpoint in turn (a total of 2 steps), but if the endpoints do not snap then each target also requires fine positioning (for a total of 4 steps). In addition, I approximate short typing sequences (such as typing the word "demand" or the axis maximum "180") as two steps.

For each program I carefully tried to find the most efficient sequence of steps that did not use keyboard shortcuts or other advanced techniques that beginners would not be familiar with. Thus it is likely that slightly faster sequences are possible for expert users, and perhaps for other charting and illustration programs that were not analyzed. However, as we will see, the disparity between Graph Sketcher and the other programs is far greater than the margin of error. In addition, I did not make any effort to format the graphs because formatting interfaces are very similar between programs and were not the focus of this investigation. Instead I simply used the default colors, sizes, fonts, etc. In a few cases the defaults were vastly too big, so I adjusted the size but did not count these adjustments in the task analysis.

4.3.1 Diagram creation

The first task is to create a simple economics diagram with numbered axes, four lines, three labels, and a filled area. The output of each program is displayed in Figure 4-6, and the analysis results are in Figure 4-5. These results show that Graph Sketcher's interface features substantially reduce the time it takes to set up every aspect of this diagram. The axis ranges can be set most quickly in Graph Sketcher because they can be typed directly into the graph, rather than requiring repeated traversal of dialog boxes or inspector panels. Lines can be drawn most quickly primarily because Graph

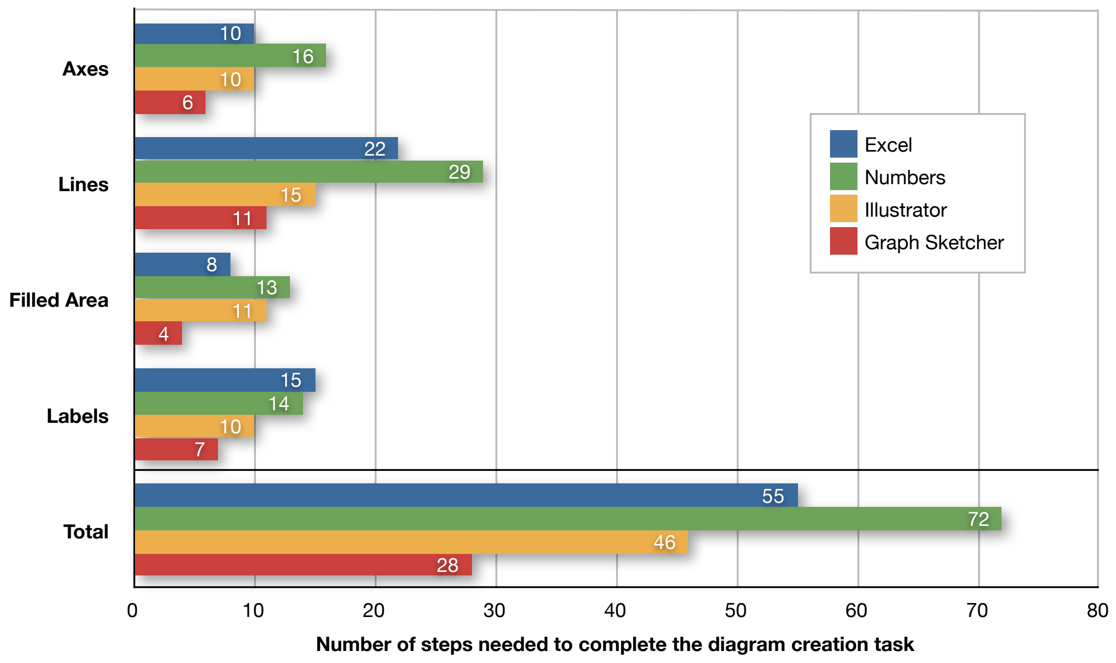


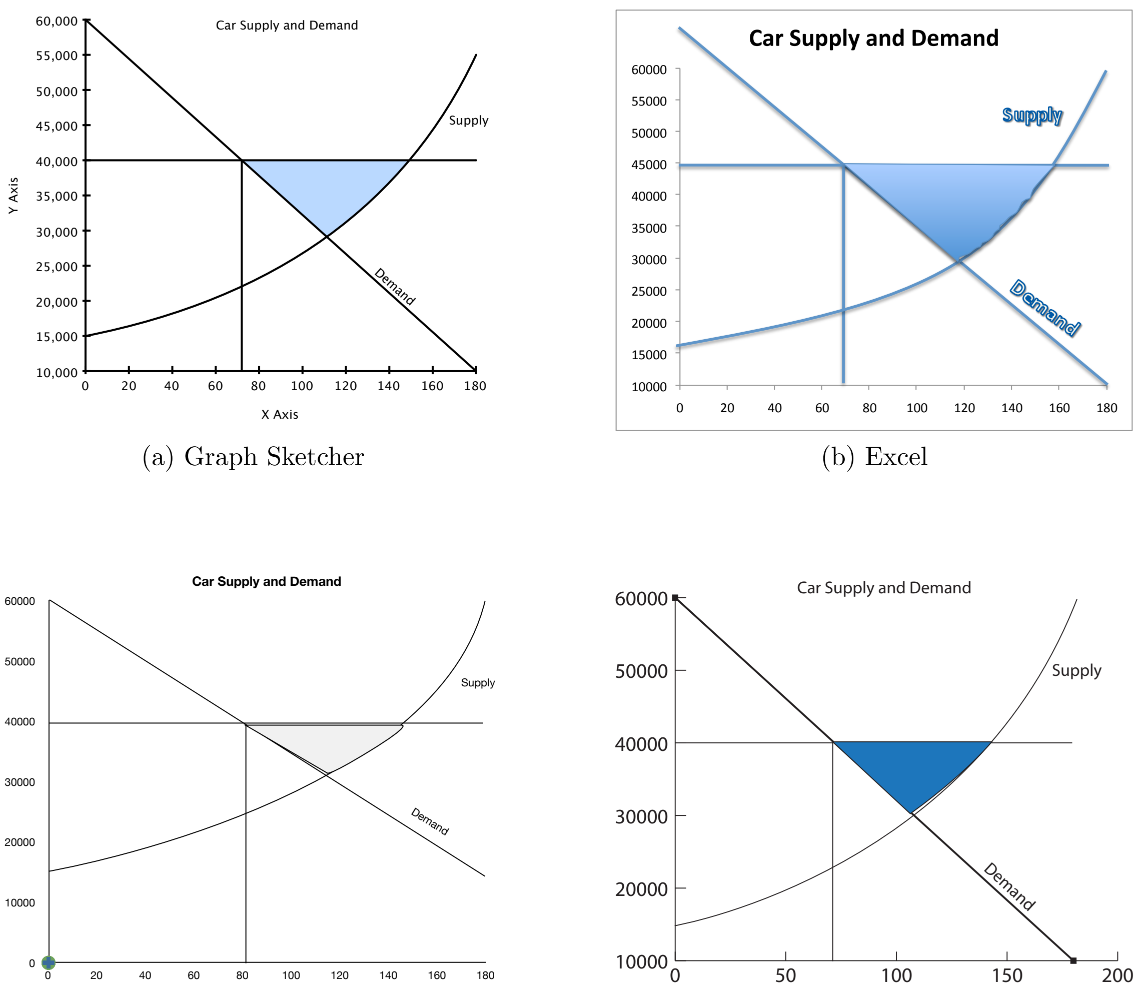
Figure 4-5: Results of the diagram creation task analysis.

Sketcher's geometric snapping features eliminate fine positioning steps. Creating the filled area is dramatically faster in Graph Sketcher because it snaps the cursor to each corner and automatically follows the curved edge (as far as I could determine, none of the other programs I analyzed let the user create filled areas whose edges are defined by existing objects). Finally, text labels were created most quickly in Graph Sketcher primarily because it automatically rotates the "Demand" label.

The results of the diagram creation task primarily illustrate the substantial time savings that can be gained by using snapping and relative-positioning features. These features could theoretically be adopted in future versions of any of the programs I analyzed, independently of whether they position visual objects relative to axis units.

4.3.2 Axis range adjustment

The second task is to simply adjust the x-axis range to allow more space for other hypothetical lines, labels, data, etc. (Figure 4-8). The results (Figure 4-7) demonstrate the utility of Graph Sketcher's treatment of all graph objects as data anchored in axis units. In Graph Sketcher, updating the x-axis maximum merely requires dragging the "180" label to the left until the desired maximum comes into view (and snaps to tick marks) — and all of the graph components are automatically repositioned accordingly in real time — so the entire operation requires just 2 task steps. In Illustrator and Numbers, the independent graph objects can be selected, grouped, and resized to the left, which unfortunately leaves the labels looking squished. In Excel, I had to



(a) Graph Sketcher (b) Excel (c) Numbers (d) Illustrator

Figure 4-6: The graphs produced by each of the programs tested in the diagram creation task.

re-adjust each component manually, requiring many more fine positioning steps (23 steps total).

The axis range adjustment task illustrates the importance of positioning graph objects relative to axis units. Axis adjustments can be necessary for many reasons, including reading units incorrectly the first time, extending a graph further into the future or the past (as in Figure 4-2), and making space for new explanatory labels.

4.3.3 Discussion

These task analyses verify that Graph Sketcher's interface — which integrates the three important capabilities of visual input, positioning relative to axis units, and snapping according to geometric relationships — indeed provides substantial user efficiency improvements over existing interfaces. Quantifying these gains more pre-

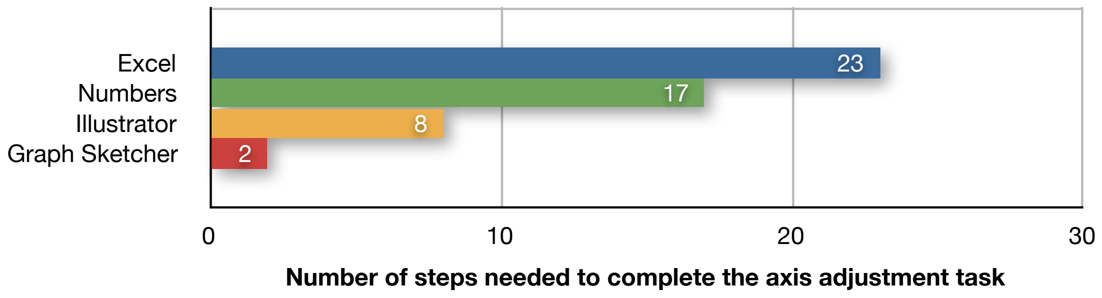
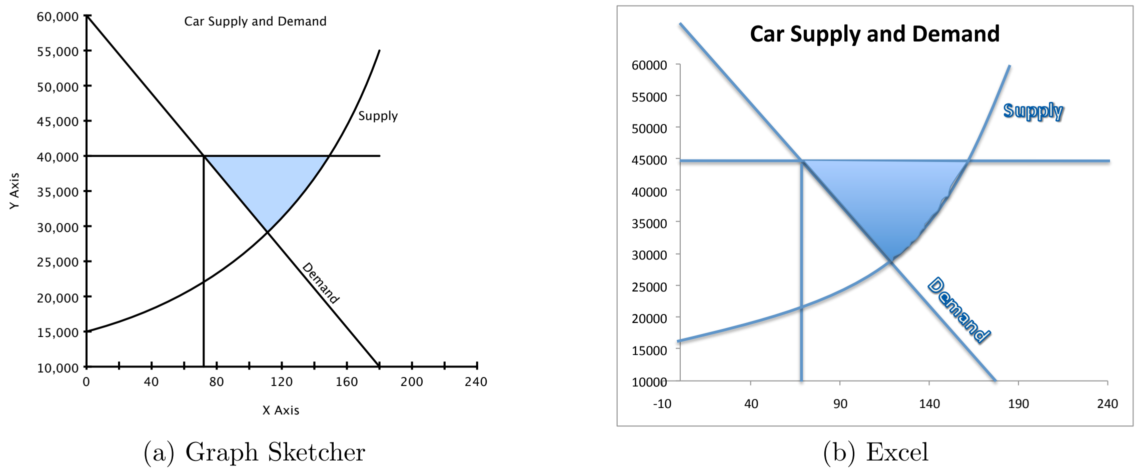
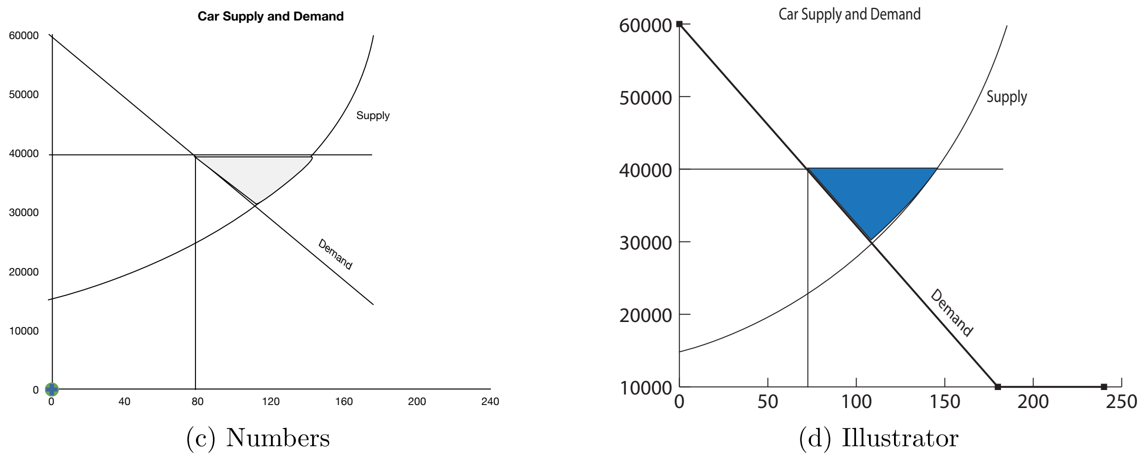


Figure 4-7: Results of the axis range adjustment task analysis.



(a) Graph Sketcher (b) Excel



(c) Numbers (d) Illustrator

Figure 4-8: The graphs produced by each of the programs tested in the axis range adjustment task.

cisely would require more detailed modeling, more comprehensive task analyses, or an in-lab user study. However, the task analyses presented here show a sufficiently large improvement that more detailed analysis is unnecessary for verifying the usefulness of the interface approach. In the next chapter, I describe how this interface can be extended to provide novel features that go beyond simply improving user efficiency.

Chapter 5

Exploration

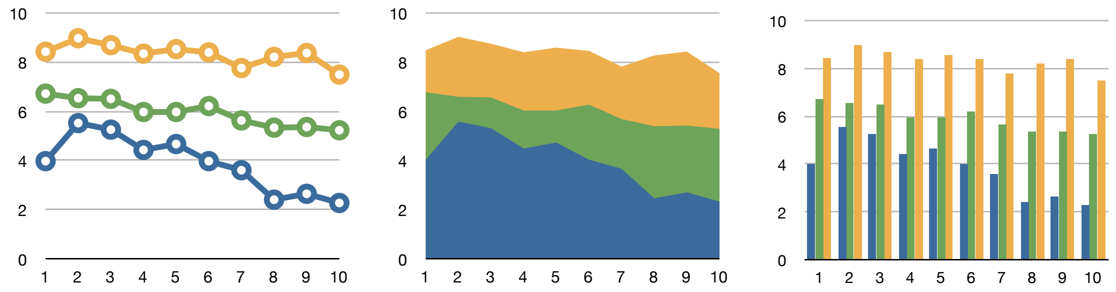
This chapter builds on Graph Sketcher's foundations to explore novel chart-based interaction techniques. In Chapter 4, I showed that integrating several important interface capabilities led to an interface design that was substantially more efficient than existing programs for creating Economics diagrams. In this chapter, I introduce several novel interaction techniques that take advantage of that unique foundation to allow users to interact with their quantitative diagrams in new ways. These new capabilities are similar to interaction techniques sometimes used in spatial domains such as illustration and GIS, but to my knowledge, they have never before been developed in the context of 2D graphs.

First, I describe how filled areas can be flexibly anchored to data series or other user-defined boundaries. This is an example of what I call *data-driven annotation* and I show that the concept extends to other objects such as statistical visualizations. Second, I describe how Graph Sketcher can be used to visually update and analyze data by taking advantage of the two-way mapping between screen and axis coordinates. Last, I dive into a detailed study of curve creation and manipulation interfaces. I critically examine the curve interfaces used in other domains and study three interfaces that are specifically optimized for quantitative conceptual diagrams.

5.1 Going Beyond "Chart Types"

5.1.1 Data-driven annotations

Chart-making software and information visualization programs try to achieve flexibility by offering in some cases hundreds of "chart types." For example, a "scatter plot" shows just the data points, while an "area" graph fills in the area underneath the data series (Figure 5-1). But both of these (and many others) are really just 2D relational graphs with the optional inclusion of special data-driven annotations such as shaded areas. A fundamental problem with the "chart types" paradigm is that even hundreds of options do not approach the combinatoric space of customizations that are possible. For example, even the simple shaded areas in Figure 5-2 are beyond the capabilities of all charting programs I am aware of.



(a) Line plot (b) Area graph (c) Bar chart

Figure 5-1: Common "chart types" found in data graphing programs.

Graph Sketcher goes beyond "chart types" to instead help users make their own *data-driven annotations*. For example, the edges of filled areas automatically follow data series and curved lines. This means that each of the shaded areas in Figure 5-2 require at most five clicks to create (and no searching through menus): with the "fill" tool, the user simply clicks on the start and end of the top and bottom data series segments to define the corners of the filled area (possibly adding a fifth corner at the intersection point in the center). The "draw" tool can be used similarly to connect the points in a data series. Thus Figure 5-2 can be re-created easily, as shown in Figure 5-3. These data-driven annotations can be thought of as providing a new type of constraint [8] that snaps objects according to data series instead of geometric relationships.

Proponents of the "chart types" paradigm might respond by adding yet another type of chart to their arsenal, perhaps a style that shades in areas between data series but not between data and the x-axis. But as noted earlier, such an approach does not scale. It is almost always easy to imagine how a given example could be extended beyond its "chart type." For instance, how should the chart type deal with a third data series, as in the examples of Figure 5-4a and 5-4b? Graph Sketcher's data-driven annotation approach allows either specification to be chosen easily. This makes it possible to easily create a far greater range of graphs than can be done with existing programs: using chart-making software, the examples of Figure 5-4 are impossible to create, and with illustration software they are at best very time-consuming.

5.1.2 Statistical annotations

Statistical features such as box plots, histograms, and best fit lines are also commonly handled by "chart types" in existing software. However, these are really just another type of data-driven annotation, and treating them as such in the interface enables new possibilities. For example, in Graph Sketcher the user can visually select a subset of data and then apply a best-fit line just to that subset. Other best-fit lines can then be applied to other, perhaps overlapping, subsets of data. Once again, this makes it possible to easily create a much greater range of graphs than is possible with the

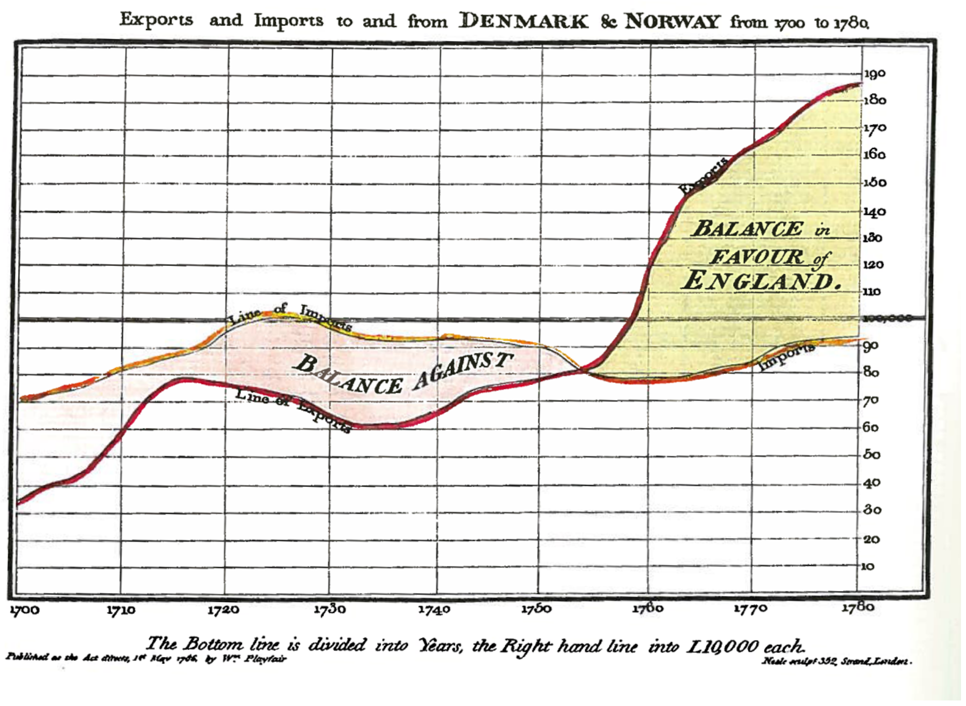


Figure 5-2: An engraving published in 1786 by William Playfair, a pioneer of information graphics who was not tied down by software limitations [38].

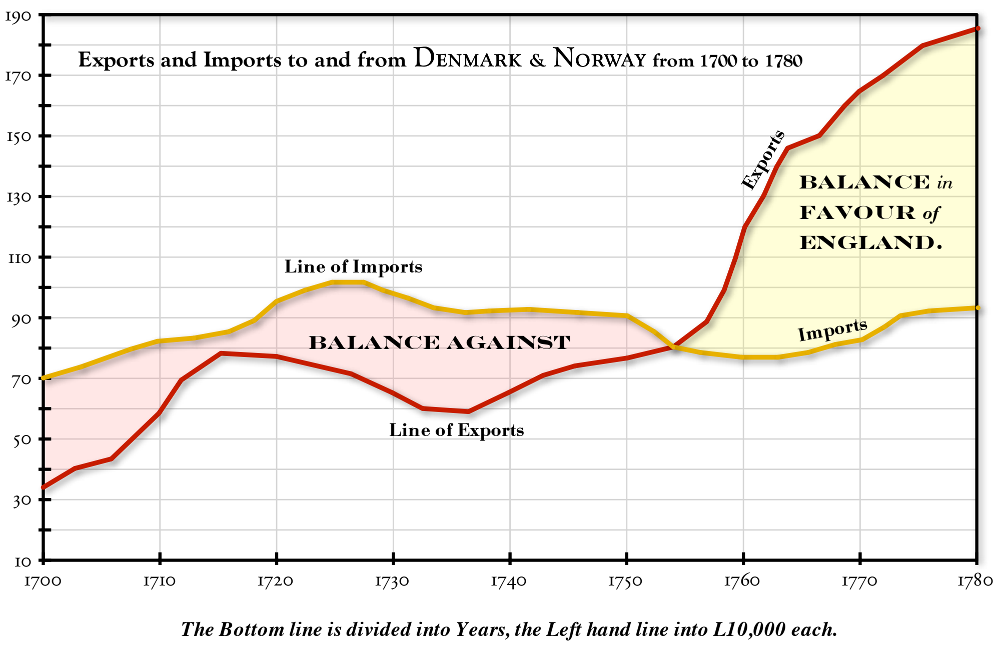
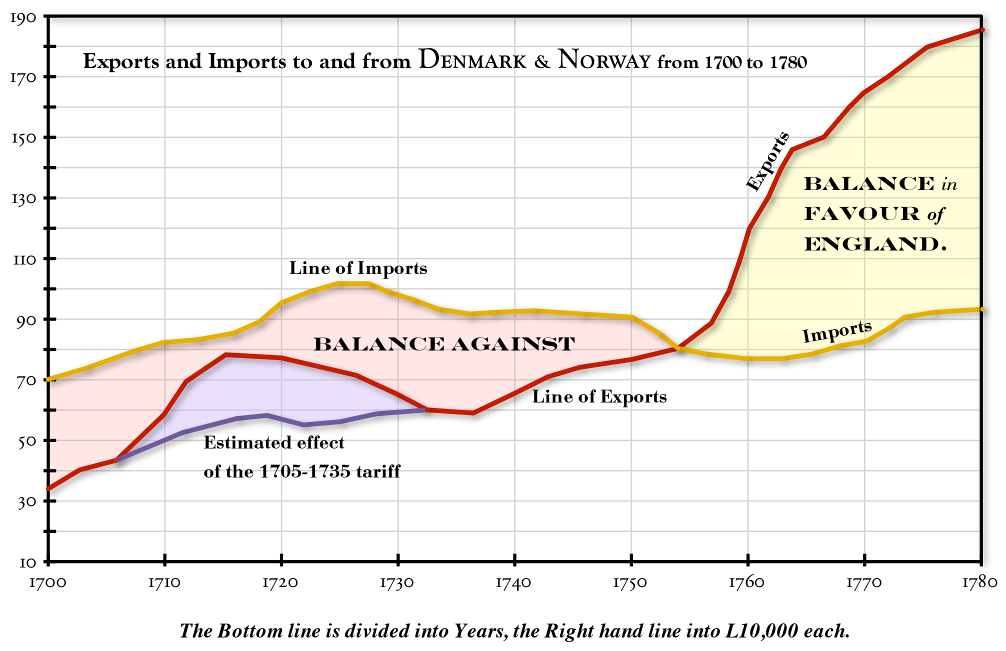
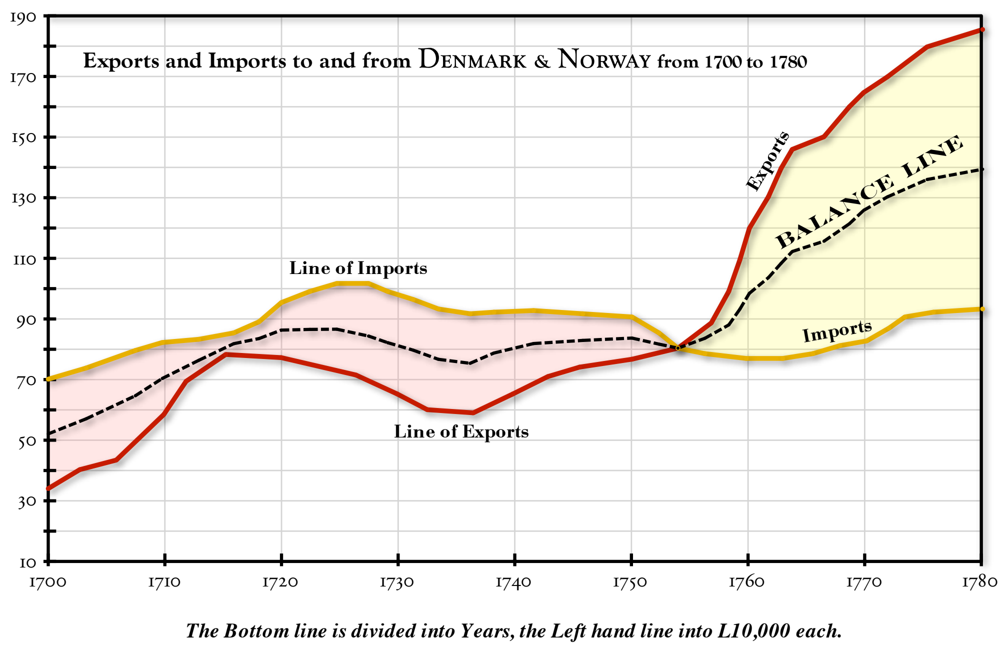


Figure 5-3: Playfair's graph (Figure 5-2) can be re-created easily in Graph Sketcher.

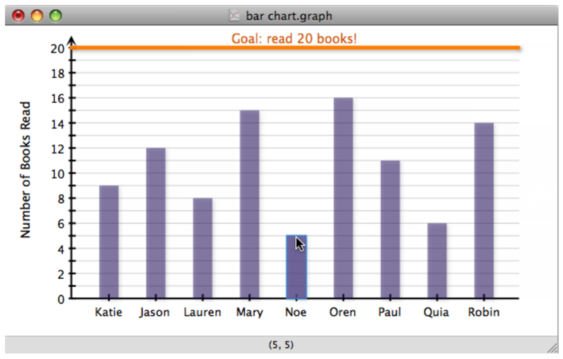


(a)

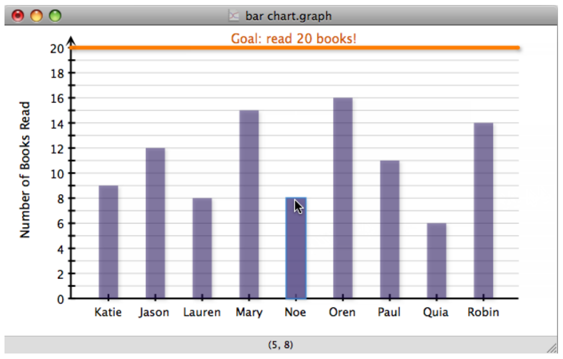


(b)

Figure 5-4: Extensions which are difficult to formulate as "chart types" can still be created easily in Graph Sketcher, as these two examples demonstrate.



(a)



(b)

Figure 5-5: Visual manipulation of data in a bar chart.

"chart types" approach of existing software, which limits statistical visualizations to whole data series. This framework generalizes beyond best-fit lines to any statistical visualization. Although Graph Sketcher only supports best-fit lines currently, I have completed proofs of concept with histograms and it is easy to imagine how this approach would apply to bell curves, box plots, and even pie charts.

5.2 Visual Data Manipulation and Analysis

So far, we have seen the benefits of being able to visually create and modify quantitative conceptual objects such as lines and filled areas. In this section, I show that these benefits apply to simple data points as well. Interactive information visualization tools such as Spotfire [37] let users visually inspect the data, but they do not let users update the data by dragging it. By contrast, all visual objects in Graph Sketcher can be directly manipulated. I also describe some of Graph Sketcher's features for quantitatively inspecting visual objects. Since all objects are positioned relative to axis units in Graph Sketcher, it can provide the type of information normally found in advanced mathematical software, such as line equations and averages. This capability is not present in any other chart-making or illustration programs because they do not treat user-defined lines and filled areas in this quantitative manner.

5.2.1 Visual data entry and manipulation

Once a data set has been plotted, Graph Sketcher allows the user to adjust data points visually rather than going back to the spreadsheet to enter new numbers. For example, if a student in a 3rd-grade class reads more books, the teacher can simply drag that student's bar upwards by the appropriate amount (Figure 5-5). The interface makes this even easier by snapping to grid lines and displaying the current coordinates in the status bar. To avoid modifying data points by accident, Graph Sketcher also provides an option to lock them in place.

If the data is not already in the computer, a user can enter it visually by simply clicking on the chart surface using the "draw" tool. For example, a patient that

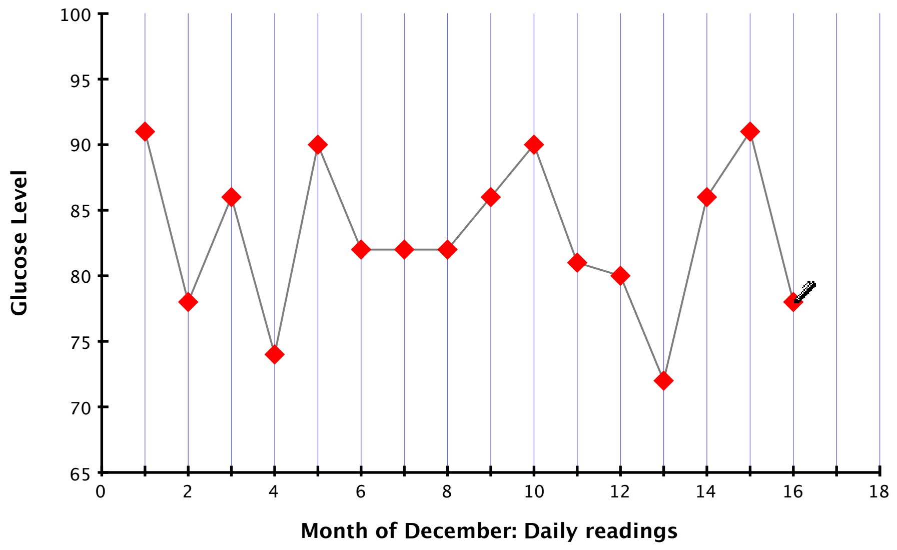


Figure 5-6: Visual data entry is performed by simply clicking on the graph surface in "draw" mode.

needs to track his glucose can take a daily reading and plot it by clicking at the corresponding spot in the graph (Figure 5-6).

5.2.2 Quantitative inspection

Graph Sketcher provides simple tools for examining data sets or conceptual components in a visual, quantitative manner. For example, selecting a point displays its coordinates in the status bar; selecting a line displays its equation. Selecting multiple points displays their average x- and y-values. An extrapolation can be explored by drawing an appropriate line and examining its position. In addition, all points can be viewed in tabular format in a spreadsheet-like listing or can be exported to another program for further numerical analysis. A tight connection between the table and the graphic is maintained by indicating the currently selected items in both views simultaneously.

Combining these techniques with statistical annotations (described in Section 5.1.2) allows for even more powerful visual exploration. For example, to understand the effect of an outlier on a best-fit line, a data point can be dragged around while the fit line is updated in real time (Figure 5-7).

5.3 Curve Manipulation

Many curve creation and manipulation techniques have been developed (e.g., [7, 17, 20, 26, 28, 2, 21]), but it is unclear which techniques best support the types of curves found in quantitative conceptual diagrams. In this section, I argue that

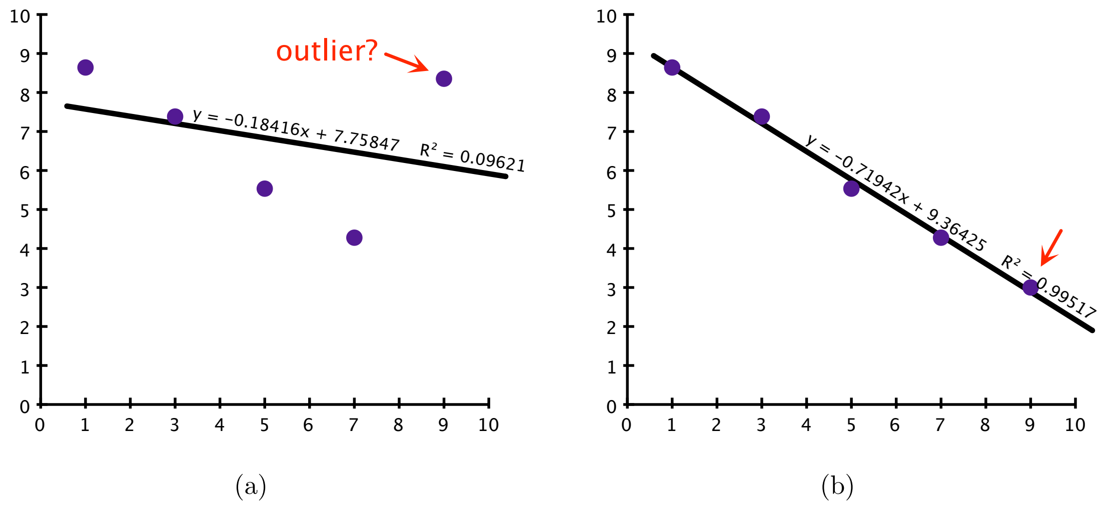


Figure 5-7: The data point marked "outlier" is moved around to demonstrate (in real time) its effect on a best-fit line.

the tangent-angle curve manipulation techniques used in illustration programs do not optimally support the types of curves that are used in Economics diagrams. I then describe Graph Sketcher's curve manipulation interface, which extends the interpolation techniques commonly used in data series plots (such as Figure 5-10) and applies them to conceptual curves.

5.3.1 Taxonomy of lines in economics diagrams

My analysis of the diagrams in a Microeconomics textbook (see Section 2.1.1) found four categories of line, which are illustrated in Figure 5-8. Ordered from most to least common, they are:

1. straight lines
2. simple curved lines
3. more complex curved lines that involve at least one inflection point
4. lines that follow data series (could be curved or straight)

Straight lines and simple curved lines are by far the most common, which suggests that the optimal curve creation and manipulation tools for quantitative conceptual diagrams may be substantially different than the tools used in illustration programs, whose users routinely create very complex curved shapes.

5.3.2 Position-based curve manipulation

Modern illustration tools allow the user to manipulate the position, tangent angle, and two-sided "weight" of each interpolation point via graphical manipulation handles, as

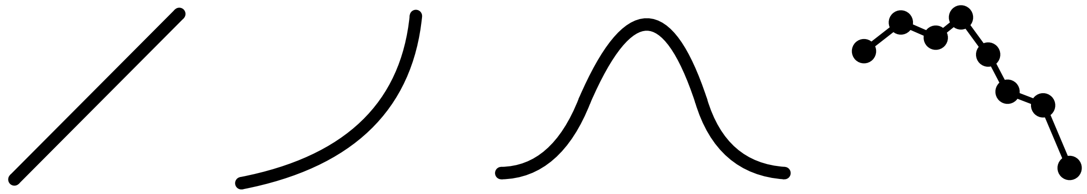


Figure 5-8: The four types of lines found in economics diagrams, ordered from most to least common: straight, simple curved, complex curved, and following data series.

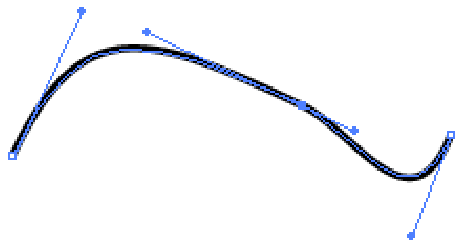


Figure 5-9: The standard curve manipulation handles provided by illustration programs.

shown in Figure 5-9 (using techniques derived by Fowler and Bartels [7]). However, in most quantitative diagrams, the position of the curve at certain key points is much more important than the tangent angles or the exact shape of the curve. This is particularly clear for lines that follow data series, such as in Figure 5-10. But it is also true for lines that are conceptual. For example, the graph for a calculus exam shown in Figure 5-11 is not based on real data but still requires that the curves pass through specific points (circled in red in the figure).

These curves must be smooth and look reasonable, but their exact shape is not important. Thus, rather than providing tangent angle controls, I chose to base all curves on a natural interpolating spline [29], which interpolates a given set of points and maintains a continuous second derivative throughout for maximum smoothness. With traditional illustration programs, the user has fine-grained control of the shape but often has to individually manipulate the tangents just to make the curve look reasonable. One way to speed up the curve manipulation is to use two-handed input to simultaneously manipulate the tangent angle and position of a point [17]. In Graph Sketcher, the user has less control over the shape but can still specify reasonable curves quickly using a standard cursor input. For example, if the curve has only one interior interpolation point, the user can explore all of the possible shapes in real time simply by dragging that point. In Graph Sketcher this interpolation point is visualized as a handle that will affect the bulge of the curve (Figure 5-12).

Manipulation of curves having more than one interior control point brings with it several interesting issues. Graph Sketcher does not yet fully implement such curves;

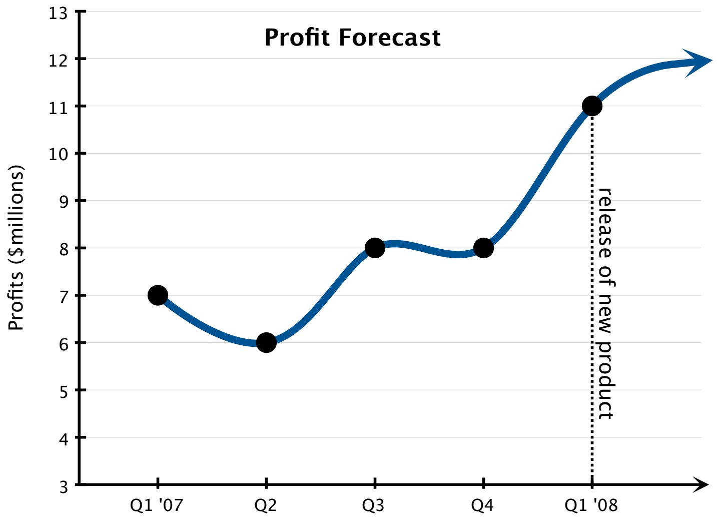


Figure 5-10: A graph with a curve that interpolates a data series.

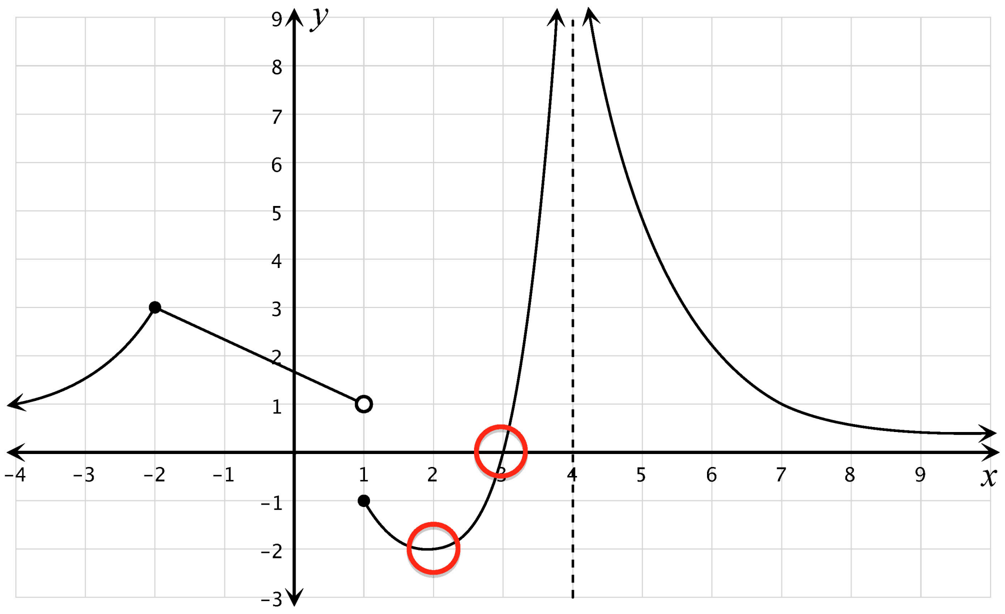


Figure 5-11: The middle curve in this graph needs to pass through (2, -2) and (3,0), both circled in red.

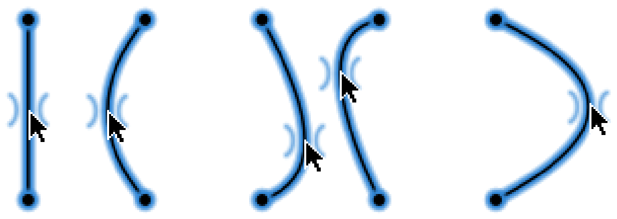


Figure 5-12: Curves can be manipulated very quickly in Graph Sketcher by simply dragging an interpolation point (here visualized as a "curve handle").

currently, users must create multiple line segments and manipulate their curve points individually in order to build up complex curves. However, the following interaction approach is planned. Between every interior point there appears a curve handle (as in Figure 5-12) which allows the user to further specify the curvature in that region by dragging. When a curve handle is dragged, it becomes an interior point and two new curve handles appear in the neighboring segments. An interior point can be removed by selecting and deleting it. An advantage of this "curve handle approach" is that it maintains the single "modify" tool — it does not require separate tools for changing the curvature vs. selecting and dragging the curve as a whole.

5.4 Curve Creation

In the previous section, I claimed that the types of curves used in Economics diagrams merit different manipulation techniques than are commonly used in illustration software. In this section, I study in depth several curve creation techniques that can generate the appropriate types of curves. The techniques I analyze are inspired by the pen-based interface approach because it clearly follows the intuitive, visual interaction paradigm used throughout Graph Sketcher (and even implied in the tool's name). But while I aimed to mimic the techniques of freehand sketching, I also aimed to improve upon the actual experience of drawing [33]. For example, it is intuitive to draw lines by hand, but they tend to be imprecise and wobbly; by contrast, computer interfaces should make it easy to create lines that are precise and smooth.[1]

I chose to implement and study three progressively more complex interfaces for curve creation, which I describe in detail below. The first is the "baseline" interface that is part of the publicly available version of Graph Sketcher, which only allows input of straight lines (which can later be curved, as illustrated in Figure 5-12). This

[1]As another example of improving upon a direct physical analogy, when designing the "fill" interface, I thought about closely imitating the behavior of a highlighter pen. The most extreme approach would force the user to manually shade the whole area by stroking back and forth. A more moderate approach has the user drag the cursor around the boundary of the fill, as one might do with a highlighter before shading in the interior. The design I eventually chose has the user simply click on each corner of the filled area in turn. This bears little similarity to using a highlighter, but it is the most efficient way to specify a polygonal visual object.

behavior is similar to “straight line” tools in illustration programs. The second is an “arc recognition” interface which allows users to specify simple curves having a single interior interpolation point. This behavior is unusual in that it allows arbitrarily complex strokes but always converts them into a simple curve. The third is a full stroke recognition interface which uses a series of calculations to convert a freehand stroke into a smooth, arbitrarily complex curved line. This functionality is based on existing work from pen-based interfaces [35] and is similar to “vector pen” tools in advanced illustration programs (e.g., [14]).

I ran an empirical user study to compare these interfaces, which led to the conclusion that the choice of interaction methods is highly dependent on the type of line being drawn. In particular, interaction methods that let users specify their input the most directly end up being the most efficient. Drawing a freehand stroke is a fairly indirect method of curve specification compared to other possibilities, and appears to be less efficient; combined with its difficulty of implementation, this study raises serious questions about the utility of stroke recognition in the domain of quantitative conceptual diagrams. As a novel alternative, I describe a composite interface that supports the creation of all common types of line, and optionally includes stroke recognition.

5.4.1 Straight Line Interface

Graph Sketcher’s “baseline” interface for creating lines assumes that new lines are straight. This is motivated by the finding that the majority of lines in the survey of textbook Economics diagrams are in fact straight (Figure 2-2). To create a new line in this baseline interface, the user enters “draw” mode and drags the mouse across the graph surface. Unlike in normal drawing, the drawn line is constrained to be straight — regardless of any meandering mouse movements. This means that the new line is simply a straight line between the current mouse location and the location at which the mouse button was first pressed down. This behavior makes it very easy to try out a variety of slopes and lengths for straight lines by simply moving the mouse around before ending the drag. To create a curve, the user must first draw a straight line and then use the curve handle to add curvature (Figure 5-12).

5.4.2 Arc Recognition Interface

I hypothesized that a more intuitive, quick, and precise interface for creating curves requires a closer approximation to freehand drawing. Such an interface requires some form of *stroke recognition*: in this case, a mechanism for converting wobbly freehand strokes into smooth, vector-based curves. Achieving robust stroke recognition is still an active area of research, but limiting the domain to the set of curves found in quantitative conceptual diagrams makes it possible to achieve a high degree of accuracy with current methods. Thus I created two new interfaces for creating lines, based on increasingly sophisticated stroke recognition techniques.

The first stroke recognition based interface assumes that each new drawn line is either straight or a simple curve (with no inflection points). These two categories are

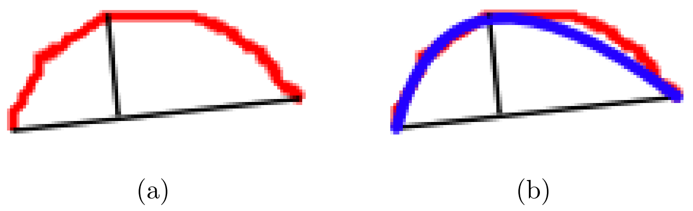


Figure 5-13: (a) A curved stroke with an illustration of the arc recognizer's geometry calculations, and (b) the resulting smooth curve overlaid.

sufficient for creating over 99% of the diagrams in an introductory Economics textbook (Figure 2-2). To create a new line with this interface, the user enters "draw" mode and draws the line by hand; the raw stroke is displayed on the screen while being drawn. When the mouse button is lifted, a simple arc recognition algorithm determines the extent and direction of curvature in the stroke and replaces the freehand line with a vector curve. As before, the user can delete and try again, or switch to "modify" mode to adjust the curvature.

This recognition approach required the addition of a stroke-capture mechanism in Graph Sketcher. During a stroke, the system continually checks the cursor location and the time using the finest temporal resolution possible: once per cycle of the event loop. However, digitized points are only recorded if the cursor falls on a different pixel than the previous data point. I will refer to these recorded points as *ink points*. Using this recording method, a typical sketched arc contains about 30 to 100 ink points.

The arc recognition algorithm calculates the perpendicular distance of each ink point from the diagonal between the stroke's start and end (bottom line in Figure 5-13a). The furthest ink point is chosen as the location of the curve point, resulting in a single smooth curve (Figure 5-13b). If the distance of the curve point from the diagonal is less than a set proportion of the diagonal length, the segment is made perfectly straight instead. Based on my own experimentation with trying to draw straight lines, I set this cutoff at 8 percent of the diagonal length.

5.4.3 Stroke Segmentation Interface

If we add the ability to recognize complex curves (involving one or more inflection points), then the user will be able to directly draw all of the lines found in all of the quantitative conceptual diagrams that were surveyed. An important observation from my analysis of economics diagrams (Section 2.1.1) was that "jagged" lines that are not based on data sets, such as the hypothetical example of Figure 5-14, are rare — none at all were found in the introductory economics textbook (Figure 2-2). There are several likely reasons for this absence: real world trend lines don't usually have such sharp discontinuities, and smoothly curving lines tend to look better for the purposes of rough forecasts or trends. Thus, it seems prudent for the stroke recognizer to assume that all drawn lines are smooth (have a continuous first derivative). Note

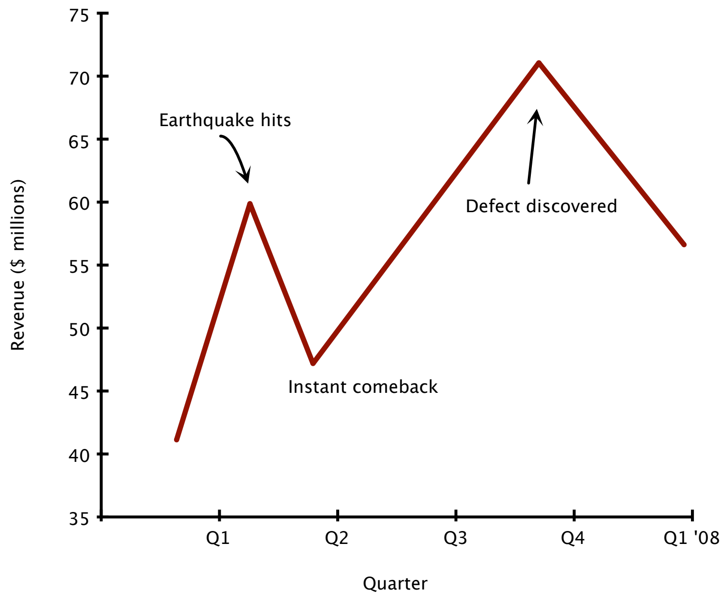


Figure 5-14: An unlikely example of a jagged line in a quantitative diagram.

that this does not limit the interface's expressive *power*; a user can still create a jagged line by simply drawing two coincident lines instead of a single stroke.

The observations throughout this chapter motivated the following desired properties of the full stroke recognizer:

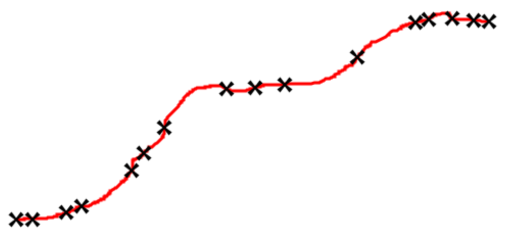
1. *It should avoid over-fitting noisily drawn strokes.* Most strokes are probably intended to be long, straight segments or gently curving arcs. If recognized as such, they will look smoother and will be easier to adjust later.

2. *It should assume all lines are smoothly curving and thus does not need to recognize sharp corners.* For the rare cases when users do want jagged lines, they can simply lift the mouse button or pen momentarily and start a new stroke.

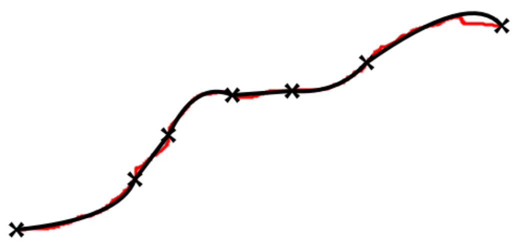
3. *It should be able to recognize inflection points.* To support complex curves, the recognizer must be able to detect changes in curvature despite a lack of sharp corners.

Implementation

My stroke recognizer implementation is based on the method in Stahovich [35]. The recognizer starts by computing the arc length, smoothed slopes, and smoothed curvature at each ink point. I used a window size of 7 ink points on either side (15 total) for computing the slopes with a linear regression, and 10 ink points (21 total) for similarly deriving the curvature. These window sizes were settled on during early experimentation; their exact values do not greatly affect the final accuracy of the



(a) initial segmentation



(b) pruned segmentation

Figure 5-15: The (a) initial and (b) pruned segmentation of a stroke.

segmentation. Using the smoothed curvature values, the algorithm sets candidate segmentation points everywhere the curvature changes sign, illustrated in Figure 5-15a. It is clear that the recognizer would create far too many segments if it stopped at this stage. Property 1 above specifies that as few as possible segments should be generated, while still doing a good job of fitting the curve. So I needed a way to prune the set of candidate segmentations.

The intuition is that neighboring segments which both curve in the same direction should really be part of one longer segment. Thus to determine whether neighboring segments should be merged, the stroke recognizer needs to determine the direction and amount of curvature of each segment. To determine this, my stroke recognizer computes the *adjusted total curvature* (ATC) of each proposed segment. This measure adds up the curvature values at each point in the segment, multiplies this sum by the segment's total arc length, and divides by the number of ink points n in the segment.

$$ATC = \frac{\Sigma(curvatures) \cdot \Sigma(lengths)}{n} \tag{5.1}$$

The rationale behind the ATC measure is as follows. The sum of the curvature values determines how substantially a segment is curved and in what direction it bends (positive or negative curvature). For noisy, squiggly segments, the positive and negative curvature values will roughly cancel each other out. Multiplying by the arc length boosts the ATC of segments that are longer, so that long segments are more likely to be retained than shorter ones. Finally, the slower the user draws a line, the more ink points get recorded, which artificially increases the summed curvature; so dividing by the number of ink points (n) in the segment cancels this effect.

The recognizer labels each segment as negatively curved, straight, or positively curved, and if two subsequent candidate segments have the same type, they are merged into a single segment. Based on my own experimentation with a range of line shapes and drawing speeds, and observation of one other user, I set an ATC

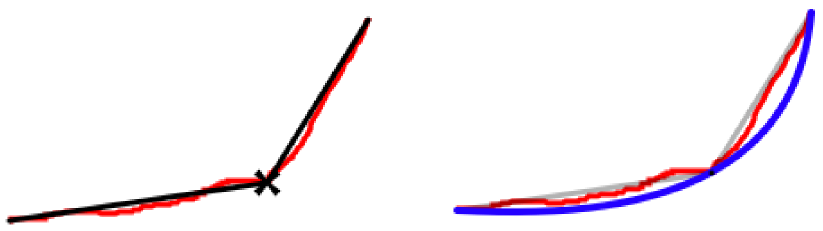

(a) initial segmentation (b) final interpretation

Figure 5-16: The (a) initial and (b) final recognition of what seemed to be two straight segments.

threshold of 30 as the cutoff between labeling segments as straight vs. curved. This means that segments with an adjusted total curvature above 30 are deemed to be positively curved (i.e. curving to the left), those below -30 are deemed negatively curved (i.e. curving to the right), and those between -30 and 30 are deemed straight. In addition, if the segment's curvature is less than the arc recognition algorithm's 8% cutoff (described in section 5.4.2), the segment is deemed straight regardless of the ATC.

This process of pruning is continued recursively until no more segmentation points can be removed. Finally, the arc recognition algorithm (see section 5.4.2) is used to determine the actual curvature of each surviving segment, and the resulting curved segments (Figure 5-15b) then replace the stroke ink. Note that segments made up of merged "straight" pieces may well end up substantially curved (Figure 5-16). As before, the user can modify these curves if necessary.

Stroke recognition interface

Even with very long and complicated strokes, the full segmentation algorithm finishes within two milliseconds on a 2.16 Ghz Macbook Pro. This low latency actually makes it possible to view the recognized line in real time as the user is drawing the stroke. While this functionality was very useful for iterating the stroke recognition algorithm, it proved disorienting for regular program use. Thus, the versions of Graph Sketcher used in the following sections do not present a recognized curve until the user has finished the stroke.

5.4.4 Comparative User Study

I ran a formative evaluation with five participants to estimate the accuracy of stroke segmentation and to compare the three interface approaches. I used a within-subjects design so that each participant used all three of the interfaces (in randomized order). For each interface, I asked participants to recreate the two hand-drawn graphs in Figure 5-17 using Graph Sketcher, "keeping in mind that this is just a sketch: you do not need to do any calculations or precision measurements. Overall, speed is more important than perfection." I also instructed participants to ignore axes, grid lines,

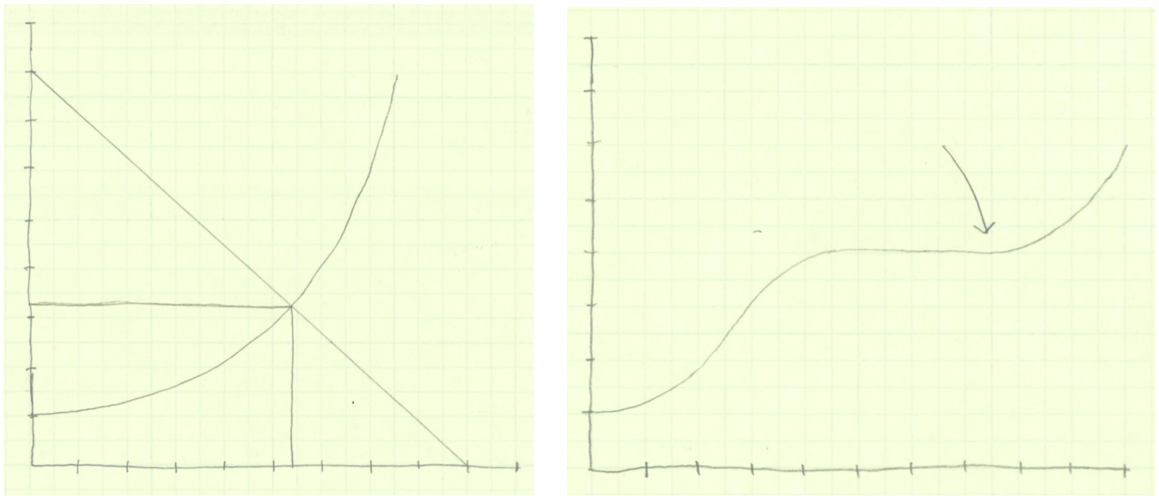
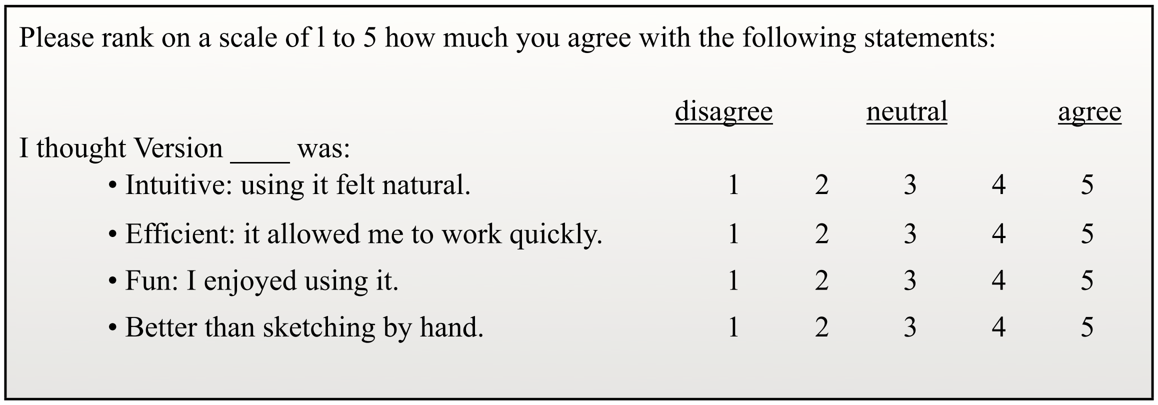


Figure 5-17: The graphs that users were asked to re-create with Graph Sketcher.



Please rank on a scale of l to 5 how much you agree with the following statements:

	disagree		neutral		agree
I thought Version ____ was:					
• Intuitive: using it felt natural.	1	2	3	4	5
• Efficient: it allowed me to work quickly.	1	2	3	4	5
• Fun: I enjoyed using it.	1	2	3	4	5
• Better than sketching by hand.	1	2	3	4	5

Figure 5-18: Study participants were asked to fill out this short questionnaire after using each interface version.

and any other formatting. I briefly described the purpose of Graph Sketcher at the beginning of the session, but I did not provide any instruction about how to use the program or any of the curve interfaces, nor any information about how the interfaces differed.

While each participant worked, I watched closely and recorded how many times lines were drawn, deleted, adjusted in position, or adjusted in curvature: in short, the number of actions needed to specify the line to satisfaction. I captured these counts separately for each line type (straight, simple curve, and complex curve). After the subject completed a pair of graphs, I had them fill out a short questionnaire evaluating the interface version they had just used (Figure 5-18). At the very end of the session, each participant was also asked to rank the three interface versions according to which they liked most.

	draw attempts	curvature adjusts	endpoint adjusts	deletions
straight	1.3	0	0.7	0.6
arc	1.2	0.4	1	0.8
complex	2	3.2	4.8	1.2

Table 5.1: Stroke segmentation accuracy as measured by the number of adjustments users made.

5.4.5 Results

Stroke recognition accuracy

The accuracy of stroke recognition is highly user-dependent. It is not always clear what the correct stroke is, and in the context of sketching graphs, it is often the case that a whole range of lines are good enough for the purposes of the sketch. Thus, I measure accuracy by recording the number of times users adjusted or deleted the lines they drew in the stroke segmentation interface. However, the results are for new users who had never used the interface (or the software) before, so some of the deletions and adjustments can be classified as learning effects. In addition, in all cases subjects used a mouse — an input device which is fairly difficult to accurately draw with. Some users also experimented with a digitizing tablet, but in the end were more comfortable using a mouse for the study.

Table 5.1 shows the average number of adjustments users made for each type of line (straight, arc, and complex curve). Straight lines were recognized extremely accurately, with none of them being interpreted as curves, though occasionally the recognizer split them into two segments, causing a deletion or a redraw. Arcs were also recognized correctly most of the time, though again there were some unwanted segmentations and on average users corrected the curvature or endpoints almost every time. Finally, complex lines almost never turned out right the first time, and usually most of the segments were adjusted later in both curvature and endpoints.

The results for complex curves would clearly be improved if Graph Sketcher supported multiple interior control points. That way, even if there were too many segmentations, the resulting curve would be smooth; and most segmentation errors could be fixed by simply deleting a control point or two. This feature has not yet been completed because the implementation of natural interpolating spline curves — and their integration with other features of Graph Sketcher such as fills with curved edges — was much more challenging than expected.

Comparative Usability

The three interfaces I tested roughly correspond to the three types of lines which users were asked to draw: in the original interface, strokes are constrained as lines;

	straight line interface	arc interface	recognizer interface	average
straight	1.3	2.4	2.6	2.1
arc	4.4	1.4	3.4	3.1
complex	11.6	11.6	11.2	11.5
overall	5.8	5.1	5.7	5.5

Table 5.2: The average number of steps users took to create each type of line in each interface version.

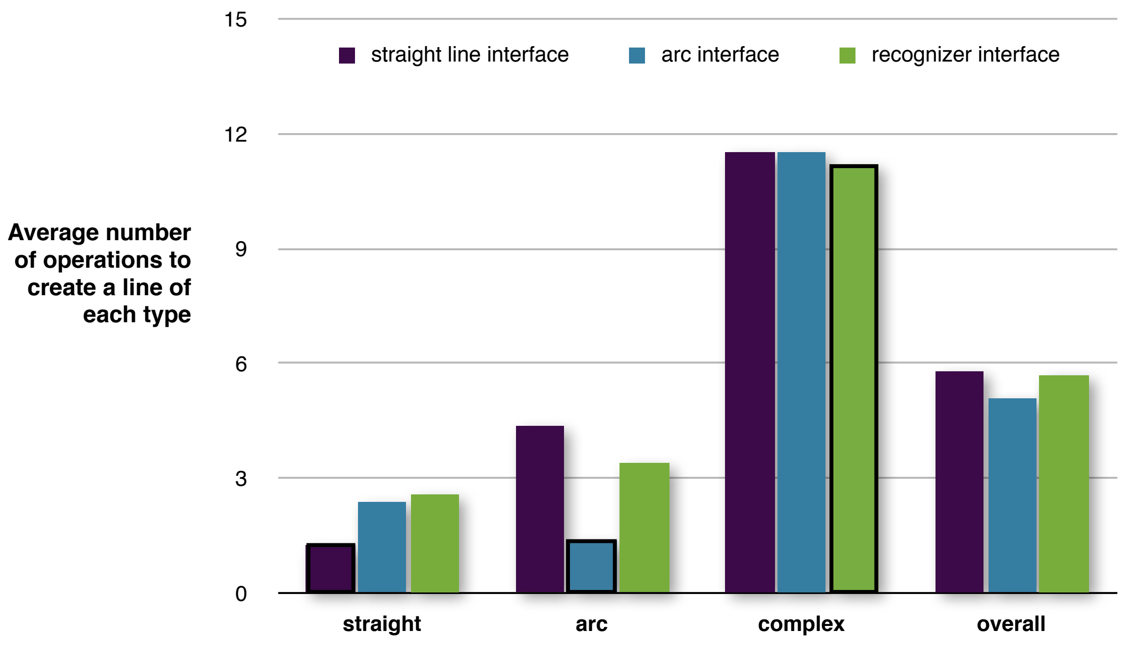


Figure 5-19: The average number of steps users took to create each type of line in each interface version. The boxed results indicate matching line and interface types.

in the intermediate interface, they are confined to single arcs, and in the full stroke recognition interface they can be complex curves. A reasonable hypothesis, then, is that each interface is best matched to drawing its corresponding type of line. The results support this hypothesis (Table 5.2 and Figure 5-19). The advantage of matched interface and line type is much more pronounced for straight lines and arcs in these results, but I suspect that support for multiple interior control points would extend this trend to complex curves.

Questionnaire results

Users did not rank the interfaces significantly differently on the measures of intuitive, efficient, and fun. However, when asked to rank the three interfaces, almost all users preferred the arc interface and least preferred the recognizer interface.

5.4.6 Discussion

When the interfaces were compared on their performance for creating *all* types of lines ("overall" results in Figure 5-19), none of the interfaces were clear winners. This suggests that the best interface will integrate the three versions studied in order to give users access to all of them. My proposal for such an interface, which still maintains a single "draw" tool, is as follows. A single click starts the drawing process. If the user lifts the mouse and moves it to another location, a straight line is shown between the original click and the current cursor position. Further single clicks establish interior control points for a smooth curve, similar to how the arc interface was really based on three input points. A double-click ends the line segment. Finally, if the user drags the mouse instead of using separate clicks, an ink stroke is drawn which gets automatically segmented when the stroke is finished. This latter method may be most intuitive for some users, even though the generalized "series of clicks" curve creation method has the same expressive power and less ambiguity.

Implications for stroke recognition

The poor user acceptance of the sophisticated stroke recognition interface helps to explain why pen interfaces have not become more widely used. Even for an application domain that clearly lends itself to a sketch-like interface, and even with a fairly accurate stroke recognizer, the recognition approach was the least useful. For one thing, users seemed disconcerted by the unpredictable nature of the stroke recognition; they were downright annoyed when the system failed to read their mind. Users also seemed stressed about having to perform accurately in order for the system to correctly recognize their intentions.

Although improvements in software and hardware interfaces could lessen both of these problems, I think the deeper issue here is that of appropriate constraints. Pen interfaces tend to be highly unconstrained, which gives them flexibility and power but also makes them overwhelming, stressful, ambiguous, and often inefficient. The most obvious example is with text input: typing is faster, more satisfying, and more accurate than tablet PC handwriting precisely because typing is so much more constrained: each button does precisely one thing. Even if there existed a handwriting recognizer that recognized with human accuracy, most people would still rather use a keyboard for the task of inputting characters.

A similar argument can be made for the domain of quantitative diagrams. All lines, whether straight or wildly twisted, can be represented as a series of interpolating control points (which specify the locations the curve passes through). The arc interface in effect lets users precisely and easily specify the three points of a simple arc segment, without any unsettling surprises from the recognizer. Rather than coping with the true ambiguity in every pen stroke, it seems more efficient and elegant to simply specify one's intentions at the outset using the "series of clicks" interface I have proposed. No matter the future improvements in automatic stroke recognition, clicking in a series of precise locations will still be easier than precisely drawing a pen stroke.

Chapter 6

Evaluation

In previous chapters I evaluated the extent to which specific interface techniques make it easier to create quantitative conceptual diagrams. In Chapter 4, I performed heuristic task analyses to measure the benefits of comprehensively integrating three fundamental capabilities: visual input, positioning relative to axis units, and snapping according to geometric relationships. Comparing Graph Sketcher to a variety of existing programs, I showed that Graph Sketcher's approach provides substantial user efficiency improvements in all areas measured, including positioning lines, creating filled areas and labels, and adjusting axis ranges. In Chapter 5, I ran a small user study with the primary goal of comparing the performance of three curve creation techniques. The results showed that each technique was best suited to the creation of a specific type of line — straight, simple curved, or complex curved — which led to the conclusion that the three techniques should be integrated into a single "draw" tool. The study also demonstrated in general that novice users with no prior instruction could quickly create simple diagrams using the software.

In this chapter, I turn to more holistic, longitudinal evaluations of Graph Sketcher's use "in the wild." The purpose of these evaluations is to verify that the improvements highlighted earlier do indeed make it easier to create a wide range of quantitative conceptual diagrams (not just in economics), by real users in their everyday settings. I also wanted to ensure that the interface is not just more efficient but also intuitive to users, and that it enables users to create graphs that they would not have even attempted in the past due to the complexity or limitations of existing tools. To do this, I studied the experiences of people who already use Graph Sketcher in their everyday work.

6.1 Longitudinal Evaluations

The most basic indication that Graph Sketcher has succeeded in its design goals is the simple fact that indeed a large number of people voluntarily use it for their everyday work. As of this writing, there are 764 registered users; these are people who independently found Graph Sketcher online, downloaded and started using it, and ultimately paid \$10 to \$30 to continue using it. To find out more specifically whether the novel interface features contributed to this success, the extent to which the program as a whole is easy to understand, and the range of diagrams its users can create, I carried out two investigations of the use of Graph Sketcher "in the wild."

First, I sent out a survey which requested examples of charts made with Graph Sketcher. Of the 478 people who were emailed the survey, 20 (4%) participated. Second, I looked at two months of support email received from a potential pool of 623 registered users. There were 34 such emails. Throughout the investigation period (and afterwards), I periodically released updated versions of Graph Sketcher, and new users continued to download and begin using the program. Most of the features described in this thesis were available throughout the investigation period; the exceptions are some of the advanced features described in Chapter 5: bar charts, filled areas bounded by data series, and multiple-interior-control-point curves. I describe the results and analysis of these investigations below.

6.1.1 Analysis of support logs

If Graph Sketcher's interface were confusing to users, we would expect some of them to ask support questions about how to use it. Yet out of 34 support emails received, none complained of or asked questions about ways to carry out any particular operation. Six emails asked whether it was possible to, e.g., add subscripts, un-snap points from each other, or automate the tool. In each case the answer was "not yet" so these emails were considered feature requests. Including these, 76% of the support emails were feature requests; none of the requests were for features which would violate Graph Sketcher's interface techniques (i.e. there were no requests to provide "chart types" or to disconnect objects from the axis scale). 21% of the emails were bug reports, and one was simply an unsolicited note of appreciation.

6.1.2 Survey of example graphs

My survey went out to the 478 users who were registered when it was administered. I requested an example graph, asked in open-response format why they used Graph Sketcher instead of another program to make their graph, and offered a $15 gift certificate for the most "interesting" graphs received. I chose to ask for interesting graphs rather than random samples in order to better understand the breadth of Graph Sketcher's capabilities. I received 20 responses (a 4% response rate), from nine different fields of study: economics (8), chemistry (3), biology (2), physics (2), calculus, materials engineering, consulting, population studies, and fly fishing. This long-tail distribution with economics making up about half is roughly representative of Graph Sketcher's full user base, according to feedback I request when users register the software.

I explicitly designed Graph Sketcher to support the use of conceptual components (such as trend lines and areas) because my analysis of economics diagrams (Section 2.1.1) indicated that those components are widely used. Thus, I expected users' graphs to contain many conceptual elements. This prediction was supported by the survey results. Defining "conceptual components" as any marks that are not data points, titles, legends, or standard axis components, 18 of the graphs (90%) contain them: 13 use a combination of data points and conceptual components, while 5 have no underlying data at all (such as Figure 6-1). The remaining two used Graph Sketcher as a quick way to plot data. I note that because I encouraged users to send their most "interesting" graphs, these results are probably biased towards the more heavily annotated. Regardless, these responses demonstrate that users understand how to create conceptual components and use them in their graphs at least some of the time.

Finally, I looked at why respondents said they used the program. Of the 15 who answered this open question, the most common reasons cited were: generally easy to use (7), fast to use (5), intuitive (3), precise annotations (3), simple features (2), visual data entry, and fit-lines on subsets of data. Most of these responses were quite vague ("easy to use") but I highlight the three most specific responses below. These users contrast Graph Sketcher with traditional programs and cite the importance of precise conceptual components.

> I teach introductory chemistry and advanced inorganic chemistry. Graph Sketcher has been of tremendous help in generating quick, qualitative graphs easily. I'm attaching the phase diagram for water to this e-mail (Figure 6-2). What made Graph Sketcher so useful was the ability to easily draw lines, and then to color in the different phase areas; the colors actually stuck to the lines! The program has definitely freed time I would have spent fiddling with other drawing programs, and has allowed me to devote more time to my course content.

A materials engineer wrote:

> This is actually the first figure I sketched with your software and it took about 5 minutes (Figure 6-3). Until now, I've been drawing these by hand because plotting in Excel was too tedious (each line represents a lot of math), and drawing them freehand in Inkscape [an illustration program] was too inexact.

A high school teacher who uses Graph Sketcher in his classroom wrote:

> Here's a graph a lab group in my physics class recently turned in (Figure 6-4). [It] shows the relationship between the object and image distance for real images formed by a concave mirror....
>
> Excel is way too complicated for our average student's comfort or ability level. Graph Sketcher takes very little time out of physics instruction, and the kids just "get it" right away.

As evidence, his students "got the lab and the graph done in less than 40 minutes" (Figure 6-4).

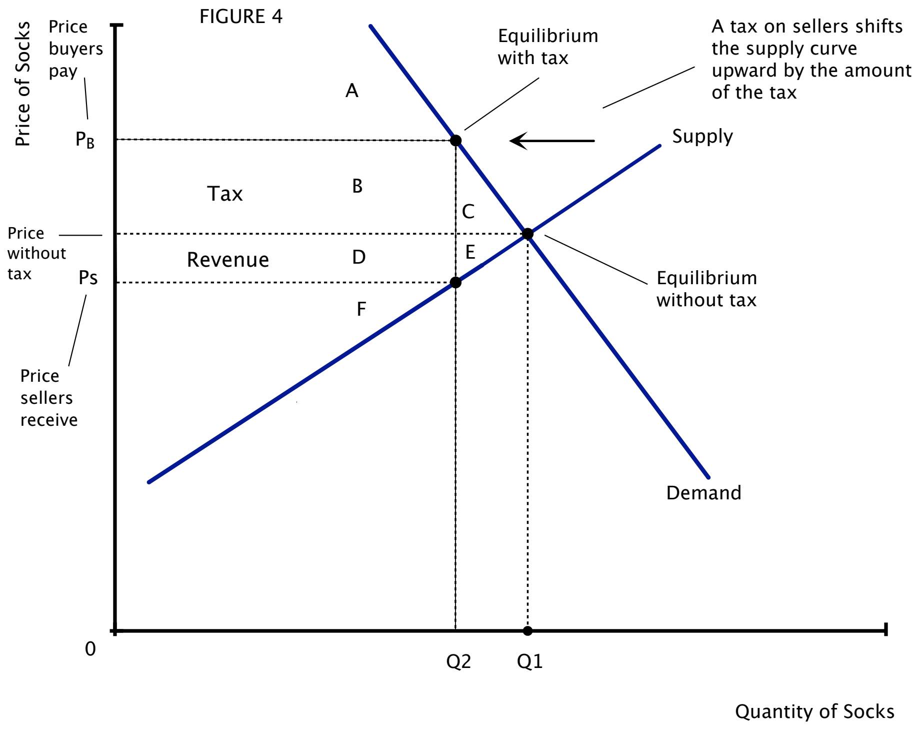


Figure 6-1: A user-created economics diagram which is not based on numerical data.

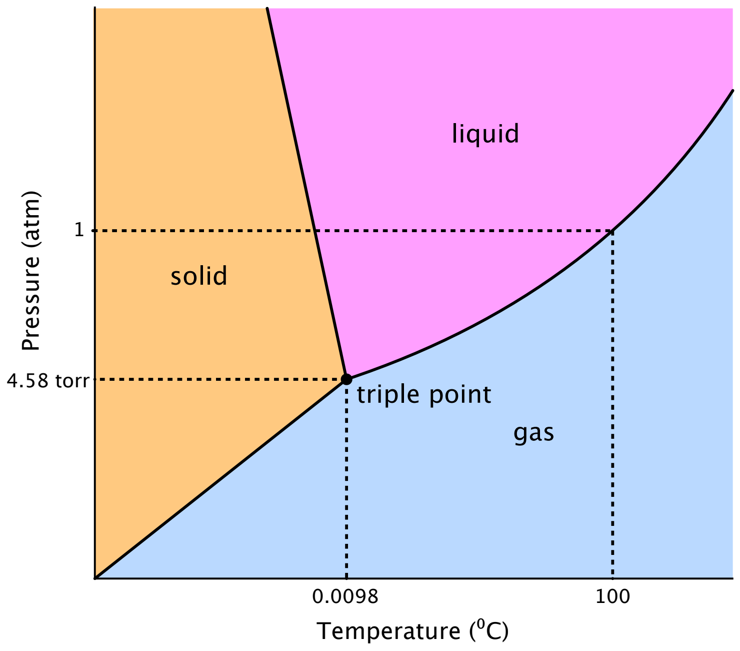


Figure 6-2: A user-created chemistry diagram showing the phases of water.

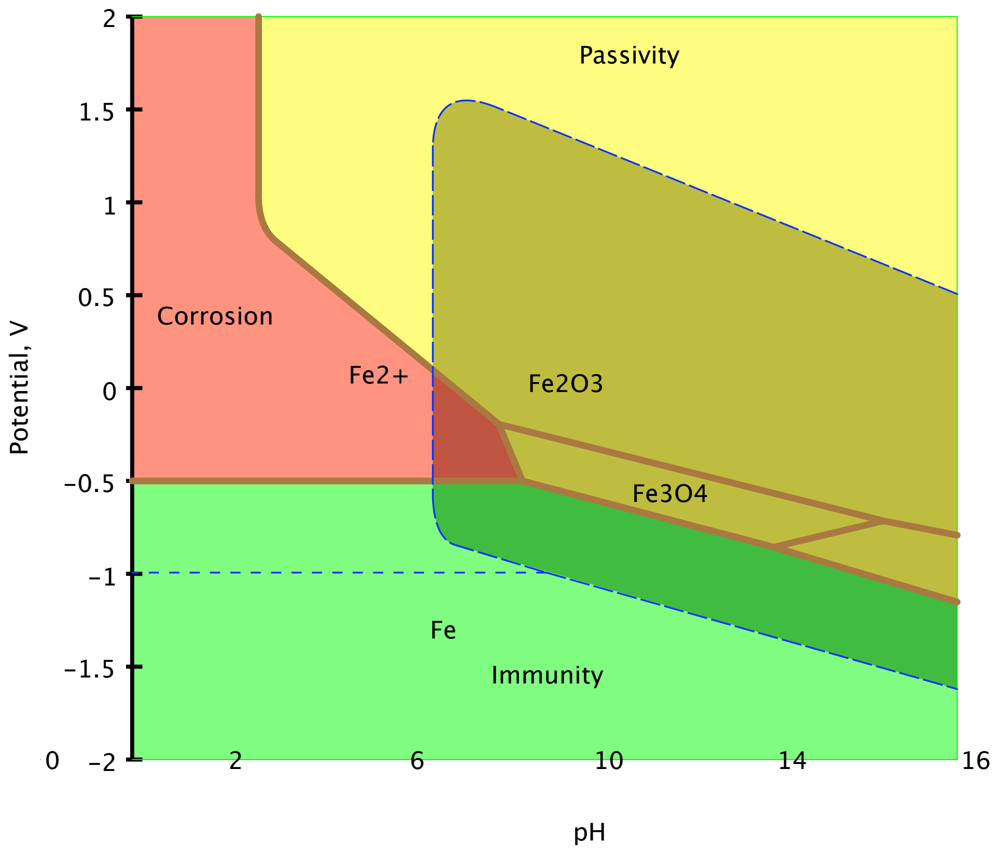


Figure 6-3: A user-created diagram for materials engineering.

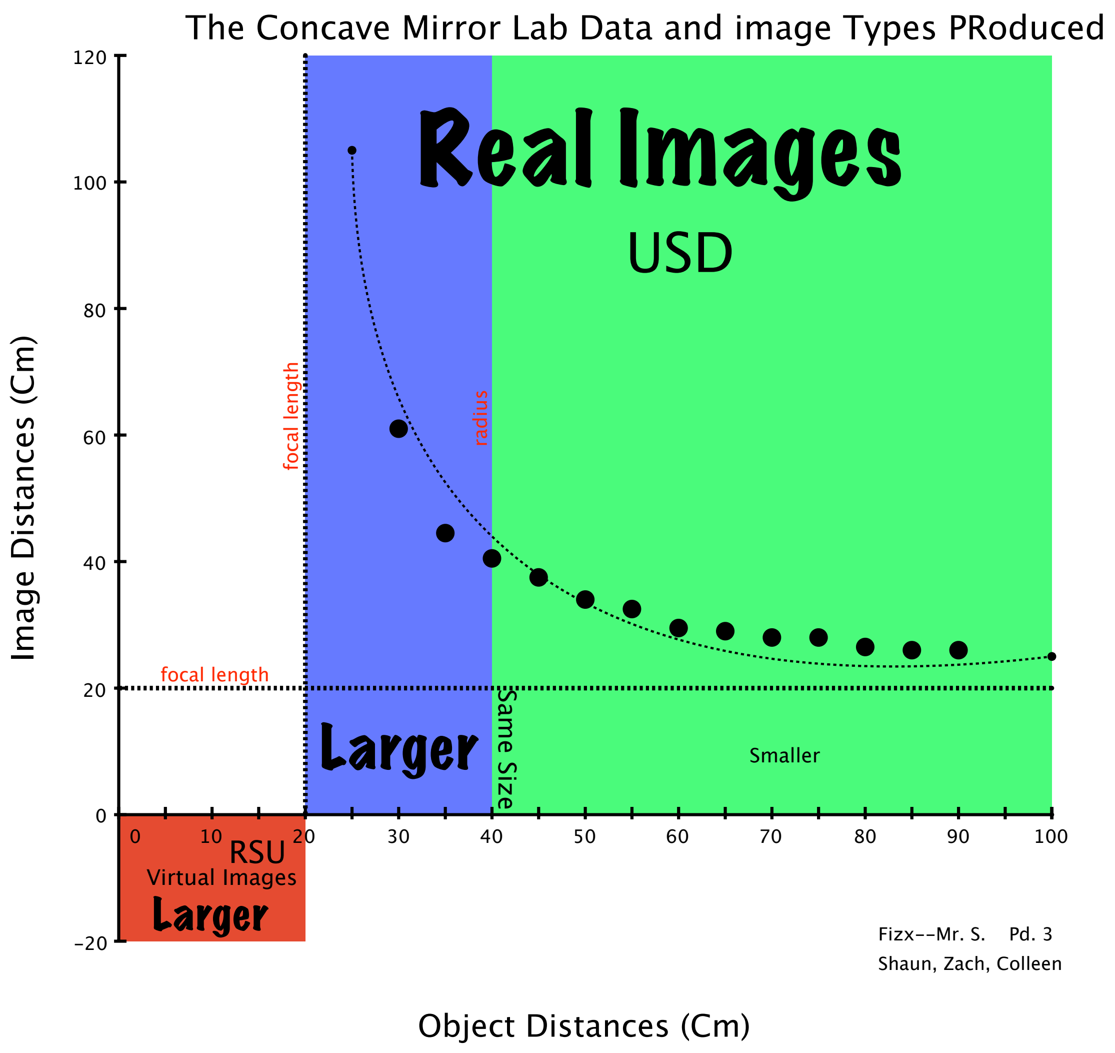


Figure 6-4: A graph produced by high school students for an in-class optics lab.

6.2 Discussion

The results of the longitudinal evaluations show that the interface is useful in a wide range of disciplines, beyond those that were used to motivate my design. Across these disciplines, users took advantage of Graph Sketcher's capabilities to produce precise, conceptual, quantitative information graphics. The survey also showed that the tool serves a wide user population that includes both university and high school students and faculty, authors, hobbyists, and consultants. Finally, the fact that many people are voluntarily using the software is compelling evidence that the existing tools are insufficient. After all, these users were not recruited to use Graph Sketcher in the lab; all of them found it online. According to my website's log statistics, more than half of visitors came via search engines — indicating that users are proactively going in search of better alternatives.

What most distinguishes Graph Sketcher from preexisting programs is not any one feature in particular, but rather the careful integration of many useful features. As discussed in Chapter 2.4, Excel includes both plotting and illustration tools but does not integrate them beyond allowing illustration objects to appear on top of data plots. By contrast, my interface treats all components as both data *and* illustration objects, so that whichever properties are most appropriate at a given time can be used. The task analyses in Chapter 4.3 showed that snapping features are crucial for tasks such as line creation, while positioning relative to axis units is most useful for axis adjustments. Importantly, these two features are inseparably linked: the lines snap to grid positions defined by axis units, and the axis adjustments themselves use snapping features. Even in the narrow subtopic of curve creation (Chapter 5.4), my studies showed that different techniques were most useful for different types of curves — and thus the challenge is to integrate these techniques in a way that is efficient to use and easy to understand.

When I asked users to explain why they used Graph Sketcher, most people could not narrow it down to any specific feature. As described above, half of the survey respondents simply sent comments equivalent to "easy to use." Outside the context of the survey, a user sent me the following feedback:

> thank you, thank you, thank you [...] you made me the little quick and easy program that doesn't do things i don't want it to and acts like a student with graph paper.

In response, I asked him what "specifically makes the difference" compared to Excel and other tools. He replied simply, "Three words for your program – Ease of Use!" and went on to describe what he used it for and what other features he wanted. It seems that what underlies the interface's usability is not one or two features in particular, but rather all of the little features that emerge from the comprehensive integration of functionality.

Chapter 7

Conclusion

In this thesis I motivated and designed a new interface for creating quantitative conceptual diagrams. This interface both provides all features necessary to make beautiful diagrams such as Figure 7-1 and is dramatically more efficient to use than existing programs. It goes beyond existing chart-making tools by comprehensively integrating three capabilities:

- Visual input of lines, labels, and filled areas;
- Detection and maintenance of geometric relationships between components;
- Positioning of all components relative to axis units.

I also explored some of the novel possibilities offered by this interface approach, including flexible data-driven annotations, visual data manipulation, and intuitive curve creation and manipulation.

To motivate the research, I showed that quantitative conceptual diagrams are widely used in economics but that there is a common desire for software that makes it easier to create such diagrams. To evaluate my design, I implemented and deployed a desktop software application, Graph Sketcher, which is already being used by over 700 students, teachers, professionals, and hobbyists worldwide. I found that Graph Sketcher users seem to understand the interface well and have created a wide variety of conceptual graphs for many disciplines beyond economics.

By making it easier to create quantitative conceptual diagrams, this research ultimately aims to improve the visual presentation of quantitative ideas everywhere. In classrooms, scientific papers, seminars, and websites, I hope that more visual diagrams will be used to better convey the quantitative ideas in any topic. Some have noted that Graph Sketcher also makes it easier than ever to create lies: professional-looking yet false data presentations. But such “lying with statistics” is nothing new. If quantitative diagrams are as easy to produce as text, perhaps people will come to look at them more critically, the same way they already do with words from unknown sources. And if quantitative diagrams are ubiquitous, perhaps society will come to better understand the quantitative concepts they depict.

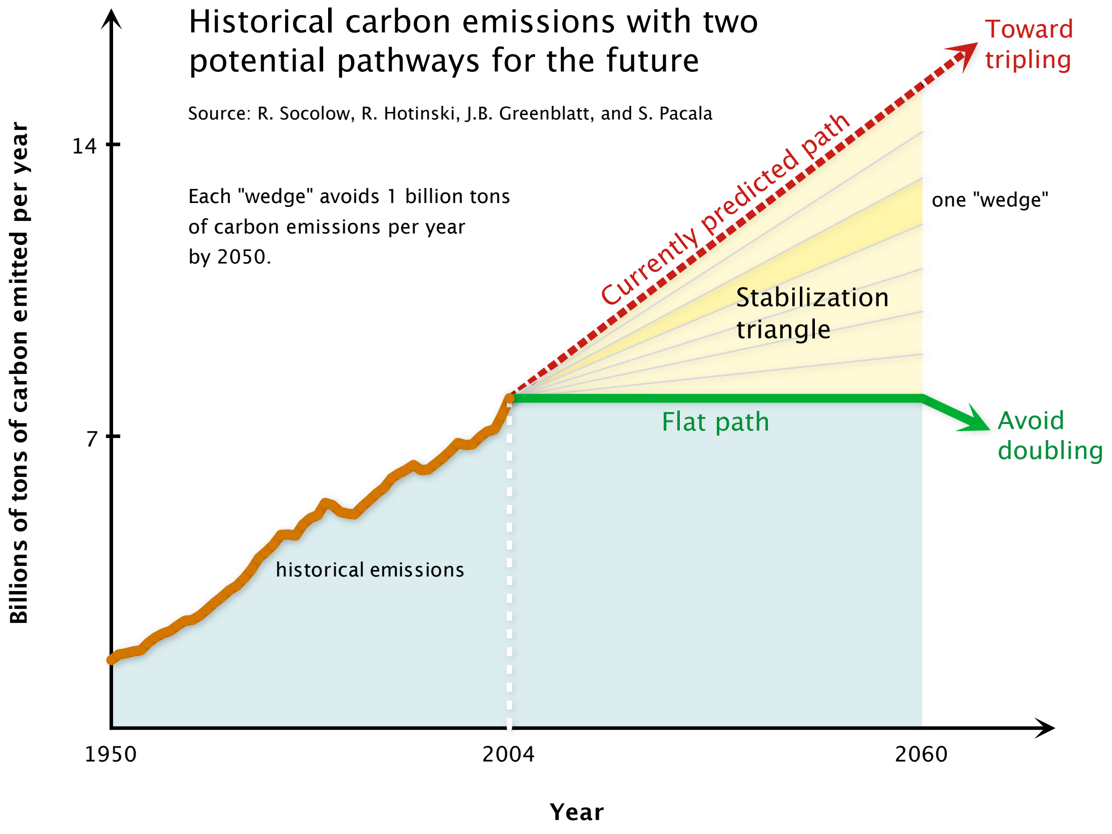


Figure 7-1: The interfaces described in this thesis make it possible to quickly and easily create beautiful quantitative conceptual diagrams such as this one.

7.1 Future Work

It is likely that the techniques I have presented will become more important over time as collaborative work demands that visualizations are computerized, shared, and contain an increasing number of contextual annotations. In addition, the more interactive and manipulable information graphics become, the more important it will be to dynamically maintain meaningful positioning of quantitative conceptual components. My evaluation of Graph Sketcher's use in the wild already shows the wide applicability of the interface in many disciplines. However, a longitudinal study of the graphs that users produce over time with and without the tool could also determine the tool's affect on their diagrams' creativity, clarity, and precision.

Many directions exist for enhancing Graph Sketcher's feature set further. I am still in the process of fully implementing curves that have multiple interior interpolation points. I would like to add more visualization types, such as logarithmic axes and pie charts; incorporate more statistical annotations such as box plots and histograms; implement some of Tufte's [38] suggestions for more elegant axes, bar charts, and labeling techniques; and provide the ability to layer multiple graphs next to or on top of each other. It would also be interesting to extend these interaction techniques to

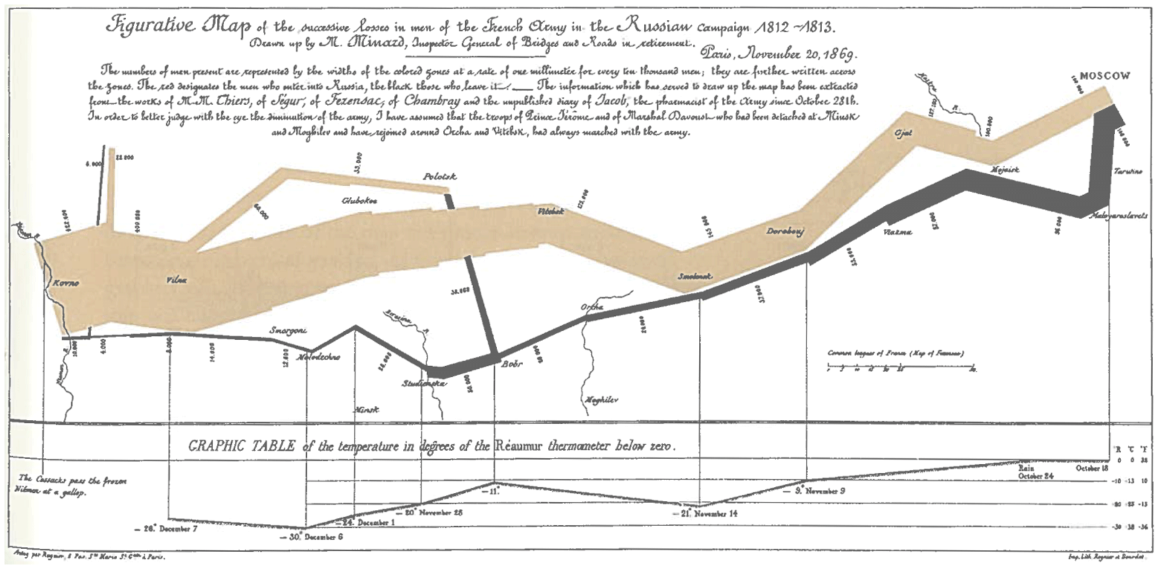

Figurative Map of the successive losses in men of the French Army in the Russian campaign 1812-1813.
Drawn up by M. Minard, Inspector General of Bridges and Roads in retirement.
Paris, November 20, 1869.
MOSCOW
GRAPHIC TABLE of the temperature in degrees of the Réaumur thermometer below zero.
Kovno
Vilna
Polotzk
Smolensk
Moscow

Figure 7-2: Tufte's [38] English translation of Minard's celebrated 6-factor depiction of Napoleon's doomed 1812 Russian campaign. The original was drawn in 1869.

mathematical functions (I imagine dragging a function graph to change its m and b equivalents).

Last, I would like to extend Graph Sketcher to handle multi-dimensional representations such as points whose sizes and colors are based on data values. I note that even Minard's celebrated 6-factor depiction of Napoleon's doomed Russian campaign (Figure 7-2) does not show the actual geography of the return trip (drawn in black). Instead, Minard shifted the return trip downward to avoid overlapping with the light brown outbound path [30]. This is exactly the type of data-focused, visual manipulation that my interface approach was designed to support.

Bibliography

[1] Peter Agar and Kevin Novins. Polygon recognition in sketch-based interfaces with immediate and continuous feedback. In *GRAPHITE '03: Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 147–150. ACM, 2003.

[2] Thomas Baudel. A mark-based interaction paradigm for free-hand drawing. In *UIST '94: Proceedings of the 7th annual ACM symposium on User interface software and technology*, pages 185–192. ACM, 1994.

[3] Eric A. Bier and Maureen C. Stone. Snap-dragging. *SIGGRAPH Comput. Graph.*, 20(4):233–240, 1986.

[4] Stuart K. Card, Thomas P. Moran, and Allen Newell. The keystroke-level model for user performance time with interactive systems. *Commun. ACM*, 23(7):396–410, 1980.

[5] Microsoft Corp. Excel:mac. Desktop software, 2008.

[6] Jerry Alan Fails Dan R. Olsen Jr., Trent Taufer. Screencrayons: Annotating anything. In *Proc. UIST '04*, 2004.

[7] Barry Fowler and Richard Bartels. Constraint-based curve manipulation. *IEEE Comput. Graph. Appl.*, 13(5):43–49, 1993.

[8] Michael Gleicher. A graphics toolkit based on differential constraints. In *Proc. UIST '93*, pages 109–120. ACM, 1993.

[9] Michael Gleicher and Andrew Witkin. Drawing with constraints. *The Visual Computer*, 11(1):39–51, 1994.

[10] Gene Golovchinsky and Laurent Denoue. Moving markup: Repositioning freeform annotations. In *Proc. UIST '02*, 2002.

[11] Omni Group. Omnigraffle 5. Desktop software, 2008.

[12] Jeffrey Heer, Fernanda B. Viégas, and Martin Wattenberg. Voyagers and voyeurs: supporting asynchronous collaborative information visualization. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 1029–1038. ACM, 2007.

[13] Takeo Igarashi, Satoshi Matsuoka, Sachiko Kawachiya, and Hidehiko Tanaka. Interactive beautification: a technique for rapid geometric design. In *Proc. UIST '97*, pages 105–114. ACM, 1997.

[14] Adobe Systems Inc. Illustrator cs3. Desktop software, released 2007.

[15] Apple Inc. iwork 2008. Desktop software, 2008.

[16] Jr. Joseph J. LaViola and Robert C. Zeleznik. Mathpad2: a system for the creation and exploration of mathematical sketches. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 432–440. ACM, 2004.

[17] Celine Latulipe, Stephen Mann, Craig S. Kaplan, and Charlie L. A. Clarke. symspline: symmetric two-handed spline manipulation. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 349–358. ACM, 2006.

[18] Jock Mackinlay. Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics*, 1987.

[19] N. Gregory Mankiw. *Principles of Economics*. Harcourt College Publishers, second edition, 2001.

[20] Paul Michalik, Dae Hyun Kim, and Beat D. Bruderlin. Sketch- and constraint-based design of b-spline surfaces. In *SMA '02: Proceedings of the seventh ACM symposium on Solid modeling and applications*, pages 297–304. ACM, 2002.

[21] Yasser F. O. Mohammad and Toyoaki Nishida. Naturaldraw: interactive perception based drawing for everyone. In *IUI '07: Proceedings of the 12th international conference on Intelligent user interfaces*, pages 251–260. ACM, 2007.

[22] Brad A. Myers, Jade Goldstein, and Matthew A. Goldberg. Creating charts by demonstration. In *CHI '94: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 106–111. ACM, 1994.

[23] OpenOffice.org. Open office 2. Desktop software, 2008.

[24] Tom Y. Ouyang and Randall Davis. Recognition of hand drawn chemical diagrams. In *Proceedings of AAAI*, pages 846–851, 2007.

[25] Robert S. Pindyck and Daniel L. Rubinfeld. *Microeconomics*. Prentice Hall, Upper Saddle River, NJ, 2001.

[26] Sviataslau Pranovich, Jarke J. van Wijk, and Kees van Overveld. The kite geometry manipulator. In *CHI '02: CHI '02 extended abstracts on Human factors in computing systems*, pages 764–765. ACM, 2002.

[27] Key Curriculum Press. Fathom dynamic data software 2.1. Desktop software, 2008.

[28] Roope Raisamo. An alternative way of drawing. In *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 175–182. ACM, 1999.

[29] Richard Rasala. *Explicit cubic spline interpolation formulas*, pages 579–584. Academic Press, Inc., Orlando, FL, 1990.

[30] Steven F. Roth, John Kolojejchick, Joe Mattis, and Jade Goldstein. Interactive graphic design using automatic presentation knowledge. In *CHI '94: Proceedings of the SIGCHI conference on Human factors in computing systems*, 1994.

[31] Kathy Ryall, Joe Marks, and Stuart Shieber. An interactive constraint-based system for drawing graphs. In *Proc. UIST '97*, pages 97–104. ACM, 1997.

[32] Tevfik Metin Sezgin, Thomas Stahovich, and Randall Davis. Sketch based interfaces: early processing for sketch understanding. In *PUI '01: Proceedings of the 2001 workshop on Perceptive user interfaces*, pages 1–8. ACM, 2001.

[33] Ben Shneiderman. Why not make interfaces better than 3d reality? In *IEEE Computer Graphics and Applications*, pages 12–15, 2003.

[34] Robert Socolow, Roberta Hotinski, Jeffery B.Greenblatt, and Stephen Pacala. Solving the climate problem: technologies available to curb co2 emissions. *Environment*, 46(10):8–19, 2004.

[35] Thomas F. Stahovich. Segmentation of pen strokes using pen speed. In *AAAI Fall Symposium Series 2004: Making Pen-Based Interaction Intelligent and Natural*, 2004.

[36] I. Sutherland. *Sketchpad: a man-machine graphical communication system.* Phd thesis, Massachusetts Institute of Technology, Cambridge, MA, 1963.

[37] Tibco. Spotfire 2.1. Desktop and online software, 2008.

[38] Edward Tufte. *The Visual Display of Quantitative Information.* Graphics Press, Cheshire, Conn., 1983.

[39] WaveMetrics. Igor pro 6. Desktop software, 2008.

[40] Gene Zelazny. *Say it with charts: the executive's guide to visual communication.* McGraw-Hill, 4th edition, 2000.