VizStudio: A Visualization Tool for Study in the Psychology of Trading

by

John Gu

Submitted to the Department of Electrical Engineering and Computer Science
on July 25, 2002, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

This thesis describes the design and implementation of VizStudio, a visualization tool developed to aid research into the psychology of trading. Developing VizStudio involved overcoming many challenges. A rigorous study into the psychology and emotion of trading involves capturing many sources of data, including records on trader physiology and performance, market movements, and video input to name a few. Moreover, new sources of data may be recorded, and additional data maybe derived from existing data. VizStudio is designed not only to integrate multiple sources of data into a single display, but to be easily extensible so as to accommodate the addition of new sources and types of data. In addition, VizStudio is required to process many sources and often large volumes of data. Thus, it is efficient in both computation and memory usage. Lastly, VizStudio applies principles of sound graphical user interface design to allow researchers to easily navigate through its features and menus, increasing its usability and overall value.

Thesis Supervisor: Andrew W. Lo
Title: Harris and Harris Group Professor

Thesis Supervisor: Dmitry Repin
Title: Postdoctoral Associate
Acknowledgments

I would like to take this opportunity to thank my thesis supervisors, Andrew Lo and Dmitry Repin, for their invaluable guidance and advice on this thesis. I also want to thank Timo Burkard for providing me the administrative knowledge involved in completing the thesis. Finally, I would like to thank the brothers of Theta Xi who provided a much-needed distraction from my thesis at times throughout my research.
Contents

1 Introduction .......................................................... 9

2 Related Work .......................................................... 13
  2.1 Existing Software ................................................. 13
  2.2 Relevant Research ................................................ 14

3 Experimental Framework ............................................. 15
  3.1 Overview ............................................................ 15
  3.2 Previous Research ................................................ 16
  3.3 Data Collected ...................................................... 17
  3.4 Data Collection Methods ......................................... 17
    3.4.1 Physiological Data Collection ............................. 18
    3.4.2 Financial Data Collection .................................. 21

4 Functional and Design Requirements ............................... 23
  4.1 Functional Requirements ......................................... 23
    4.1.1 Time Series and Video Display ............................ 24
    4.1.2 Time Synchronization and Management .................... 24
    4.1.3 Saving/Loading User Configuration ......................... 25
  4.2 Design Requirements .............................................. 26
    4.2.1 Intuitive User Interface .................................... 26
    4.2.2 Modular Decomposition ..................................... 29
    4.2.3 Efficient Computation and Memory Usage .................. 30
5 Design and Implementation

5.1 Design ........................................... 33

5.2 Implementation .................................. 35
  5.2.1 Language Choice ............................... 35
  5.2.2 Implementation Overview ................. 36
  5.2.3 The Visualizer Interface .................. 41
  5.2.4 The VisualizerFactory ....................... 42
  5.2.5 The TimeKeeper ............................... 43
  5.2.6 Saving/Loading User Configuration Files . 45
  5.2.7 The FileSourceManager ..................... 46

6 Conclusion and Future Work .......................... 49
Chapter 1

Introduction

A longstanding controversy in economics and finance is whether financial markets are controlled by rational forces or dominated by the emotions and psychological biases of market participants. Most economists subscribe to the “Efficient Market Hypothesis” [26]. The hypothesis argues that market security prices reflect all available information, thus making the forecasting of future prices impossible. The logic behind the Efficient Market Hypothesis is intuitively appealing. Security prices are determined by a large number of rational, well-informed traders attempting to forecast future prices based on past and expected future events. Any informational advantage that a trader obtains may enhance his or her ability to predict future price movements, giving every trader a strong incentive to obtain complete information on which to trade. The result is that any informational advantage will be short-lived. While short-term profit opportunities may exist, in the long-term, prices will accurately reflect all available information.

Critics of Efficient Market Hypothesis argue that investors are people whose judgments are influence by emotions and who exhibit irrational and well-documented psychological biases. These biases include overconfidence [1, 11, 13], overreaction [9], herding [15], psychological accounting [34], mis-calibration of probabilities [16], regret [3, 6], and loss aversion [23, 29, 33]. The critics contend that because these psychological biases are consistently exhibited by a large number of market participants, the market will systematically misprice securities, making possible the existence of a predictable market. Although no clear alternative to the Efficient Market Hypothesis has yet to emerge, a growing num-
ber of economists and financial professionals have began using the term “behavioral economics” and “behavioral finance” to differentiate themselves from the mainstream [28]. These critics point to the spectacular rise and fall of the U.S. stock market in the past few years as indicative of investor irrationality and market inefficiency. They also point to research by the likes of Kahneman and Tversky [33, 34] to illustrate the existence of behavioral and emotional biases.

As an example of Kahneman and Tversky’s research, the following problem was presented to a group of subjects:

You are given $1000 dollars. You must now choose between the following options:

A. A sure gain of $500.

B. A 50% chance to gain $1000, and a 50% change to gain nothing.

Another group of subjects was presented with a similar problem:

You are given $2000 dollars. You must now choose between the following options:

A. A sure loss of $500.

B. A 50% chance to lose $1000, and a 50% chance to lose nothing.

Even though the two options are identical in payoff, Kahneman and Tversky found that in the first group 84% chose option A while in the second group 69% chose option B. These and similar findings have been replicated numerous times by different researchers.

While emotional influences may at first seem detrimental to rational decision making, recent research in cognitive neuroscience and financial economics have suggested that the two concepts may in fact be complementary [10, 18, 20, 24]. A systematic investigation of this question is currently underway by Andrew Lo and Dmitry Repin [19] at MIT’s Laboratory for Financial Engineering. Their research non-intrusively measures the physiological responses of financial securities traders during actual trading. These physiological responses include body temperature, respiration rate, heart rate (HR), blood volume pulse
(BVP), skin conductance (SCR), and electromyopgraphical data (EMG). From these measurements, the emotional states of the traders are inferred and analyzed in the context of market data and matched up to specific market events.

While statistical analysis and the resulting aggregate statistics can provide insight into the question at hand, obtaining an intuitive feel for the data is also important. For example, one might want to see roughly how a specific market event (e.g. a price dip) influences a trader's emotional state as manifested in his physiological responses. To this end, a visualization tool with the ability to synchronously display the trader's physiological and market data, as well as the video recording of the experimental scene, would be valuable, giving the researcher an extra dimension upon which to interpret the results of the experiment. Moreover, such a tool would enable better methods of statistics analysis to be developed, offering more meaningful results. This thesis describes the design and implementation of VizStudio, a visualization tool that will aid in the study of the psychology of trading and real-time risk processing.

The remainder of this thesis will be organized as follows. Chapter two will review some software packages similar to VizStudio as well as some related research. Chapter three will describe the experimental setup of which VizStudio is a part, providing a context in which to discuss the functional and design requirements of the tool in Chapter four. Chapter five will describe the actual design and implementation of VizStudio. Finally, Chapter six will describe possible improvements and extensions to VizStudio and offer concluding remarks.
Chapter 2

Related Work

Currently, no known existing software provides the same functionality that VizStudio will provide when implemented. In this chapter, we look at some software packages that provides functionality similar to what will be provided by VizStudio, and discuss the shortcomings of these software packages. We also examine some research that will aid in the design and implementation of VizStudio.

2.1 Existing Software

Currently, two software packages are used at the Laboratory for Financial Engineering for visualizing the physiological data of traders. The first is the BioGraph (version 2.1) biofeedback software from Thought Technology, Ltd. This software came packaged with the physiological data acquisition hardware, and displays the physiological data being measured. Unfortunately, the visual interface of BioGraph is not very intuitive, and the software tends to crash often on the Windows 2000 machine that it runs on. Furthermore, BioGraph does not give the user the ability to display multiple types of data such as market security prices or other financial data. The second piece of software is the Mind-body Physiology from J&J Engineering, Inc. This software also has the shortcoming that it is designed purely for the display of physiological data, and cannot handle alternative data types. Moreover, the software does not provide an easy mechanism for the user to configure the display of physiological information. For example, the user cannot easily change the labeling of a
signal, nor can he easily change the color in which a signal is displayed.

2.2 Relevant Research

The design of VizStudio can be guided by research in many disciplines. First, VizStudio will integrate the synchronous display of several different types of data. As such, VizStudio should have a highly effective and intuitive graphical user interface (GUI). Many articles and books [14, 21, 22, 25] have been written on the topic of effective GUI design, and will be consulted. Another relevant field of research is in effective information display and data visualization. Prof. Edward Tufte of Yale University has published a number of books [30, 31, 32] on the effective visual display of quantitative information, which should help in the design of VizStudio as it will be responsible for managing the display of large amounts of quantitative information. A third relevant field of research is the modular decomposition of large software systems. Many techniques, design principles, and design patterns have been developed to help modularize code and reduce the number of cross-module dependencies [12, 27, 35]. The developments in these three fields as they relate to the design of VizStudio will be explored in Chapter four.
Chapter 3

Experimental Framework

In this chapter, we present an overview of the research that is currently underway at MIT's Laboratory for Financial Engineering. An understanding of this research is crucial to understanding the functional and design requirements that VizStudio should satisfy. We examine the purpose of the research, some previous related research, the type of data collected, and the data collection techniques.

3.1 Overview

VizStudio is intended to aid research studying the link between emotion and rationality in decision making by measuring the real-time psychophysiological characteristics of professional securities traders during live trading. For this purpose, financial securities traders are ideal subjects for several reasons [19]. Because the basic functions of securities trading involve frequent decisions concerning the tradeoffs between risk and reward, traders are almost continuously engaged in decision making under uncertain conditions. The study can thus be conducted in vivo and with minimal interference to and, therefore, contamination of the subjects' natural motives and behavior. Traders are typically provided with significant economic incentives to avoid many of the biases that are often associated with irrational investment practices. Moreover, they are highly paid professionals that have undergone a variety of training exercises, apprenticeships, and trial periods through which their skills have been finely honed. Therefore, they are likely to be among the most rational
decision makers in the general population, hence ideal subjects for examining the role of emotion in rational decision-making processes. Finally, due to the real-time nature of most professional trading operations, it is possible to construct accurate real-time records of the environment in which traders make their decisions, i.e., the fluctuations of market prices of the securities they trade. With such real-time records, market events such as periods of high price-volatility can be matched with synchronous real-time measurements of physiological characteristics.

### 3.2 Previous Research

Previous studies have focused primarily on time-averaged (over hours or tens of minutes) levels of autonomic activity as a function of task complexity or mental strain. For example some experiments have considered the link between autonomic activity and driving conditions and road familiarity for non-professional drivers [5], and the stage of flight (take-off, steady flight, landing) in a jetfighter flight simulator [17]. Recently, the focus of research has started to shift towards finer temporal scales (seconds) of autonomic responses associated with cognitive and emotional processes. Another set of influential experiments in this area was conducted in the broad context of an investigation of the role of emotion in decision making processes [8]. In one of these experiments, skin conductance responses were measured in subjects involved in a gambling task [2]. The results indicated that the anticipation of the more risky outcomes led to more skin conductance responses than of the less risky ones. The brain circuitry involved in anticipating monetary rewards has also been localized [4]. Recent experiments [7] support the significance of autonomic responses during risk-taking and reward-related behavior. They provide more details on the emotional correlates of autonomic physiological responses, and also claim the possibility of discriminating the activity patterns related to changing versus continuing the behavior based on the immediate gain/loss history in a gambling task.
3.3 Data Collected

To measure the emotional responses of the subjects during their trading activities, the research focuses on indirect manifestations through the responses of the autonomic nervous system (ANS). The ANS is responsible for the regulation of internal states that are mediated by emotional and cognitive processes. Its responses are relatively easy to measure since many of them can be measured non-invasively from external body sites without interfering with cognitive tasks performed by the subject. ANS responses occur on the scale of seconds, which is essential for investigation of real-time risk processing and decision making. A pilot study involving ten professional traders has been conducted. The pilot study focused on five physiological characteristics: skin conductance (SCR), cardiovascular data including blood volume pulse (BVP) and heart rate (HR), electromyographical data (EMG), respiration rate and body temperature. The placement of sensors for measuring these physiological responses is shown in Figure 3-1. The real-time market pricing data used by the trader are also recorded synchronously with the physiological responses. Future studies will add electroencephalogram data (EEG) to this list of physiological characteristics, from which a number of derivative characteristics such as the power spectra can be extracted. Future studies will also incorporate eye tracking sensors and digital video of the trader and the trader's computer screen. This information will allow researchers to discriminate between physiological responses that result from market events and those produced by outside stimuli.

3.4 Data Collection Methods

This section describes the data collection techniques used to collect the data described in the previous section. Figure 3-2 shows a typical setup for the measurements of real-time physiological response of financial traders during live trading sessions. Real-time market data used by the traders are recorded synchronously and subsequently analyzed together with the physiological response data.
3.4.1 Physiological Data Collection

A ProComp+ data-acquisition unit and BioGraph (Version 2.1) biofeedback software from Thought Technology, Ltd. were used to measure physiological data for all subjects. All six sensors were connected to a small control unit with a battery power supply, which was placed on each subject’s belt and from which a fiber-optic connection led to a laptop computer equipped with real-time data acquisition software BioGraph (see Figures 3-1 and 3-2). Each sensor was equipped with a built-in notch filter at 60 Hz for automatic elimination of external power line noise. The sampling rate for all data collection was fixed at 32 Hz. All physiological data except for respiration and facial EMG were collected from each subject’s non-dominant arm. SCR electrodes were placed on the palmar sites, the BVP photopleysymographic sensor was placed on the inside of the ring or middle finger, the arm EMG triode electrode was placed on the inside surface of the forearm, over the flexor digitorum muscle group, and the temperature sensor was inserted between the elastic band placed around the wrist and the skin surface. The facial EMG electrode was placed on a masseter muscle, which controls jaw movement and is active during speech or any other
Figure 3-2: Typical Setup for Experimental Data Collection
activity involving the jaw. The respiration signal was measured by chest expansion using a sensor attached to an elastic band placed around the subject’s chest. An example of the real-time physiological data collected over a two-minute interval for one subject is given in Figure 3-3.

![Figure 3-3: Typical Physiological Response Data](image)

The entire procedure of outfitting each subject with sensors and connecting the sensors to the laptop required approximately five minutes, and was often performed either before the trading day began or during relatively calm trading periods. Subjects indicated that presence of the sensors, wires, and a control unit did not compromise or influence their trading in any significant manner, and that their workflow was not impaired in any way. This was verified not only by the subject, but also by their supervisors. Given the magnitudes of the financial transactions that were being processed, and the economic and legal responsibilities that the subjects and their supervisors bore, even the slightest interference with the subjects’ workflow or performance standards would have caused the supervisors or the subjects to terminate the sessions immediately. None of the sessions were terminated prematurely.
3.4.2 Financial Data Collection

At the start of each session, a common time-marker was set in the biofeedback unit and in the subject’s trading console (a networked PC or workstation with real-time data feeds such as Bloomberg and Reuters) and software installed on the trading console (MarketSheet, by Tibco, Inc.) stored all market data for the key financial instruments in an Excel spreadsheet, time-stamped to the nearest second. The initial time-markers and time-stamped spreadsheets allowed us to align the market and physiological data to within 0.5 seconds of accuracy. Figure 3-4 displays an example of the real-time financial data – the euro/US-dollar exchange rate – collected over a 60-minute interval.

![Figure 3-4: Typical Real-Time Market Data](image)

With this understanding of the research that VizStudio is intended to aid, we can now proceed to discuss the functional and design requirements of VizStudio.
Chapter 4

Functional and Design Requirements

VizStudio should satisfy two sets of requirements. The first is a collection of functional specifications that will enhance the value and usability of the tool. These specifications document different functionalities that VizStudio will provide to the user. The second is a set of design principles that VizStudio should follow. These principles will ensure that VizStudio is easily extensible and adaptable to future needs. Rather than specifying what the capability of VizStudio are, these design principles will dictate how VizStudio should go about providing these capabilities.

4.1 Functional Requirements

Based on the discussion in the previous chapter, VizStudio should satisfy three major functional requirements. The first is the ability to display physiological and market time series data, as well as digital media capturing the behavior of the traders during the experimental sessions. The second is the ability to synchronize and manage the progression of time among the various displays. Lastly, VizStudio should offer the ability to save and load user configuration information to enhance its usability.
4.1.1 Time Series and Video Display

The most basic functional requirement that VizStudio should provide is the ability to display the time series data collected during the experimental sessions. This includes the physiological signals measured from the traders and the financial data recorded during the live trading sessions. However, other time series data, such as the trader's profit and loss and the trader's transactions, may also need to be displayed. Each time series can be presented in its own display panel, or be displayed with another time series in the same panel. For example, while we would display the BVP and EEG data of a trader in separate display panels, we should display the ask and bid prices for a particular stock in the same display panel. Each display panel should allow for the ability to easily adjust the length of the time interval and the range of the data being displayed. The ability to adjust the background and foreground colors of the display panel, and the ability to change the color in which a particular type of data is displayed will also be useful.

Another basic functionality that will be immensely useful is the ability to play digital media capturing the behavior of the traders during the experimental sessions. Plans are being made not only to record the actions of the traders during the trading sessions, but also their computer screens along with eye tracking data. These digital recordings will allow us to determine whether the physiological responses measured from a trader resulted from market events observed on a computer screen or some other outside stimuli. Consequently, we can further process the physiological data measured from the trader to identify only the relevant physiological responses.

4.1.2 Time Synchronization and Management

Time is the single unifying variable across all the different types of data collected during an experimental trading session. As such, the ability to synchronize and manipulate the progress of time among the different data displays is extremely important. Properly synchronizing the different types of data in time involves taking into account a number of issues. First, different types of data often will not be recorded starting at the same point in time. For example, in a particular experimental trading session, the researcher may start
recording physiological data a length of time prior to recording the market data. As such, VizStudio must provide a mechanism for the researcher to input this information and then act on it accordingly. Second, different types of data are recorded at different frequencies. Currently, physiological data is recorded 32 times every second, while market data is recorded roughly once a second. VizStudio must be able to deal with these differences when synchronizing the display of the data in time.

VizStudio should also offer the ability to manage the progress of time among the different data displays. Some basic time management functionality include the ability to position the data displays at a particular point in time, the ability to progress time forward and backward at varying rates of speed, as well as the ability to stop the progress of time. These functionalities will enable the user to identify key events quickly by scrolling the data displays, and then return to these events later to study them in more detail. Of course, all the time management functionality must operate while maintaining synchronized time between the different data displays.

4.1.3 Saving/Loading User Configuration

When VizStudio is instructed to open a collection of data for the first time, the user will probably spend a significant amount of time configuring VizStudio to display the data. For example, the user will have to specify to VizStudio the location of the physiology data file, which data columns in the file are relevant, what each data column should be labeled, the background and foreground colors of the data display, in what color to display each data column, whether some data columns should be displayed in the same panel, and so forth. The user will then have to repeat the same configuration process for the market data. After this, the user will need to enter information that will help VizStudio synchronize the display of the different data types. For example, the user will have to specify that physiology data was recorded at a rate of 32 times per second, while market data was recorded only once a second. The user will then specify that the recording of market data started 15 seconds after that of physiological data, and that the digital video of the trader began recording 7 seconds later. The user will then resize and reposition the data displays to suit his liking.
Going through this entire configuration process each and every time would be extremely inefficient. Instead, VizStudio should offer an option to save these configuration settings, and load them later without the need to reconfigure. Similarly, each data display, for the physiological data for example, may offer the option to save a template file, so that the configurations for that particular data file can be easily applied to another data file.

### 4.2 Design Requirements

Aside from functional requirements, VizStudio should satisfy a number of design requirements that will make it more efficient and extensible piece of software. First, VizStudio should sport an intuitive user interface. Second, the code structure of VizStudio should be highly modular and have few cross-module dependencies. Lastly, VizStudio should be efficient in computation and memory usage.

#### 4.2.1 Intuitive User Interface

VizStudio combines many sources of data into one integrated display. Consequently, the usability of VizStudio will depend largely on the quality of the user interface it presents. If managing the display of the different data sources become too much of a chore, the usefulness of VizStudio will be severely limited. As such, the design of VizStudio’s user interface component must incorporate the principles of effective GUI design. The first of these principles is to remember and obtain constant feedback from the user. Often times, developers forget the user when designing a user interface. They write the program based on what they know, not what the user knows. Much of the success of Intuit’s flagship product Quicken can be attributed to the developers’ extensive interaction with potential users and incorporating the feedback into the design of Quicken’s user interface. Similarly, the design of VizStudio should be guided by feedback from its users.

The second principle of effective GUI design is to empower the user. GUI designers will often attempt to control user navigation by dynamically graying out menu items or controls. controlling the user is completely contradictory to the design of event-driven software, where the user should dictate when and what events will occur. Thus, VizStudio
should be designed to allow the user to access the application in as many ways as possible.

The third design principle is to avoid exposing too many features at the top level. Instead, only frequently used features should be exposed, unobscured by other less used features. The infrequently used features are probably more confusing, and should instead be tucked away behind a dropdown menu or be implemented in a popup dialog. To further illustrate this principle, we can look at the user interface presented by a recent VCR. Such a VCR will only have a few buttons readily available on the faceplate: play, fast forward, rewind, stop, pause, eject and power. The buttons accessing the other functions of the VCR will be located behind a sliding or dropdown panel. VizStudio should follow the same design principle in constructing its interface. Not only will the resulting interface be cleaner and more pleasant visually, it will be easier and more efficient to use.

The forth principle of effective user interface design is consistency. Consistency in words is key to conveying a message clearly to the user. For example, rather than using different words like “item,” “product” and “merchandise” to refer to the same concept, one word should be chosen and used consistently throughout. VizStudio should apply the same consistency in word use. Consistency in locating control elements is also important. When the user navigates to a certain location looking for a particular button or menu item, it should be there. Placing controls where the user expects them to be will increase the overall efficiency of VizStudio. Along the same lines, VizStudio should provide keyboard support in a manner consistent with existing software conventions. For example, Ctrl-S should be mapped to save, Ctrl-C to close, and Ctrl-X to exit. Using existing conventions help to flatten the learning curve for the user, allowing the user to quickly gain proficiency in using VizStudio.

The last principle of effective user interface design is to provide visual feedback to the user. We have all probably found ourselves staring at an hourglass on the screen while waiting for an operation to complete. This is the result of poor visual feedback. When possible, VizStudio should provide the user with text messages and progress bars indicating the amount of remaining work. This will keep the user informed about the state of the application.

Outside of the above GUI design principles, the effective display of quantitative data
will also help in constructing an intuitive user interface. The individual VizStudio data displays will have the task of presenting the data in a way that is concise and easy to interpret. The ideal presentation would compress the most amount of information into the least amount of space without any loss of clarity. A quintessential example of such a presentation is Charles Minard’s map of Napoleon’s Army’s Russian campaign. Minard’s map is show in Figure 4-1, and compresses 6 different types of data into the single display. The eventual display of the trader physiology and financial data should strive to achieve the same level of richness and clarity as exemplified by Minard’s map. In addition, the presentation of data should take advantage of dynamic and interactive features available in a computer-based display as compared to a static medium such a paper.

Figure 4-1: This map displays six variables in a single graph: the size of the army indicated by the width of the gray and black bands, the north-south and east-west location of the army, the direction of the army’s movement (gray for advancing and black for retreating), temperature and finally various dates and events.
4.2.2 Modular Decomposition

The code structure of VizStudio should be highly modularized. Code modularity has long been recognized in software engineering as being the foundation for good design. Dividing a program into smaller parts offers the following advantages.

**Faster Development** A modular design allows many programmers to work concurrently on the program, shortening development time.

**Simpler Testing** Building a program in smaller parts allows incremental testing and verification of correctness. When a bug is discovered, the process of uncovering the reasons for the bug is simplified.

**Cleaner Design** Breaking a program down into smaller parts forces the programmer to develop clean interface layers between the different components. Thus, the overall complexity of the application is reduced. The ability to reason about the overall correctness of the program is reduced to reasoning about the correctness of the individual components, and then reasoning about how the components interact with each other. The ability to better reason about the correctness of the program, coupled with incremental testing, will significantly reduce the amount of debugging time needed to complete the program.

**Reusability** When building a program, it is often possible to break the program down into parts that can be reused many times, not only in the immediate application, but in other applications as well. Modularity facilitates this reuse process.

**Easier to Maintain** Any useful program will undergo changes and extensions over the course of its lifetime. A modular design allows the change to be localized to small parts, making the process of making and validating the change easier. For example, so long as a particular module’s interface is maintained, its implementation can be changed entirely without modifying the remainder of the program.

In the case of VizStudio, modularity is particularly important. Not only can new data types be added that will need visual display, existing data can be displayed in different
ways. For example, rather than just displaying the time series of the physiological data, we may choose instead to display some aggregate statistics calculated based on the data. Another more concrete example would be to display the power spectra of EEG recordings. VizStudio should provide simple hooks to accommodate the addition of new displays with various sampling rates, and the synchronization of these displays with the rest of the system.

To realize the full benefits of using a modular design, VizStudio must also reduce the number of cross-module dependencies. A module $A$ depends on another module $B$ if $A$'s behavior depends on $B$'s behavior. When a program contains too many cross-module dependencies, the ability to reason about the overall correctness of the program is diminished. Moreover, the ability to make localized changes is also reduced.

A wide range of techniques and design patterns have been developed to help programmers with the task of designing modular programs and reducing cross-module dependencies. For example, one technique calls for the use of specifications and interfaces to abstract functionality from implementation. Another recognizes the power of extra levels of indirection in decoupling highly interdependent modules. Design patterns include the Singleton, which ensures that a class can have only one instance, and the Facade, which implements a simple interface to access the functionalities of an otherwise complex subsystem.

### 4.2.3 Efficient Computation and Memory Usage

The third design requirement of VizStudio is computational and memory efficiency. Computational efficiency is necessary because VizStudio may be required to support many visual displays at the same time. For example, VizStudio may need to display many types of physiology and market data, digital video, and some derivative data, e.g. the power spectra of an EEG recording, all at the same time. If any visual display takes up excessive computational resources, the performance of the other visual displays will be adversely affected. Memory efficiency is necessary because the data files that VizStudio works are large. For example, a typical experimental session with a trader lasts for about an hour. Physiological data is recorded at a frequency of 32 hertz, giving a total of $32 \times 3600 = 30$
115200 samples. Currently, six types of physiological data is recorded, with each sample treated as a double precision floating point number consuming 8 bytes of memory. A quick calculate will show that the amount of memory required to load a typical physiology data file will be 6 megabytes. While this is not an amazingly large amount of memory, we must consider that many physiology files may be open at the same time. For example, it may be useful to have open a file containing the physiology data of the trader during a resting period, and another file containing the physiology data of the trader during trading. Outside of physiology data files, VizStudio will also have to work with market data file and digital video files. From this perspective, it is easy to see how system memory can be quickly exhausted.
Chapter 5

Design and Implementation

This chapter describes the design and implementation of VizStudio. The design of VizStudio, in terms of code architecture and user interface, was guided largely by the design requirements set forth in the previous chapter. The code structure is modular and minimizes the number of cross-module dependencies, all the while seeking to reduce system memory usage. The user interface was designed with the consultation of the user and is easy and efficient to use. The implementation provides all the functional requirements discussed previously and easy hooks to extend the system with new functionality.

5.1 Design

The overall design of VizStudio is very modular, allowing incremental testing and verification of correctness. Great care was taken to minimize cross-module dependencies, supporting localized changes and allowing the system to be extended easily. The first attempt at crafting a viable design for VizStudio focused on a high-level modular decomposition of the entire system, and is shown in Figure 5-1. This design introduces the concept of a Visualizer. A Visualizer can be thought of as a module that is responsible for presenting the visual display of one or more data sources. For example, one can imagine a simple Visualizer that displays the numerical values of a stock’s bid and ask price, or a sophisticated Visualizer that takes as input the EEG signal from a trader and displays the power spectrum of the signal. In this sense, each Visualizer can be considered a standalone application,
except that it operates within the framework defined by VizStudio.

Figure 5-1: First Attempt at Decomposing VizStudio into its Modular Components

Each Visualizer must adhere to a Visualizer Interface, through which its interactions with VizStudio GUI is completely defined. The VisualizerFactory contains knowledge of all the implemented Visualizers, and the VizStudio GUI uses the facilities provided by VisualizerFactory to instantiate Visualizers. In this way, any explicit knowledge of which Visualizers exist, as well as the specific implementation details of each Visualizer, are removed from the Visualizer GUI. In order to create another Visualizer, we would simply program the Visualizer so that it adheres to the Visualizer Interface, and register the Visualizer with the VisualizerFactory. No other changes would need to be made. In this way, cross-module dependencies are reduced, thus simplifying the overall complexity of the system. This design provides a very general and flexible framework in which individual Visualizers can be optimized for efficiency without the loss of flexibility or generality.

Unfortunately, this first design has two major drawbacks. First, this design is not memory efficient. Because Visualizer do not have knowledge of each other, when two Visualizers share the same data source, the data is loaded into memory twice, or possibly even
more times if more Visualizers shared the same data source. Second, the design lacks a mechanism for synchronizing time between the different Visualizers. Because the design implies that the Visualizers do not know of each other’s existence, we need to provide some mechanism by which they can all synchronize regardless.

The resulting design correcting these drawbacks is shown in Figure 5-2. To improve efficiency in memory usage, we insert a memory management module in between the different Visualizers and the data sources. All accesses to the data sources must pass through this module. If a requested data source is not in memory, the module pulls the data from disk into memory and returns the memory reference to the requesting Visualizer. If another Visualizer requests the same data source, the module would recognize that the data already exists in memory and return the memory reference. When the data is no longer needed by any Visualizers, the data management module would remove it from memory, reclaiming the used resources.

The time management module acts as VizStudio’s internal clock. It ticks away at the specified rate, either forwards or backwards in time. It can be paused and stopped. All Visualizers are passed a reference to the same instance of the time management module, and thus all Visualizers can use the module to synchronize with each other.

5.2 Implementation

This section describes the implementation of VizStudio. First, the choice of Java as the development language is explained. We then give an overview of the different modules, and their respective functionalities before examining the major components of VizStudio in more detail.

5.2.1 Language Choice

VizStudio was implemented in Java, and totals slightly more than 4500 lines. The choice of development language was divided between Java and C++, both chosen for their object oriented design. Ultimately, Java was chosen because it is platform independent, and the large number of Application Programming Interfaces (API) it provides would shorten de-
5.2.2 Implementation Overview

The module dependency diagram (MDD) of the final implementation of VizStudio is shown in Figure 5-3. The main user interface component is implemented in the `VizStudio` container class, and serves as the main container inside which Visualizers are placed. As such, VizStudio supports a Multiple Document Interface (MDI). A screenshot of VizStudio is shown in Figures 5-4 and 5-5. `WindowMenu` and `MDIDesktopPane` are convenience classes to support having multiple documents placed inside the `VizStudio` container.
MDIDesktopPane also enables VizStudio to display scroll bars when parts of Visualizers move outside the visible area of the VizStudio container.

Currently, two Visualizers classes are implemented, TimeSeriesVisualizer and VideoPlayerVisualizer. Each Visualizer adheres to the Visualizer interface, and subclasses JInternalFrame. As previously discussed, the Visualizer interface provides a mechanism to remove knowledge of specific implementations of Visualizer from the VizStudio container. Instead, knowledge of the different implementations of Visualizer resides in VisualizerFactory. The VizStudio container uses VisualizerFactory to instantiate the Visualizers, after which its interactions with the instantiated Visualizers are confined to the functions defined in the Visualizer interface. Aside from being highly modular, this implementation provides a general and flexible framework in which each Visualizer can be made very specific and optimized for its intended task. Thus, flexibility and efficiency are no longer opposing goals as often is the case in software design.

A number of classes help to complete the implementation of TimeSeriesVisualizer. FileInputDialog and FileInputData help to obtain information from the user about which data file to load, which columns in the data file are relevant, and how to parse the data file. TimeSeriesInputDialog and TimeSeriesInputData help to obtain data display options from the user. For example, the foreground and background colors of the display panels, the color in which to display the different data columns, and the names given to a data column are all obtained using these two classes. TimeSeriesDisplayPanel implements the individual display panels inside which one or more data columns can be displayed. The implementation of VideoPlayerVisualizer is much simpler, and only uses VideoInputDialog and VideoInputData to get information from the user about which digital video file to load.
Figure 5.3: The Module Dependency Diagram of Actual VizStudio Code
Figure 5-4: VizStudio Information
Figure 5-5: A Screen Shot of VizStudio in Action
TimeKeeper acts as the internal clock to VizStudio, and is largely responsible for the task of synchronizing time between the different Visualizers. It implements functionalities of the time management module discussed in the previous section. In any running instance of VizStudio, there is only one instance of TimeKeeper, and it is passed to all Visualizers on creation. In fact, TimeKeeper operates as a singleton class, although it is not implemented as such. Finally, FileSourceManager, FileData, and DataSet implements the behavior of the memory management module discussed previously, and acts as a generalized data cache. When a Visualizer requests the data located in a particular file, FileSourceManager would load the data into memory if the data does not already reside there, and return the data to the requesting Visualizer. If the data do reside in memory, FileSourceManager would simply return the data without reloading. Any data no longer needed by a Visualizer are purged from memory. With this high-level overview of the implementation, we are ready to discuss the major components of VizStudio in greater detail.

5.2.3 The Visualizer Interface

The Visualizer interface defines a set of functions that each Visualizer must implement. The list is small, and includes only eight functions. These are listed below.

**Configuration File I/O Functions**

- public boolean readFile(BufferedReader fr);
- public boolean writeFile(BufferedReader fw);

**Time Management Functions**

- public void start(double rate);
- public void stop();
- public void pause();
- public void setTIme();

41
**Other Miscellaneous Functions**

    public boolean getUserConfiguration();
    public long getRunningTime();

When a Visualizer is first created, either getUserConfiguration or readFile must be called. Both functions return true if the intended operation is successful, and false otherwise. getUserConfiguration prompts the user for configuration information related to the display of the Visualizer, and readFile reads this information instead from a previously saved file. As the actual look of the Visualizer will depend on the information obtained by these two functions, one of them must have been called successfully before the Visualizer can be set visible. The function writeFile is the counterpart of readFile, and writes the user configuration information to the file linked to the BufferedWriter so that it can be read back later. getRunningTime returns the total duration of the data being displayed. This function is used by VizStudio to position the internal clock (TimeKeeper) of VizStudio based on the position of the scroll bar. Finally, the functions start, stop, pause and setTime are used to help synchronize the different Visualizers. Whenever a user presses a control button, such as Stop or Play, VizStudio iterates through the list of instantiated Visualizers, calling the associated function on each one. start is a generalized function that is passed a rate variable. rate specifies both the direction and rate at which time progresses. A rate of 1.0 implies normal playback, and a rate of -2.0 means playing the dataset backwards through time at twice the normal speed.

### 5.2.4 The VisualizerFactory

VisualizerFactory abstracts away knowledge of the different implementations of Visualizer from VizStudio. The class contains two static functions. The first is:

    String[] getVisualizers();

which returns a String[] array of the names of the different implemented Visualizers.
The second function is:

```java
Visualizer createVisualizer(String name, JFrame parent,
                            TimeKeeper timekeeper, StatusLabel label);
```

The `name` argument identifies which Visualizer to create, and should match one of the names returned by `getVisualizers()`. The `timekeeper` is a reference to the single `TimeKeeper` instance in the VizStudio application, and is passed to the Visualizer when it is created. The `label` argument is a reference to a label that the Visualizer can use to alert the user of certain events, or warn the user when an error occurs.

### 5.2.5 The TimeKeeper

One of the central mechanisms used for synchronizing time among the different Visualizers is the TimeKeeper. `TimeKeeper` functions as the internal clock for VizStudio. `TimeKeeper` is thread-safe, meaning that all of its functions are synchronized and acquires a lock on the instance of `TimeKeeper` when called. This is necessary as the functions of `TimeKeeper` are called from multiple competing threads. Internally, `TimeKeeper` keeps the variable `current_time`, initially set to 0, and uses a fixed-rate Java utility timer to update the `current_time`. When the `TimeKeeper` is first created, it is passed an argument `execution_rate`, in milliseconds, specifying the rate at which the `TimeKeeper` updates its internal clock. Every `execution_rate` milliseconds, `TimeKeeper` would progress its internal clock a certain amount of time, as determined by the `playback_rate`. The general algorithm is shown below:

```
current_time = current_time + execution_rate * playback_rate
```

If the `playback_rate` is 1.0, then the `TimeKeeper`'s internal time would progress at the same rate as real time. If the `playback_rate` is negative, then the `TimeKeeper`'s internal time would progress backwards in time. `TimeKeeper` offers the following public
void start(double playback_rate)
void pause()
void stop()
void setTime()
long getTime()
double getRate()
int getStatus()

The functions start, pause, stop and setTime are analogous to the same set of functions defined in the Visualizer interface. Whenever the user presses a control button, the appropriate function in TimeKeeper is called. The functions getRate and getTime return the playback_rate and current_time, respectively. The function getStatus returns the status of the TimeKeeper, which is either TimeKeeper.STARTED, TimeKeeper.STOPPED, or TimeKeeper.PAUSED.

It may seem redundant to use both TimeKeeper and the time management functions defined in the Visualizer interface to synchronized among the Visualizers. However, this is not the case. To illustrate, imagine a Visualizer being inserted into the VizStudio container while the time is running. The Visualizer would not be synchronized with the other Visualizers and will need to use the internal clock of TimeKeeper to orient itself to the correct time. This example illustrates that timing skew occasionally do occur among different Visualizers. Having a TimeKeeper around provides each Visualizer with the ability to correct for this skew. In fact, in the case of TimeSeriesVisualizer, the calls to the time management functions simply return without performing any task. Instead, TimeSeriesVisualizer regularly polls TimeKeeper for the internal current_time and behaves according to the value returned. Similarly, VideoPlayerVisualizer regularly uses the TimeKeeper’s internal clock to resynchronize itself with the rest of the system.
5.2.6 Saving/Loading User Configuration Files

The ability to save and load user configuration information is very important to increase the overall value and usability of VizStudio. VizStudio provides a generalized mechanism for allowing each Visualizer to save its configuration information. Inside every configuration file, keywords BeginVisualizer and EndVisualizer denote the beginning and end of the configuration information for a particular Visualizer. Following BeginVisualizer is the name of the Visualizer for which the subsequent configuration information is relevant. For example, a configuration file might look like this:

```
BeginVisualizer: Time Series Visualizer

<Configuration Information for Time Series Visualizer>

EndVisualizer

BeginVisualizer: Video Player Visualizer

<Configuration Information for Video Player Visualizer>

EndVisualizer
```

When loading and parsing a configuration file, VizStudio will encounter the keyword BeginVisualizer, and use the name of the Visualizer to create the appropriate Visualizer instance through VisualizerFactory. VizStudio will then pass the responsibility for parsing the configuration file to the newly created Visualizer by calling the readFile function. The Visualizer will then continue parsing the configuration file until it encounters the keyword EndVisualizer, upon which it returns parsing control to VizStudio. Similarly, when saving a configuration file, VizStudio opens the configuration file and calls writeFile on each currently active Visualizer. This mechanism allows each Visualizer to define its own configuration file format without any constraints, increasing the overall flexibility of the system.
5.2.7 The FileSourceManager

FileSourceManager implements the functionality of the memory management module described in the previous section. Internally, FileSourceManager does reference counting on the individual data columns in a data file. The module keeps count of how many distinct Visualizers reference a particular data column. Whenever any data column is no longer needed by a Visualizer, all references to the data in the column are removed, and the memory consumed by the data is reclaimed when the Java Virtual Machine performs garbage collection. The functions offered by FileSourceManager are all static, and are listed below.

```java
String checkFileSource(String filename, String nvalid,
                       String delimiter);
boolean openFileSource(String filename, String nvalid,
                       String delimiter, Object ref);
void closeFileSource(String filename, Object ref);
Vector getDataSet(String filename, Object ref);
```

Before we discuss the behavior of the individual functions of FileSourceManager, we must first introduce the concept of a valid string. A valid string specifies which columns of a data file are relevant to a Visualizer. A valid string contains only the characters ‘0’ and ‘1’, where ‘0’ denotes that a column is to be ignored, while ‘1’ denote a column is relevant. For example, a valid string of “11001” would be associated with a data file containing 6 columns, where the fifth and sixth columns are not needed. A numbered valid string not only specifies which columns of a file are needed, but how those columns are grouped together. A numbered valid string can contain all numeric characters, where a numeric character c greater than ‘1’ denotes that c data columns are relevant, and are to be grouped together. For example, a numbered valid string of “203” would be associated with a data file containing 6 columns, where the third column is ignored. The first two columns are considered to be grouped together, while the last three columns are considered to be
The function checkFileSource simply checks to see that filename exists, and that when the file is parsed with delimiter, the number of data columns matches the number of data columns implied by nvalid, the numbered valid string. If this is the case, the function will return null. Otherwise, it will return an error message. The function openFileSource takes the same arguments as checkFileSource, and also a referencing object ref. The object ref refers to the Visualizer that is requesting access to the data, and is used by FileSourceManager for reference counting. When a Visualizer wants to access data in a particular file, it calls openFileSource and passes itself as the reference. When it no longer needs the data, it calls closeFileSource and passes itself again as the reference.

FileSourceManager keeps track of which file is being referenced by which objects. It also keeps track of which columns in a particular file are being referenced via the nvalid numbered valid string argument. Internally, FileSourceManager converts each numbered valid string to a regular valid string containing only ‘0’ and ‘1’. To find which columns of a particular file are being referenced by at least one Visualizer, FileSourceManager iterates through the valid strings of all Visualizers referencing that file. Whenever a data column is no longer needed, all references to that data column are removed allowing it to be garbage collected.

To access the data in the file loaded by FileSourceManager, a Visualizer would call getDataSet and pass in the filename and itself as the referencing object ref. The argument ref is used to find the associated numbered valid string, which defines the structure of the Vector that is returned. For example, if a Visualizer called openFileSource with nvalid equaling “1202”, then the Vector returned by getDataSet would contain three elements, where the first element is an array of length one, and the remaining two elements are arrays of length two. In this way, FileSourceManager serves as an intermediate data cache between Visualizers and the data files, thus eliminating the need for replicating data in memory and improving the memory efficiency of VizStudio.
The remaining modules of VizStudio implements the TimeSeriesVisualizer and VideoPlayerVisualizer. As these modules are specific Visualizer implementations, knowledge of their inner circuitry is not necessary to understanding the general framework established by VizStudio and how to implement new Visualizers within this framework. Therefore, the remaining modules will not be described in further detail in this document.
Chapter 6

Conclusion and Future Work

The overall reported user experience with VizStudio has been very positive, and the design and implementation of VizStudio lends easily to future extensions. Nevertheless, VizStudio does exhibit some implementation flaws that should be corrected in later versions. The first change should be to re-implement TimeKeeper as a singleton class. Rather than having to pass the single instance of TimeKeeper around, each Visualizer can simply instantiate the TimeKeeper to obtain a reference to the single instance of the class.

The second change would be to revise the semantics of numbered valid string so that it is more general. Currently, only data columns next to each other can be considered to be grouped together. There is no way to express that data columns two and four should be grouped together, and data columns one and three should be grouped together. That being said, data grouping semantics should be removed entirely from FileSourceManager and moved entirely to the individual Visualizers. From a design perspective, FileSourceManager should have no notions about how a Visualizer wants its data to be organized. It should simply return the data when requested, and leave the task of organizing the data to the Visualizer. This would provide a cleaner division of labor between the different modules.

Finally, FileSourceManager should be augmented with the ability to delay purging data files and data columns for a short period of time even after all references to the data file has been removed. This feature is useful because users will often close a Visualizer only to open another that uses the same data file as the one he just closed. We do not want
to purge data that will be used again shortly. Currently, this feature is being emulated by
having Visualizers call closeFileSource through a timer a minute later than when the
Visualizer is actually closed. However, this hack is unclean, and the feature should instead
be moved inside the actual implementation of FileSourceManager.

Aside from fixing the existing implementation flaws in VizStudio, the application can
be extended in many useful ways. The first is through the addition of new Visualizers.
A Visualizer that can display aggregate real-time statistical information about the physiology
data would be useful, as would a Visualizer that displays the power spectra of EEG
recordings. Another way to extend VizStudio is to add support for real-time data feeds.
This would obviate the need to use third-party software to first record the physiology and
market data.

These changes and extensions aside, VizStudio is proving to be a useful tool in helping
to understand how emotion and rationality interact in the context of decision making under
uncertainty. Its modular design provides hooks to allow more sophisticated displays to be
added easily, extending the usefulness of the tool well beyond its current implementation.
Bibliography


