AN EXTENDED SPREADSHEET PARADIGM FOR DATA VISUALISATION SYSTEMS, AND ITS IMPLEMENTATION

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,

FACULTY OF SCIENCE

AT THE UNIVERSITY OF CAPE TOWN

IN FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

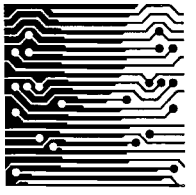
By

Fabian Nuñez

November 2000

Supervised by

Edwin H. Blake



Abstract

We describe a data visualisation system which uses spreadsheets as its user interface metaphor. Similar systems implemented in the past were hampered by the contradiction between an imperative formula language and the declarative spreadsheet framework. We have analysed spreadsheets from a data visualisation point of view, and built a system that is an improvement over past efforts to combine spreadsheets and data visualisation. Our prototype combines the following three techniques: we store lists of values in each spreadsheet cell; we use the functional programming language Scheme as the formula language and we make use of lazy evaluation. The novel combination of these techniques makes our system consistently declarative in nature, and gives it several advantages such as small, uncluttered visual programs, the ability to deal with arbitrarily large datasets and the use of advanced functional language features. We have demonstrated the validity of our work through examples where real-world data is visualised, and through Green's *Cognitive Dimensions Framework*, which shows that our extended spreadsheet metaphor is at least as usable as commonly-used dataflow methods.

Acknowledgements

I would like to thank Prof. Edwin Blake, my supervisor, without whose advice and helpful suggestions ViSSh would not be nearly as useful as it is now. I would also like to thank Dr. Gary Marsden for his help and encouraging advice, Dennis Burford for the Kaap Vaal Seismic Experiment data and Oliver Saal and Jinsong Feng for the use of their ATM traffic data. Finally, I would also like to thank the South African National Research Foundation for their support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

In this chapter we first set out the aims of our research project. We argue that a new paradigm for data visualisation systems is needed, and suggest why spreadsheets are suitable for this role. Unfortunately current spreadsheets are lacking in several ways as user interfaces for data visualisation systems. We have developed extensions to the spreadsheet paradigm to address these shortcomings. We cover the main results of this research in Section 1.7, and finally provide a chapter-by-chapter outline.

1.2 An Alternative User Interface for Data Visualisation

The aim of this research is to develop an alternative user interface paradigm for data visualisation systems, based on the well-known spreadsheet user interaction paradigm. This work describes the design of a software prototype, *ViSSh* (short for **Vi**sualisation **S**pread**Sh**eet), which was used to determine the usefulness of the new paradigm, as well as our experiences with this prototype.

Our contribution lies in the fact that we have taken the traditional spreadsheet paradigm, analysed it and devised an extension of this paradigm that addresses the shortcomings that spreadsheets have with respect to their use as data visualisation systems. This extension, however, retains the basic "feel" of the spreadsheet paradigm so that users familiar with products such as *Microsoft Excel* or *Quattro Pro* can easily adapt to the new paradigm. Although there have been previous efforts in this direction, we believe that ours is a novel improvement because of the consistently declarative nature of our extended spreadsheet paradigm. We have also integrated and rationalised several

ideas that have been suggested in the literature. Furthermore, we provide a theoretical foundation to support our extension of the spreadsheet paradigm; previous work has mostly glossed over the theoretical implications and concentrated on the practical aspects.

This theoretical basis consists of an alternative, functional view of spreadsheets and a demonstration of the equivalence that exists between spreadsheets and dataflow systems. The importance of these is described below.

1.2.1 Why Spreadsheets?

The reason for our using spreadsheets for data visualisation is simply that users of data visualisation systems are not usually computer programmers. This fact has already been recognised by makers of dataflow-based data visualisation systems such as *Iris Explorer* [15]. These systems make use of the dataflow programming paradigm; while this paradigm is not particularly complex in nature, it does have its problems. Dataflow graphs tend to become cluttered if they are large or heavily-edited, as is often the case with exploratory programming.

Spreadsheets are not as susceptible to this problem because the connections between cells are *implicit*, as opposed to the explicit connections that exist between dataflow nodes. This results in a programming environment that is not as demanding as textual programming languages such as Fortran or C, but which has many of the positive qualities of these languages; the greatest of these is the relative ease with which programs can be made easy to read. Large dataflow diagrams, by comparison, tend to get very cluttered after heavy editing. The drawbacks of implicit connections inherent in spreadsheets are dealt with below.

Spreadsheets do have problems when it comes to dealing with large datasets. Datasets consisting of tens of thousands of items would need enormous, mostly-empty spreadsheets just to contain the data (since spreadsheets manipulate blocks of data as rows or columns, a spreadsheet containing a thousand data points would need to be at least a thousand cells wide or tall). Such a huge spreadsheet would be awkward to manipulate, and recalculations would take unacceptably long to complete. Furthermore, spreadsheet formula languages are not generally flexible enough to be useful for general-case data visualisation; most do not have any graphics primitives, for example (one notable exception to this is *Forms/3* [7], which has explicit built-in support for graphics).

These problems can be solved by analysing the spreadsheet paradigm from a functional point of view. This allows us to make use of software techniques (such as lazy evaluation) which increase the flexibility of the spreadsheet and provide the groundwork for demonstrating the equivalence that exists between spreadsheets and dataflow systems; this equivalence is important because it implies

that for any given spreadsheet there always exists an equivalent dataflow diagram, and *vice versa.* This result guarantees that any dataset that can be visualised with an existing dataflow-based system can also be visualised with a spreadsheet-based system.

We also use the equivalence in a more direct way to add functionality to our prototype that overcomes one of the main deficiencies of spreadsheets, namely the fact that "is depended on by" relationships are implicit and not visible (as opposed to "depends on" relationships, which can be easily deduced by inspection of spreadsheet formulas). This functionality lets the user see a dataflow diagram equivalent to the spreadsheet being edited, which makes all inter-cell dependencies explicit. This is superior to simply using a dataflow editor because the algorithmically generated dataflow diagram does not need "cosmetic" maintenance, since the algorithm used to generate it minimises clutter.

1.3 Theoretical Foundations

We have based our work on a consistent theoretical framework. This framework was derived by taking the current spreadsheet paradigm, analysing it and then extending it where the analysis revealed shortcomings with regard to data visualisation.

In Section 2.3 we define what we understand by a spreadsheet. Given the diversity that exists in the marketplace at this moment in time, we have taken the elements common to most currently-available spreadsheet software, and come up with the following definition:

> A *traditional spreadsheet* is a program used to manipulate objects arranged in a tabular fashion. These objects include text, numbers and formulas. Formulas are used to calculate values for cells from those stored in other cells. A formula may only read values from other cells and not write new values into those cells. The result of the formula appears only in the cell the formula resides in.

This definition of the spreadsheet paradigm hints at the functional nature of spreadsheets, since it reveals the fact that spreadsheets can be considered to be *Applicative State Transition* (AST) systems [2] (see Section 2.5).

We started our theoretical analysis by looking at Isakowitz and Schocken's analysis of spreadsheets [23], where spreadsheets are described as consisting of four parts. These are formulas, stored constants, comments and the binding of these three to rows and columns. Isakowitz and Schocken

based their conclusions on the analysis of database systems; however, when we looked at their results from a functional point of view, as suggested by the fact that spreadsheets are AST systems, we found that spreadsheets can also be described as a functional computational system coupled with a grid-based editing layer (see Section 3.4). This led us to the following, alternative definition of a spreadsheet:

> A spreadsheet is a finite set of functions (the *Dataflow Logic Layer*) and a grid-based visualisation of them (the *Adjacency-based Editing Layer*), which taken as a whole solve a given problem.

The term *Dataflow Logic Layer* in the definition above reflects a similarity we found between this definition of spreadsheets and the dataflow programming paradigm: our definition essentially states that spreadsheets are an editing environment for functional programs, while dataflow diagrams (the most common way of representing dataflow systems) can be used as a shorthand notation for mathematical functions (see Section 2.2).

We further explored this similarity between spreadsheets and dataflow systems, and arrived at the conclusion that both systems are equivalent in nature; a proof for this conclusion can be found in Section 3.5.

With these findings, namely the alternative functional definition of spreadsheets and the equivalence of spreadsheets and dataflow systems, we set out to extend the spreadsheet paradigm to make up for its shortcomings with respect to data visualisation.

1.4 Extensions to the Spreadsheet Paradigm

Although spreadsheets have many qualities that make their use desirable for data visualisation [30], they also have shortcomings due to the fact that they were not originally designed to handle the volumes of data commonly dealt with in a data visualisation environment. Using the theoretical foundation we have described above, we have extended the spreadsheet paradigm so that it may be used for data visualisation, while retaining its useful qualities (such as ease of modification, tidiness, etc.)

The extensions we have made to the spreadsheet paradigm are threefold (Chapter 4 describes these extensions at length): Firstly, we store multiple items in a single spreadsheet cell, instead of a single value. We have opted to store lists of items in our sample implementation, but there is no reason why other data structures could not be used. Secondly, we make use of a full-featured

functional programming language instead of the specialised formula languages commonly used in spreadsheet software. Making use of such a language makes the extended spreadsheet paradigm much more flexible, since it both supports the current usage of specialised functions while allowing more functions to be created when needed.

Finally, spreadsheet recalculations are implemented using a lazy evaluation model. Since data visualisation typically involves discarding large volumes of data (in filtering operations or simply because it lies outside the area being observed), lazy evaluation can greatly speed up recalculations. This is because in a lazy evaluation environment, data processing is automatically deferred until the results are needed; should those results never be requested, they would never be calculated. More interestingly, these decisions are made *dynamically at run-time*, relieving the programmer of the burden of optimising the program for any given input dataset. In an environment where data decimation is common, such as data visualisation, this means that functions can be written under the assumption that the entire dataset will be processed, even though an unknown percentage of it will not be. This can allow even non-expert programmers (i.e., the intended users of data visualisation systems) to run their visualisations with a high level of efficiency.

1.5 The ViSSh Prototype

After having devised an extended spreadsheet paradigm capable of coping with the demands imposed by data visualisation (described in detail in Chapter 4), we built a software prototype to test our claims on real data. The prototype, which we named *ViSSh* (**Vi**sualisation **S**pread**Sh**eet), implements the extended spreadsheet paradigm in the following manner (Chapter 5 describes the prototype in detail):

- Linked lists of values are stored in each spreadsheet cell.
- The programming language *Scheme* is used as a formula language, since the fact that both programs and data are lists of values simplified the design.
- Lazy evaluation was implemented as a layer over the *Scheme* interpreter, since that language does not natively support lazy evaluation.

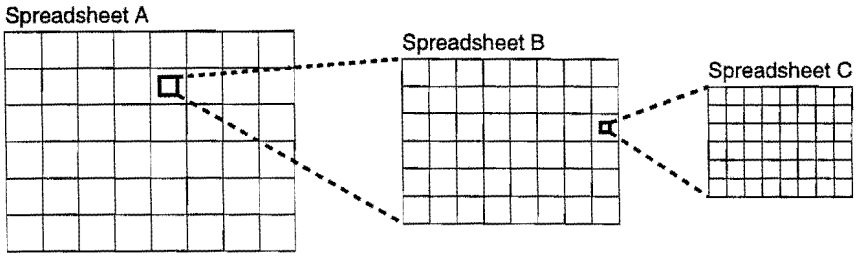


Figure 1: This illustrates the concept of layered spreadsheets. One of the cells in Spreadsheet A evaluates a cell in Spreadsheet B, causing a recalculation of Spreadsheet B and effectively treating the entire spreadsheet as if it were a spreadsheet formula. Spreadsheet B can, in turn, evaluate cells from Spreadsheet C, and so on. This mechanism allows spreadsheets to benefit from encapsulation, since only the result of the evaluated cells are visible by the spreadsheet doing the evaluation. If each of the spreadsheets is running in a different computer, a simple model of distributed spreadsheets ensues.

Additionally, the ViSSh program has several features that were found to be useful during its development:

- Spreadsheets can functionally "call" other spreadsheets, allowing the development of modular function libraries, implemented entirely as spreadsheets. This follows from the functional view of spreadsheets described in Section 1.3 above, and leads to a hierarchical view of layered spreadsheets (see Figure 1), which has several benefits, such as introducing the concept of encapsulation into the spreadsheet paradigm and allowing for a simple model of distributed spreadsheets (by allowing each of the linked spreadsheets to exist in a separate machine).

- Users can summon an equivalent dataflow view of the spreadsheet that illustrates the flow of data between cells in the spreadsheet (see Section 5.2.4). This follows from the equivalence of spreadsheets and dataflow systems described in Section 1.3 above and allows users to view spreadsheet dependencies that are hidden in the tabular view of the spreadsheet ("which cells depend on this one?"). It may also allow transfer of skills from current dataflow visualisation systems.

- Users can summon a view of the spreadsheet in which each spreadsheet cell is represented by a small icon. This was found to greatly simplify the task of spreadsheet navigation, since a much larger area of the spreadsheet can be visually scanned at a time.

- ViSSh implements animation in a declarative fashion by processing list items in a definite order, with a known delay between successive items. Since the declarative paradigm does

not specify any ordering when operating on independent data items and is completely time-independent, this implementation of animation does not affect the correctness of the visualisation.

- Interaction with the rendered 3D environment in the context of spreadsheets is implemented by having a type of spreadsheet cell that has an associated 3D representation, and triggers a recalculation whenever that 3D representation is interacted with. Examples of this type of cell are the *1D Dragger*, which looks like a double-headed arrow and reports how far it's been dragged along its longitudinal axis, and the *Pick Cell* which reports which 3D object from a given list has been selected by the user.

1.6 Cognitive Analysis

In order to demonstrate the usability of the extended spreadsheet paradigm, we visualised some real-world datasets (two of the visualisations are described in detail in Chapter 6) and used the experience gained to make a Cognitive Dimensions Framework [18] analysis of the prototype (Section 2.8.1 briefly describes the technique). We then took the results of this analysis and compared them with published results obtained from existing data visualisation packages [18]. This comparison reveals that, from a usability standpoint, ViSSh is quite suitable for use in data visualisation tasks. Since the prototype was designed to be an implementation of the extended spreadsheet paradigm, the results of this comparative analysis also apply to the extended paradigm itself.

1.7 Summary of Findings

In this dissertation we argue the case for using an extended spreadsheet paradigm as the user interface for data visualisation systems, providing both a theoretical framework (outlined in Chapters 3 and 4) and practical demonstrations of the practicality of our approach. These practical demonstrations consist of a complete data visualisation system, described in Chapter 5, two case studies involving real-world data (in Chapter 6) and finally, a cognitive analysis based around Green's Cognitive Dimensions Framework [18] to demonstrate the usability of the new spreadsheet paradigm; see Chapter 7 for details.

We began the dissertation by noting that although the dataflow paradigm is quite suitable for data visualisation tasks, the use of dataflow diagrams as the user interface has problems with scalability;

namely, the clutter in a dataflow diagram is proportional to both the number of nodes in the diagram and the amount of editing that the diagram is subjected to.

We then followed with an analytical study of the common features of current spreadsheets, based on a survey of commercial and experimental designs, and found that spreadsheets can be described entirely in functional terms (see Section 3.3.3). This alternate view of spreadsheets was used as a basis to describe spreadsheets as visual environments for the editing and visualisation of functional programs (Section 3.4). Based on this, we were able to show that spreadsheets are equivalent to dataflow systems (Section 3.5 describes algorithms to convert a dataflow diagram into a spreadsheet, and to reverse the process).

We have found that spreadsheets in their present form have certain shortcomings as far as their use for data visualisation systems is concerned, mostly due to the fact that current spreadsheets cannot deal with datasets containing more that a few hundred elements. However, we have found that the combined use of compound datatypes (Section 4.3.1), functional programming languages as formula languages (Section 4.3.3) and lazy evaluation (Section 4.3.2) address these shortcomings, while retaining the basic spreadsheet "feel." We have used these findings to define an extended spreadsheet paradigm which consists of the traditional spreadsheet paradigm augmented by the three features described above. This extended paradigm retains all the advantages associated with spreadsheets with regard to exploratory programming, while allowing the efficient processing of very large datasets within the context of data visualisation.

In order to test the extended spreadsheet paradigm, we have built a software prototype, which is described in detail in Chapter 5. This program implements the extended spreadsheet paradigm, and also includes certain features which, although not deemed fundamental enough to include in our extended spreadsheet paradigm, are nevertheless quite useful. These are the ability for spreadsheets to evaluate cells in other spreadsheets, effectively implementing function calls and encapsulation at the spreadsheet level (this follows from the functional nature of spreadsheets discussed above); a window that generates and displays a dataflow diagram equivalent to the spreadsheet currently being edited (this is a very useful debugging tool that follows from our discovery of the equivalence of spreadsheets and dataflow systems); and a "zoomed out" view of the spreadsheet which makes use of small icons representing spreadsheet cells in order to give users a wider view of the spreadsheet — this was found to be another useful debugging aid, supported by Nardi's work that indicates that users prefer to be able to view as much data as possible without scrolling [41].

Finally, we have used Green's Cognitive Dimensions Framework [18] to compare our software prototype to existing data visualisation systems. Based on this comparative study, we have arrived at

the conclusion that our system (and hence the extended spreadsheet paradigm), is at least as usable as existing dataflow systems, with several improvements (see Section 7.4).

1.8 Outline of this Dissertation

Chapter 2 In Chapter 2 we describe the work that we base our findings and extension of the spreadsheet paradigm on. We first examine the dataflow paradigm, which underlies most current data visualisation systems. Then we provide a working definition of a "Spreadsheet," to ensure that our discussion does not become obscured by ancillary features added to recent spreadsheet software as a response to market pressures. This is followed by a description of the spreadsheet paradigm, giving brief descriptions of several commercial and experimental spreadsheet systems which we believe constitute the current state of the art in this field.

We then describe the dataflow programming paradigm and describe its pros and cons with regard to data visualisation, followed by a discussion of spreadsheets and data visualisation, in which we outline previous work in this area. This is followed by a discussion of the similarities that exist between spreadsheets and dataflow systems.

Finally, we introduce Green's Cognitive Dimensions framework [18], which we shall use in Chapter 7 to measure the usability of the extended prototype, and provide a simple example to explain the technique.

Chapter 3 This Chapter examines the underlying nature of spreadsheets. We begin by using Isakowitz and Schocken's segmented view of spreadsheets [23], and expanding on this from a functional point of view to arrive at the conclusion that spreadsheets can be described as consisting of a set of functions plus an editing layer. We then show that spreadsheets and dataflow systems are equivalent in nature.

Chapter 4 In Chapter 4 we introduce the extended spreadsheet paradigm that forms the core of this dissertation. We note what failings traditional spreadsheets have with respect to data visualisation systems, and show how these are overcome by the combined use of *Compound Datatypes* (e.g., lists), *Lazy Evaluation* and the use of a *Functional language* as a formula language.

Chapter 5 We have built a prototype to test our extended spreadsheet paradigm, and described it in detail in Chapter 5. This chapter describes ViSSh both from a user's point of view and from our point of view as its designers. All the program's internals are described in detail, from the choice

of rendering toolkit used to the details behind the dual data pipeline. The chapter also discusses ViSSh's animation capabilities as well as two features which are not part of the extended spreadsheet metaphor, but which are useful when editing spreadsheets: the "Broad Overview" window, which eases grid navigation, and the "Show Dependencies" window, which uses the equivalence of spreadsheets and dataflow systems to render a dataflow diagram which corresponds to the current spreadsheet. This is a useful debugging aid, as well as easing the transition from a dataflow-based data visualisation system to ViSSh.

Chapter 6 During and after its development, ViSSh has been tested with real-world data both to improve the program itself and to demonstrate the viability of the extended spreadsheet metaphor as a user interface for a data visualisation system. Chapter 6 describes two such data visualisations: seismic disturbance analysis and ATM network traffic analysis. These applications are described to demonstrate the different features of ViSSh, and to illustrate the process of building visualisations of data using ViSSh.

Chapter 7 In the previous chapters we have noted the advantages spreadsheets have for data visualisation systems, as well as their shortcomings. We then built up the theoretical framework needed to extend the spreadsheet paradigm to address these shortcomings, and devised an extended spreadsheet paradigm based on this framework. A software prototype, implementing this extended paradigm, was then built and in Chapter 7 we present a Cognitive Dimensions framework [18] analysis of this prototype. This analysis reveals the cognitive properties of the prototype, which when compared to those of existing data visualisation systems [18], indicate the suitability of the extended spreadsheet paradigm as a data visualisation platform.

Chapter 8 Finally, in Chapter 8 we end this dissertation with some concluding remarks and suggestions for future work.

Chapter 2

Background

2.1 Introduction

This chapter first examines the dataflow paradigm, which at the time of writing (2000) seems to be the most popular with data visualisation system designers (e.g., apE [14], IBM's Data Explorer [33], Irix Explorer [15], AVS [61], etc). Then the spreadsheet paradigm is discussed, together with the advantages it has for data visualisation over the dataflow paradigm, as well as its disadvantages in this area. These disadvantages shall be addressed in a later chapter, where some basic extensions to the spreadsheet paradigm will be discussed that will alleviate the problems.

Spreadsheets have been a part of computing almost since the emergence of the personal computer (e.g., Bricklin and Frankston's *VisiCalc*). Although more than twenty years have elapsed since their invention, their basic nature (i.e., a grid populated by text, numbers and formulas in fixed relationships which updates itself when some part of it is altered) has remained unchanged, and as such most computer users have been exposed to the spreadsheet paradigm at some time or another.

In order to more objectively quantify the cognitive differences between the dataflow and (extended) spreadsheet paradigms, we shall be using an analytical tool known as the Cognitive Dimensions Framework [18, 17]. This analysis tool breaks down interactive systems environments along fourteen mostly-orthogonal, cognitive axes. At the end of this chapter we shall briefly introduce the technique, which shall be used in Chapter 7.

2.2 The Dataflow Paradigm

Traditionally, data visualisation systems have used the dataflow paradigm. Its underlying mechanism is semantically quite simple: a program is composed of a set of functional units connected by data pipes. Data can be seen as "flowing" through the pipes into the functional units, each of which implements a given function. The data that enters each of these functional units is treated as the arguments to the function, while the results of these functions "flow" out of the functional units through other pipes. These results may then flow into other functional units, where they form the arguments to other functions, and so on. In this way a dataflow program can be seen as simply a graphical notation for mathematical functions.

Semantically, dataflow systems are very different from Von Neumann machines. The dataflow model has no concept of global memory or a program counter [56, pp. 173–177]; instead, the model deals only with values and execution is governed by *node firing rules*, the most common of which is to trigger evaluation of the current node as soon as all inputs contain valid data. This data moves from node to node in the form of discrete *tokens*. Since dataflow systems have no state and their behaviour is firmly based on mathematical functions, they are declarative in nature.

Dataflow diagrams are a very common editing metaphor for dataflow systems, since they directly represent the logical structure of dataflow systems. A dataflow diagram typically represents each functional unit with a box containing the implemented function, with the data pipes represented as lines connecting the boxes. For example, the dataflow diagram in Figure 2 implements the function $H(G(F(x)))$.

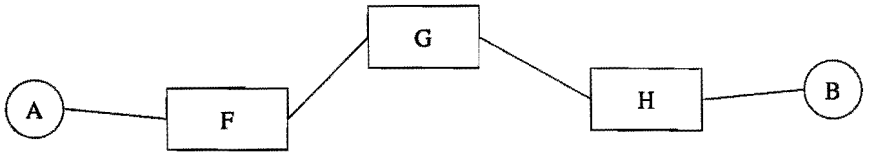


Figure 2: This is an illustration of a dataflow diagram: Several boxes (representing functions) are joined together by lines (representing "pipes" through which data flows). Data enters the system from A, flows to the box labeled "F," which applies some function to the data. The result of this function then flows to the box labeled "G," which applies another function, and so on until the transformed data leaves the system at B.

2.3 A Brief Definition of Spreadsheets

Although the term "spreadsheet" is quite well-known, enough variety exists in the field to merit a brief definition, for the purposes of this document:

> A *traditional spreadsheet* is a program used to manipulate objects arranged in a tabular fashion. These objects include text, numbers and formulas. Formulas are used to calculate values for cells from those stored in other cells. A formula may only read values from other cells and not write new values into those cells. The result of the formula appears only in the cell the formula resides in.

This defines the "lowest common denominator" spreadsheet (such as the original *VisiCalc*). Our definition excludes hybrid systems such as those with embedded BASIC interpreters, that allow cells to directly modify other cells. This leads to a lack of referential transparency, and as will be shown, is not necessary for the functioning of spreadsheets.

2.4 The Spreadsheet Paradigm

In 1978, Daniel Bricklin and Robert Frankston released VisiCalc (see Figure 3). This was a flexible analytical tool that gave non-programmers the ability to perform complex data analysis on non-trivial volumes of data. The real impact of this program was that it allowed users to develop their own data models, as opposed to relying on someone else's programming skills [52]. It also gave users the ability to perform speculative analysis of data quickly and easily. This could be useful, for example, in forecasting the repercussions of a particular financial decision such as raising the profit margin.

VisiCalc based its user interface metaphor on the journals and ledgers used by accountants, and became a popular application with the financial community. The user interface itself is quite simple: the user directly manipulates a workspace, which is divided into a regular rectangular grid of cells. Into each grid cell the user writes either a number, a formula that will derive a result from other numbers already on the spreadsheet, or a short descriptive phrase.

The tabular grid used to display and edit the spreadsheet is at the heart of the spreadsheet's usability. Tables have been in use since the days of Ptolemy [40], making them one of the earliest computational tools in existence. The regular grid structure, coupled with a simple yet effective way of locating items within it (by a row/column reference) makes tables, even very large ones, easy to navigate (once an item has been located, it is easy to find it again — this fact has been used by cartographers for centuries). Likewise, spreadsheet cells are arranged in rows and columns, and addressed using a letter/number system, where the letters indicate the column number and the number the row (e.g., cell B3 is the cell third from the left edge and second from the top edge of the spreadsheet).

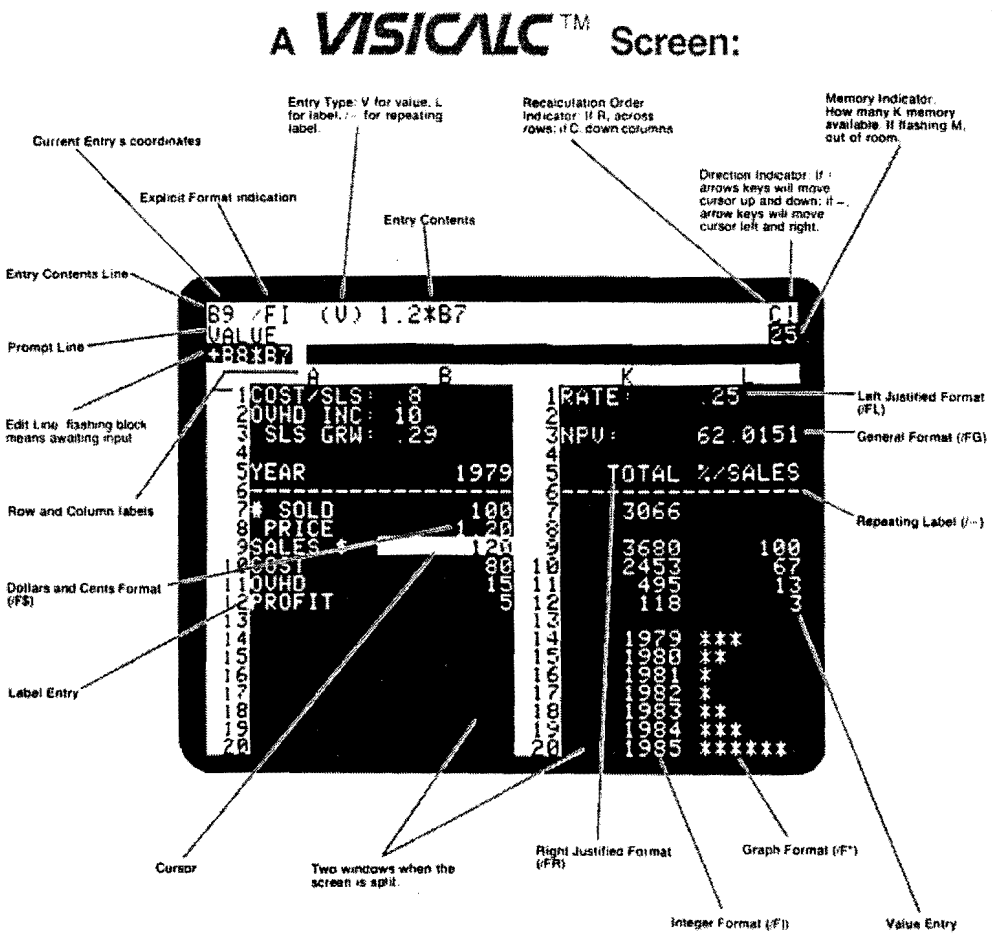


Figure 3: This is what a typical VisiCalc screen looks like (this image was scanned from the reference card). VisiCalc was the very first spreadsheet program, launched in 1979 for the Apple II computer.

Although the tabular representation of spreadsheets is inherently easier to navigate than other methods, the reality of a limited display can make navigation a difficult exercise. The problem is that although the grid-like nature of spreadsheets makes a given location easy to return to, actually *finding* the information in the first place may be a trying exercise. One way to alleviate this problem is by the use of a lenticular interface, as suggested by Rao and Card's *Table Lens* [50]. This system always displays the entire spreadsheet, but compensates for the limited size of the screen by changing the size of the spreadsheet cells. The cell that is being edited at any given time is shown at full size, while the size of all other cells diminishes with increasing distance from the current cell. Figure 4 illustrates this user interface technique. The *Focus* system [57] improves on this technique by

also assigning attributes to groups of cells and restricting the display of cells to those with attributes that match the ones specified by the user.

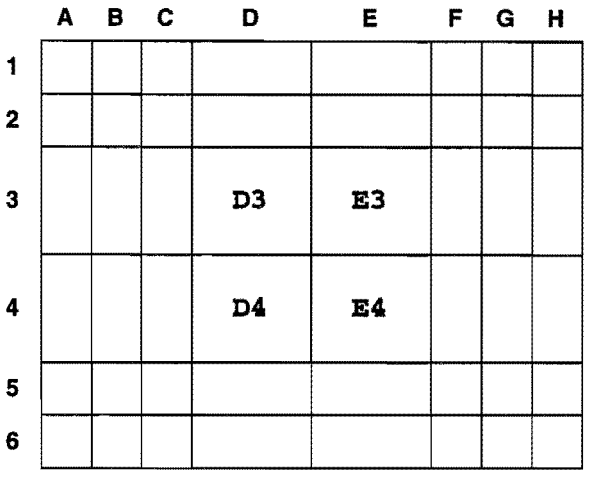


Figure 4: This is a lenticular spreadsheet interface for a spreadsheet. The four spreadsheet cells comprising the area of interest (in this case, cells D3, E3, D4 and E4) are shown magnified, while all other spreadsheet cells are minimised.

Although the spreadsheet paradigm is quite simple, its pencil-and-paper implementation (e.g., a ledger) can be quite clumsy, since changing one number can involve large amounts of erasing and tedious recalculation. On the other hand, this paradigm is ideally suited for a computer, since large volumes of calculations can be quickly performed when the need arises.

Spreadsheets have been shown to be highly successful tools for interacting with numerical data, such as applying algebraic operations, manipulating rows or columns and exploring "what-if" scenarios [9]. Their usability is not limited to single users, either; for example, Nardi and Miller [42] have studied the possibilities of using spreadsheets in a collaborative environment. Although spreadsheets seem to have evolved considerably since the early days of VisiCalc, in fact they have not changed substantially since then. The basic grid structure remains the same, with most improvements centering around "syntactic sugar" for the macro language used for the composing of formulas, and useful additions such as integrated charting packages. There are some interesting deviations from this formula, for example Eriksson's *Scheme in a Grid* [16], which uses the functional programming language *Scheme* as a formula language; Piersol's *Analytic Spreadsheet Package* [49], which uses the object-oriented language *Smalltalk 80* for this same purpose, and Kriwaczek's *LogiCalc* [29], which combines spreadsheets with the logic programming language *Prolog*. These variants do not stray far from the basic paradigm, however: the basic grid structure remains in place

and Eriksson, Piersol and Kriwaczec have ensured that the basic spreadsheet "feel" remains intact in their respective works.

The visual language *Forms/3* [7, 19] retains the concept of cells and formulas, but it does away entirely with the grid structure normally associated with spreadsheets; instead, cells are labelled by the user when they are created and these names are used by formulas instead of cell references. This makes Forms/3 an interesting hybrid between "normal" programming languages (since the cell labels resemble variable names) and spreadsheets (since automatic recalculation still holds). The commercial program AgentSheets [1] works along similar lines as this, but implements recalculation using pervasive multi-threading. This has the effect of making all calculations that are not interdependent run in parallel, thereby making the system highly suitable for simulation. A more radical approach is taken by Lewis's *NoPumpG* [31] and *NoPumpII* [65] prototypes, in which all spreadsheet cells are free-floating and graphical interaction is used to modify cell values. Casady & Greene's *Spreadsheet 2000* [8] is conceptually similar to Lewis's work, but does away with textual formulas altogether. Instead, it uses a dataflow method in which small spreadsheets (which in traditional spreadsheets would be implemented as cell ranges) are linked together via operator nodes (Figure 5 illustrates this). Myers' *C32* constraint-based system [39] is also similar to this, although his use of spreadsheets is limited to the two-column variety currently known as *property sheets*.

The usefulness of the spreadsheet paradigm is not limited to spreadsheets, however: Johnson *et al* have extended the basic spreadsheet paradigm to use it as the basis for the *ACE* programming environment [25]. This provides users with a system that is spreadsheet-based but which is so far beyond products such as *Microsoft Excel* so as to be more accurately described as a general-purpose programming environment.

Compared to more traditional programming paradigms, spreadsheets have several important advantages. Nardi and Miller [40] remark that the biggest advantage of spreadsheets is not cognitive but motivational: after only a few hours of work, spreadsheet users are rewarded by simple but functioning programs. Levoy [30] found that for data visualisation tasks, compared to the commonly used dataflow model [14, 15, 33], spreadsheets are "more expressive, more scalable, and easier to program." These advantages arise mainly from two properties of spreadsheets and their formula languages [40]:

- **High level, task-specific functions**
  The formula languages normally found in spreadsheets have most basic arithmetic operators (addition, subtraction, multiplication, division, exponentiation, etc.), as well as many simple, yet specialized functions such as time and date arithmetic. Since these high-level operations are already present as primitives, they do not need to be built from other building blocks.

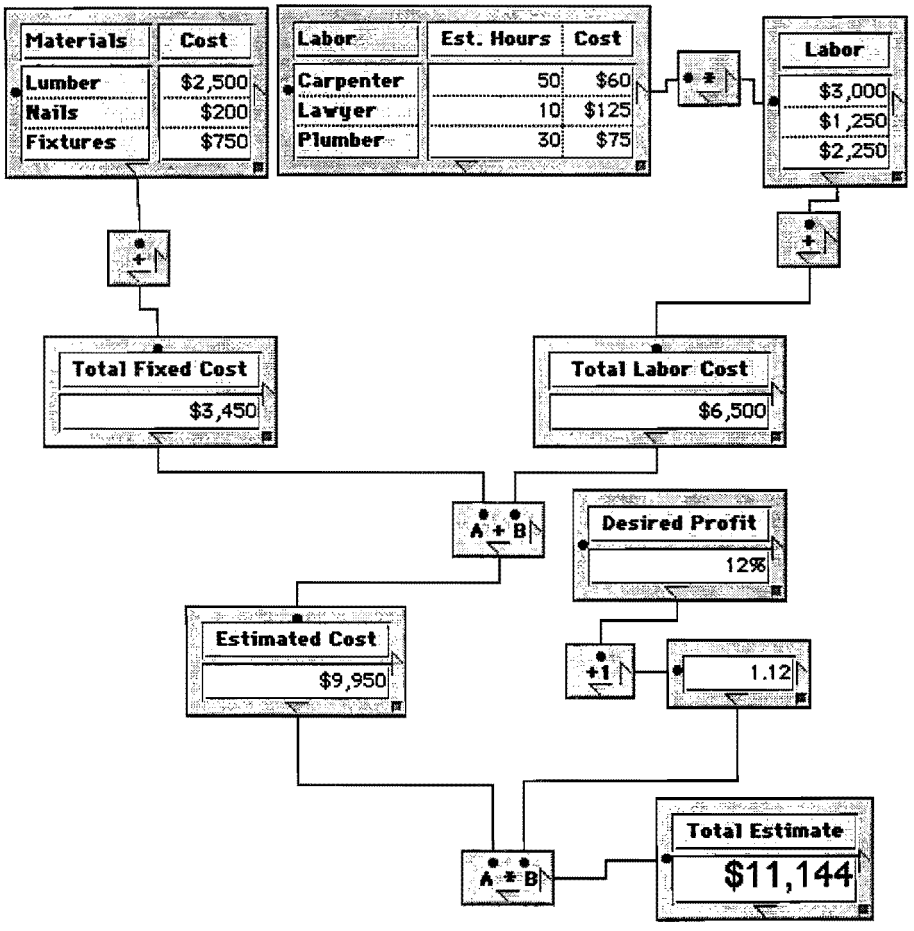


Figure 5: This is a screenshot of SpreadSheet 2000. This program combines the spreadsheet and dataflow paradigms to provide a system that is easier to navigate than traditional spreadsheets.

- **Very basic control-flow constructs**

  Flow control is a difficult programming concept [40]. Therefore it makes sense that a programming paradigm which somehow either eliminates or hides flow control will be easier than one which exposes flow control. Spreadsheet formulas partially hide flow control because spreadsheet cells do not explicitly need to "call" each other — they simply use the values stored in the other cells. Although this implicitly means that cells will have to be recalculated when the cells they depend on have been modified either by recalculation or user interaction, this is handled automatically and so this fact is hidden from the user. Iteration and recursion, both common control-flow constructs, are obviated by the fact that spreadsheet formulas can operate on cell ranges, applying an operation to many values.

As an illustration of these advantages, consider the simple operation of calculating the average of 100 numbers. This is illustrated in Figure 6 — the C [27] code necessary to implement the AVG function is not as readable as the spreadsheet formula. All the necessary punctuation, data typing, etc. helps to obscure the workings of the function, and eases the introduction of bugs. In all fairness, the functionality could have been encapsulated in a C function (the C function call `avg(A,1,100)` is as readable as the spreadsheet formula `=AVG(A1:A100)`), but the code to implement the function (i.e., the code in Figure 6) would still be required. By contrast, in the spreadsheet version of the code, the looping construct (the `for` loop) is eliminated, together with the temporary variables (variables `i` and `total`) and the fact that these variables have different data types. Additionally, the contents of cells A1 to A100 are guaranteed to be current at all times. A C programmer would need to enforce this by the addition of further code.

```
=AVG(A1:A100)
```

Averaging with a spreadsheet language

```
int i;
double total = 0.0;

for (i = 1;i <= 100;i++)
  total += A[i];

return total / 100.0;
```

Averaging with the C programming language

Figure 6: This contrasts the procedure to be followed when the average of 100 numbers must be calculated, using a typical spreadsheet formula or the C programming language [27]. The spreadsheet version is more readable and concise than the C version, which is typical of the imperative paradigm — contrast the explicit looping construct in the C code with the implicit spreadsheet expression. Imperative programs, being larger, are also more error-prone: for example, when altering the example above to calculate the average of a different range, the C programmer must remember to update the "100" in both the `for` loop and the `return` statement.

2.5 Spreadsheets and the Declarative Paradigm

In his seminal 1977 Turing Award Lecture [2], John Backus outlined what he believed were the main problems with the imperative programming paradigm. He then described a framework which he believed can solve these problems, which is now widely known as the declarative programming paradigm. To illustrate, he outlined several programming systems which implement this paradigm. One of the systems he discussed was the Applicative State Transition (or AST) system. This is

characterised by the fact that AST programs are mostly functional in nature, but unlike functional languages such as pure Lisp, they keep track of state. However, in contrast with the imperative programming paradigm, rules must be observed when updating this state.

The rules of AST systems are quite simple, however, and can be summarised into the following: *a program will accept an initial state and input at startup, but will not accept further input until an output has been issued and the new state, if any, installed.* In other words, AST systems combine the purely functional and imperative programming paradigms by being "piecewise functional."

Some examination of the spreadsheet paradigm will reveal that most spreadsheets fall under this category, if one considers a spreadsheet to be a program, and the formulas in that spreadsheet to be the statements that comprise that program. When a spreadsheet recalculation is triggered, each cell is completely evaluated and its result stored before any other cells can be evaluated. The actual process of evaluation of spreadsheet cells does not result in any state changes, since spreadsheet formulas can only read the values stored in other cells, and not write to them. In the context of spreadsheets, "input" means reading the contents of other cells, while "output" is the updating of the value stored in the cell the formula resides in. The only way to change a cell's state is by this "output" operation, i.e. a cell may change only its own state. Therefore, spreadsheet recalculations obey the rules stipulated by the definition of AST systems, and can therefore be considered to be declarative in nature.

2.6 Data Visualisation and the Declarative Paradigm

The declarative programming paradigm, especially under the guise of dataflow diagrams (see Section 2.2), can be quite intuitive and readily grasped by non-programmers. This has been taken advantage of by the makers of recent data visualisation systems which, with few exceptions, have used the declarative paradigm as a means of expressing the system that is being visualised. Systems such as apE [14], SGI's Iris Explorer [15], IBM's Data Explorer [33] and Advanced Visual Systems' AVS [61] have used the dataflow paradigm to allow their users to do extensive visualisation of large volumes of data without needing to learn the skills needed when using traditional programming languages. The premise behind the dataflow model is quite simple: the user interactively creates the application as a directed graph of modules. Each of these modules consists of a "black box" with several inputs and one output, which performs a relatively complex and specialised computation on its inputs and places the result in its output. The modules are linked to each other (output of A to input of B) by means of a directed graph typically called a visualisation network.

The end result of this process is a set of "black boxes" connected to each other by "data pipes." Data flows through the pipes and is altered by whatever is inside the boxes. A system, such as this one, in which the underlying paradigms carry over into its user interface is said to be *reflexive* [59, pages 80–81]. Reflexive systems, from a user interface point of view, have the substantial advantage of presenting the user with a view of the process that mimics the process itself. This translates into a system that is more efficient, since there is no user interface layer that shields the user from the underlying mechanism.

2.6.1 Limitations of the Dataflow Model

As was discussed above, most modern data visualisation systems use the dataflow programming paradigm. Although the declarative nature of this paradigm makes complex visualisations possible to the layperson, the choice of dataflow diagrams is not ideal. Dataflow diagrams are both easy to create and to maintain, yet they have certain drawbacks. The largest of these is scalability [30]. As a visualisation network becomes more and more complex, the number of modules and their associated interconnections steadily becomes more and more unmanageable; this is due to the fact that connections between dataflow nodes (i.e. the arcs between nodes) are explicit, and after a few dozen nodes have been placed, the screen the arcs connecting these nodes tend to become a tangled mass as editing progresses.

2.7 Spreadsheets and Data Visualisation

After the launch of VisiCalc, spreadsheets gained in popularity with the financial community, as evidenced by several similar programs which were written in the following years, such as *Lotus 1,2,3* [32], *Quattro* [4] and *Microsoft Excel* [35]. However, their use remained constrained to financial calculations for most of the following two decades. The commercial success of these spreadsheet programs eventually led researchers to experiment with the idea of using spreadsheets (or at least the spreadsheet paradigm) in their respective fields, such as proving combinatorial identities [43, 44], user interface development [38], teaching computer graphics [34] and the object-oriented programming paradigm [46]. Others have used the tabular grid as a visualisation aid, without making full use of the spreadsheet paradigm; for example Van Wijk and Van Liere's *HyperSlice* [62] makes use of the tabular grid to simplify the visualisation of multi-dimensional data. It does not qualify as a spreadsheet, however, because each cell cannot be given an independent formula; instead, a single multidimensional dataset is visualised by a set of cells, each of which

presents a different "slice" through the data.

One of the first "true" data visualisation spreadsheets is described in Marc Levoy's 1994 paper "Spreadsheets for Images" [30]. This paper describes a system in which spreadsheet cells are populated not only by numbers and formulas (written in the imperative scripting language *Tcl* of John Ousterhout [45]), but also by graphical objects (such as images or movies) and user interface objects (such as buttons or sliders). Levoy used the intrinsic extensibility of Tcl to add several functions that would allow the user to perform various types of transformations on the images. Levoy found that, compared to the commonly used dataflow model, spreadsheets are "more expressive, more scalable, and easier to program" [30].

More recent work by Chi, Riedl, Barry and Konstan [9, 10] extends the work of Levoy by not constraining the system to images, but instead allowing the user to visualise any type of information. Although they also use the programming language Tcl, Chi *et al* make use of the data visualisation toolkit *VTK* [54] as the underlying data visualisation engine (Levoy used the low-level graphics toolkit *OpenGL* [53], writing the high-level visualisation routines himself). By using these toolkits, Chi *et al* gave their *Spreadsheet for Visualisation* primitives for animation, dynamic visual filtering as well as several well-known data visualisation algorithms such as *marching cubes.*

Hasler *et al* have also built a data visualisation spreadsheet (called the *Interactive Image SpreadSheet*, or IISS) [47] which operates in a similar way to Levoy's work, but uses a different spreadsheet language. It can arguably be deemed to be a better system due to the fact that it functions as a layer above Khoral Research's well-established data visualisation tool *Khoros.*

Spreadsheets can be considered to be adequate tools for data visualisation because they allow data to be easily input, dynamically updated and readily viewed [34]. Additionally, they provide a flexible and easy-to-learn environment for user programming [10], and give the user simple but functioning programs after only a few hours of work [40]. Developers can create spreadsheet templates that users can modify with moderate ease to suit their own needs. Furthermore, developers can do this without needing to worry about details such as memory management or input/output, as these issues are dealt with by the spreadsheet software itself.

Spreadsheets have certain properties that elegantly address some of the problems that are met when visualising multidimensional data, such as the mutable nature of visualisations and the need to have different yet simultaneous views of the same data [9]. The mutable nature of visualisation is addressed by the ease with which a user can make changes to the spreadsheet [30], while the multiple representation issue is dealt with by the emphasis which the spreadsheet places upon operands, as opposed to operators [9] (this means that operands are always visible, whereas operators, i.e.

formulas, must be explicitly requested). Additionally, the fact that spreadsheets automatically keep track of data dependencies between cells and update relevant cells when necessary frees the modeller from one of the many tasks a traditional visualisation programmer is normally faced with. Also, work by Chi, Riedl, Barry and Konstan [10] indicates that for exploratory programming tasks such as data visualisation, temporary values may be as relevant as the end result itself. Since spreadsheet users have access to these intermediate results (whereas these results are hidden by the dataflow diagram), spreadsheets can be considered to be more suitable for data visualisation. However, spreadsheets in their current form are not an ideal tool for data visualisation; they are optimised for financial calculations, and so have two areas that need improvement: they can deal with datasets containing at most a few hundred numbers (several orders of magnitude too small for practical data visualisation), and their formula languages are too rigid for general visualisation use. In Chapter 4 we shall examine these shortcomings in detail, and suggest ways to extend the spreadsheet metaphor to cope with them.

2.7.1 Spreadsheets and the Functional Paradigm

Although current spreadsheets, due mostly to market pressures, have grown far beyond the original scope of early spreadsheets, the essential nature of these programs remains close to that of older programs such as VisiCalc. As was shown in Section 2.5, such basic spreadsheets are declarative in nature.

Although spreadsheets based on imperative languages do exist (in fact some commercial spreadsheets, such as *Microsoft Excel* [35] can be extended by routines written in a dialect of the BASIC programming language [36]), the functional programming paradigm has some definite advantages, especially with regard to the programming style normally associated with spreadsheets. Functional languages can provide much greater modularity than imperative languages due to their use of high-order functions and lazy evaluation [21]. This is particularly useful in a spreadsheet environment because spreadsheets consist of large numbers of disjoint cells. In this environment, enhancements in modularity translate into an increased ability to compose complex functions that are not intractably complex to debug.

Although the ability to debug spreadsheet models may not seem overly important, several studies cited by Panko [48] indicates that errors in spreadsheet models are prevalent. Ronen believes that that this could be because, for many users, the spreadsheet program represents the user's first hands-on experience with a computing device of any kind. This also includes the first hands-on experience with many disciplines which are, for most other computer users, normally acquired

through training. These include such critical disciplines as programming and documentation [52]. Work by Isakowitz, Schocken and Lucas [23] indicates that 25% of a sample of spreadsheets collected from ten large organizations contained errors. Isakowitz, Schocken and Lucas also mention several studies that conclude that spreadsheet errors are not only prevalent, but also elusive.

Proponents of the functional programming paradigm have shown that functional programs are exempt from several classes of bugs that appear in imperative programs [21]. They have also shown that functional programs also have the potential for increased modularity due to the availability of features such as high-order functions, which take other functions as arguments [21]. This increased modularity would be desirable in a spreadsheet, since the spreadsheet's grid architecture and reliance on primitive formulas (such as `add` or `average`) promotes modular programming. Given this, it seems logical that spreadsheets with a functional basis would, sooner or later, merit investigation.

Examples of this functional approach to spreadsheet design are *Scheme in a Grid* [16], which combines traditional spreadsheets with the functional language Scheme, and De Hoon's MSc thesis [11], which discusses the implementation of a spreadsheet using the functional language *clean* [5].

2.7.2 Spreadsheets and Dataflow

Systems such as *Scheme in a Grid* [16] and *FunSheet* [11] demonstrate a link between spreadsheets and the declarative programming paradigm. Some systems exploit this link by providing a dataflow interpretation of spreadsheets, providing spreadsheet users with useful debugging information. According to Igarashi [22], the state of the art of this feature in current commercial systems, can be found in *Microsoft Excel 97's* spreadsheet auditing tools. These can be used by users to query the flow of data into or out of individual spreadsheet cells. This system is very similar to Shiozawa's *Nattospread* [55], which extends this capability to all cells that, directly or indirectly, contribute to the result displayed in the selected cell. Shiozawa's work is innovative in that it makes use of 3D to attempt to reduce the clutter of the generated dataflow diagram (this clutter arises from the fact that all nodes in the diagram retain the position of the original spreadsheet cells).

Igarashi [22] has described a system of three separate dataflow views of a spreadsheet, also overlaying the dataflow graph over the spreadsheet. Firstly, there is a *Transient Local View*; this allows the user to see the dependencies for the cell that the mouse cursor is currently over, in a similar way that *tooltips* are displayed when the mouse hovers over a user interface control. Then, there is the *Static Global View*. This overlays all dataflow information for the visible cells on the cells themselves. If the spreadsheet is not carefully built, this can lead to clutter, which obscures

both the spreadsheet and the dataflow graph. In order to remedy this problem, there is also an *Animated Global View*. This presents the same information as the Static Global View, but makes use of animated colouring of spreadsheet cells instead of drawing arrows to demonstrate the flow of data. Since arrows are no longer drawn, the amount of clutter is significantly reduced.

2.8 The Ergonomics of Spreadsheets

Several researchers have worked on the cognitive aspects of programming in general and spreadsheet programming in particular (e.g., Nardi [40]). One of the fruits of this research has been analysis tools for the cognitive implications of different features of programming systems.

In particular, Green [18] has devised a useful cognitive analysis tool for information-based artifacts. Since both spreadsheets and dataflow diagrams belong to this class of programming environments, Green's work can be used to fairly compare them from a cognitive point of view.

2.8.1 The Cognitive Dimensions Framework

Green proposes using a "broad brush" analysis method, as opposed to the fine-grained analysis of simple tasks traditionally used by HCI researchers. The analysis tool, called a "cognitive dimensions framework," is task-specific, concentrating on process rather than content. It provides us with a set of mutually-orthogonal cognitive dimensions, which in essence allow the broad description of any visual programming environment by conceptually plotting points in a 13-dimensional set of axes. Although the dimensions are meant to be orthogonal, Green admits that there is a certain amount of interaction between them, such that changing one aspect of a visual programming environment to improve its position along a given axis is likely to affect the environment's position along several other axes. Using the Cognitive Dimensions Framework it is possible to quantify the differences between existing programming environments, so they can be objectively compared and conclusions drawn about the suitability of these environments for any given task. This is one of the main reasons we decided to use this technique: it separates the interaction of the system being investigated into several aspects, and rates these aspects. This is far more useful in this case than a single usability measure, since it lets us concentrate on the "important" aspects of the system, given its role as an exploratory programming environment (such as how easy it is make changes to an existing program), while overlooking less important aspects, again within the context of exploratory programming (such as the ability to add a comment to every single line of code). These properties make the Cognitive Dimensions Framework useful in our work, and so we shall be making use of

the technique in Chapter 7.

The cognitive dimensions as defined by Green are the following:

- **Abstraction Gradient.** This deals with the minimum and maximum level of abstraction describable within the environment (or, in this case, language), and includes such things as procedure and data encapsulation. If an *abstraction* is defined as a grouping of elements to be treated as one entity, then systems can be classified as *abstraction-hating*, meaning that there is in general no way to implement abstractions (e.g., spreadsheets, which allow no abstractions), *abstraction-tolerant*, meaning that although abstractions are supported, they are not required (e.g., the C programming language, which has support for functions and abstract datatypes, but does not require their use), and *abstraction-hungry*, meaning that abstractions are required (e.g., the Java programming language, which requires everything to be implemented as a class).

- **Closeness of Mapping.** This deals with how much the syntax of the language separates the problem being solved from the program being used to solve it. This extends from *low*, meaning that there is a lot of syntactic and semantic clutter (e.g., the C programming language, with its semicolons, datatypes and obscure operator precedence) to *high*, meaning that users can state their programs directly, without worrying about syntax (e.g., most visual programming languages).

- **Consistency.** This asks the question: "Once a user has learned part of the language, how much of the rest can be deduced?" This is also known as the "principle of least astonishment." This ranks from *low* (e.g., the commands used to control the *vi* text editor) to *high* (e.g. the *LaTeX* document layout system, with its highly consistent command set).

- **Diffuseness** deals with how many symbols are needed, on the average, to represent any given meaning. This ranks from *high* (e.g., the Cobol programming language expression "`MULTIPLY Num1 BY Num2 GIVING Result.`") to *low* (e.g., a spreadsheet formula such as "`=SUM B1:B52`").

- **Error-proneness** asks if the design of the notation induces the making of careless mistakes. This also ranks from *high* to *low*.

- **Hard Mental Operations.** Are there places where the user needs to resort to external aids (such as pencil and paper) to keep track of what is happening? This ranks from *high* (e.g.,

keeping track of operands in the stack of stack-based languages such as *FORTH* or *PostScript*) to *low*.

- **Hidden Dependencies** exist if two or more entities influence each other "behind the programmer's back." This is also known, especially in the functional programming literature, as *side effects*. This ranges from *few* (e.g., the Scheme programming language) to *many* (e.g., an assembly-language program).

- **Premature Commitment** forces programmers to make a decision before they have all the necessary information. This ranges from *low* (e.g., word-processing software) to *high* (e.g., circuit design software in which wires can be drawn only between components already in place).

- **Progressive Evaluation** allows partially finished programs to be executed so that the programmer may obtain feedback on their progress. This can be either *good* (e.g., a spreadsheet) or *bad* (e.g. a C program, which must have all its brackets matched, semicolons, etc.).

- **Role-expressiveness.** This asks if the user can see how each component of the program relates to the whole. This can be *high* (e.g., in a well-designed and commented C program), or *low* (e.g., in a spreadsheet, which has no support for any way of expressing inter-component relationships).

- **Secondary Notation** consists of a set of primitives the programmer can use to convey more meaning than is normally allowed by the semantics of the software environment. This can range from *high* (e.g., indenting of control constructs in C) to *low* (e.g., WYSIWYG word-processing software, by its nature, does not allow this sort of information coding).

- **Viscosity** deals with how difficult it is to make a single change to a program. This ranges from *low* (e.g., a circuit design program that treats wires as rubber bands, so that moving a component does not break connections between components) to *high* (e.g., a circuit design program that does *not* treat wires as rubber bands, requiring each moving of a component to be followed by a rewiring step).

- **Visibility and Juxtaposability** deals with the ease, or difficulty, of seeing how any two parts of a program relate to each other. This also ranges from *high* (e.g., the *Microsoft Visual BASIC* GUI editing environment, in which double-clicking on a control takes the user to the

event-handler subroutine for that control) to *low* (e.g., early versions of the MS-DOS editor *edit*, which did not allow separate views of the same document).

The cognitive dimensions framework has been designed to be easily mastered [18]; Modugno *et al* [37] have verified that little experience is needed to correctly apply the technique. As a result, the technique has now been in use for several years by members of the HCI community, and as such has been used and commented on by several others (e.g., Blackwell [3], Modugno *et al* [37], Yazdani [67]) and has even been suggested as a teaching aid [66]. A useful tutorial illustrating how this analysis technique can be applied can be found in [17].

In order to provide a feel for cognitive dimensions analysis, a very simple example is outlined below. The information device being analysed is well-known by most researchers: a pencil and paper pad.

2.8.2 Cognitive Dimensions Analysis of a Pencil and Pad

The information device being analysed here is possibly one of the oldest — paper was invented several thousand years ago and the modern pencil has existed since Napoleonic times. The pencil in question is enhanced by the addition of a small rubber eraser on the non-writing end, while the pad consists of a ring binder filled with blank pieces of paper. For the purposes of this discussion, the pencil and the pad are considered to be a single information device, the pencil-and-pad.

Abstraction Gradient

Being real, physical devices, the pencil and paper pad combination cannot be said to incorporate any abstractions. Therefore the pencil-and-pad scores very low on this axis.

Closeness of Mapping

Closeness of mapping deals with how much the usage of the information device separates the problem being solved (in this case, drawing and writing) from the device itself. The pencil-and-pad is a direct-manipulation device, and so the act of drawing is very closely mapped to the drawing itself (e.g., to draw a circle, one moves one's hand in a circular motion).

Therefore, the pencil-and-pad can be said to have a very high closeness of mapping.

Consistency

This is also known as the "principle of least astonishment," which states that once a part of the system has been learned, the user should be able to infer the remaining parts of it.

It should be quite obvious that once the basic principles behind making marks on paper with the pencil have been grasped, all further use will be quite intuitive. Therefore the pencil-and-pad can be considered to be highly consistent.

Diffuseness

This cognitive dimension deals with how "verbose" a given information device is i.e., whether many primitives are required on the average to express any given basic concept.

Again, the direct manipulation nature of the pencil-and-pad makes it a highly concise information device — no extraneous gestures (besides the need to occasionally turn the page) are needed to perform the primitives of drawing and erasing.

Hence the pencil-and-pad has a low degree of diffuseness.

Error-proneness

This cognitive dimension asks, "How easy is it to make mistakes with this system?"

Green differentiates between "mistakes" and "slips" [18], where a "slip" is doing something one "didn't mean" to do (i.e., one knows what the correct action is, but still makes an error), and a "mistake" indicates an error caused by an underlying complexity in the system. The pencil-and-pad can be prone to slips, since the crude user interface does not allow for any form of constraints checking; however, this same lack of complexity prevents the user from making "mistakes."

However, since the design of the pencil-and-pad does not aid the process of making errors, this device can be said not to be error-prone.

Hard Mental Operations

The very simplicity of the pencil-and-pad means that there are very few mental operations involved in its use; in fact, most of the time the user will be using motor skills. Although it could be argued that most users will find drawing a convincing portrait to be "hard," this difficulty is inherent in the problem, not the information device itself.

Therefore, the pencil-and-pad has no hard mental operations.

Hidden Dependencies

A hidden dependency is a relationship between two parts of a system such that the one depends on the other, but this dependency is not fully visible.

The only interactions that can occur between the pencil and the pad can only happen when a stroke is either drawn or erased, both interactions being fully visible to the user. Therefore the pencil-and-pad can be said to have no hidden dependencies. Note that this could not be said if carbon paper were inserted into the pad, since drawing on one page could affect other pages without the knowledge of the user.

Premature Commitment

This occurs when the user is forced to make a decision before all necessary information is available.

The pencil-and-pad can be considered to have a certain amount of premature commitment for the simple reason that pages have boundaries — the user must decide, before starting, in which directions a drawing will grow, and how large it is likely to become. Guessing wrong will mean that the user must erase parts of the markings on the paper. Although this is not a problem for writing (where at most one partial word may have to be erased), for drawings significant parts of the drawing may have to be erased, or worse the entire drawing may have to be scrapped.

Therefore the pencil-and-pad can be said to have a medium degree of premature commitment for drawings, and a low degree for writing.

Progressive Evaluation

If a system supports progressive evaluation, then any partially-completed work can be tested at any time.

The primitive nature of the pencil-and-pad imposes no constraints on evaluation of partially-completed drawings, therefore progressive evaluation is fully supported by the pencil-and-pad.

Role-expressiveness

This cognitive dimension is used to find out how difficult it is to answer the question "what does this part of the drawing do?"

The unstructured nature of the pencil-and-pad makes it simple for users to cluster related markings together, for example written words may be grouped into sentences.

The pencil-and-pad therefore has a large amount of support for role-expressiveness.

Secondary Notation

This is extra information carried by other means than the official syntax. An example from text-based programming languages could be the indenting of loop bodies relative to loop-delimiting statements.

Again, the unstructured nature of the pencil-and-pad allows for secondary notation to be easily added to any set of pencil markings, e.g., margin notes next to a written paragraph.

Viscosity

For the pencil-and-pad, viscosity is defined as *the cost of making small changes to an existing drawing*.

Since all markings on paper exist independently of all other markings, there is no cost associated with adding new markings to an existing drawing. Erasing could be more problematic: a slip of the wrist can cause markings to be accidentally erased.

Overall, however, we consider the pencil-and-pad to have a low viscosity.

Visibility and Juxtaposability

Visibility is defined as *the ability to view components easily*, while juxtaposability is defined as *the ability to place components side by side*. These two are related in that, combined, they imply the ease, or difficulty, of seeing how any two parts of a drawing relate to each other.

Complete visibility is ensured by the simple fact that an entire page can be viewed at a time.

Since the pad is in a ring binder, pages can be pulled out and later replaced. This means that any number of pages can be placed side-by-side and compared (this is limited only by the size of the desktop), and afterwards replaced.

Therefore, the pencil-and-pad can be said to score highly both in visibility and juxtaposability.

Summary of Results

Above, we have seen that the greatest strengths of the pencil-and-pad lie in the simplicity of the user interface. The low abstraction gradient and high closeness of mapping and consistency makes this system easy to learn for beginners (indeed, any child can quickly learn its use), while the low diffuseness, high role-expressiveness and support for secondary notation make it popular with power users.

Additionally, the low viscosity, high provisionality and juxtaposability, as well as its support for progressive evaluation make the pencil-and-pad an ideal prototyping system.

2.9 Chapter Summary

Although spreadsheets have been in existence for quite some time, their basic structure has not changed much, in spite of the fact that some novel concepts have been tried (e.g.,. *Spreadsheet 2000* [8]). The mutability associated with spreadsheets make them quite suited for data visualisation, where such mutability is a requirement. Although the traditional paradigm for data visualisation systems has been dataflow, this paradigm has some problems, which spreadsheets can successfully address. There has been some work in this direction, for example Levoy's "Spreadsheets for Images" [30].

Spreadsheets have many advantages over dataflow systems, especially in the area of scalability. As dataflow diagrams get larger, their connections become more cluttered, resulting in programs that are difficult to maintain and extend.

In order to be able to objectively compare spreadsheets and dataflow systems, as well as to systematically improve on existing visualisation systems, a tool must be used to analyse the cognitive properties of programming environments. The *Cognitive Dimensions Framework* devised by Green [18, 17] provides such a tool; it makes it possible to quantify the differences between existing programming environments, so they can be objectively compared and conclusions drawn about the suitability of these environments for any given task.

Chapter 3

Spreadsheets and the Declarative Paradigm

3.1 Introduction

Although some work has been carried out on functional spreadsheets [11, 12, 16, 30], and on "alternative" uses for spreadsheets (that is, using spreadsheets for functions other than numerical computation) [30, 9], exactly what constitutes a spreadsheet, at its most basic level, has been less investigated. Isakowitz and Schocken [23] argue that the present state of the art in spreadsheet design is quite primitive; there is in general no way to enforce data dependencies between cells, and no way to separate the physical structure of a spreadsheet from its logical structure. Although this may be acceptable for small ad-hoc calculations, the lack of an underlying structure to the spreadsheet makes validation of the model impossible. Work by Panko [48], Isakowitz and Schocken [23], Ronen *et al* [52] and others indicates that this lack of underlying structure is a source of errors. In this chapter we investigate a spreadsheet at its most basic level, and arrive at the conclusion that any spreadsheet is, at its heart, a data visualization system for functional programs.

3.2 Overview of Functional Spreadsheets

The functional programming paradigm stipulates that functions may only read their arguments and generate only the result, without making any changes to their surroundings. Spreadsheet formulas on most commercial spreadsheets [32, 35, 4] normally adhere to this paradigm, if one considers the formula to be the function, the cells referred to by the formula as the function's arguments and

the value displayed in the cell occupied by the formula as the function's result. De Hoon [12] has shown that a spreadsheet can indeed be built from a purely functional perspective, and describes his implementation of a general spreadsheet using the functional programming language *Clean* [11].

3.3 An Abstract View of a Spreadsheet

The main aim of this dissertation is to extend the traditional spreadsheet paradigm to accommodate the demands of data visualisation systems. However, we must first understand spreadsheets at an abstract level. We have to discover the common underlying paradigm that is shared by all programs that merit the name "spreadsheet." Only when this underlying paradigm is understood, can we begin to enhance it.

3.3.1 The Common View of Spreadsheets

As mentioned in Section 2.3 of Chapter 2, the common view of a spreadsheet consists of a rectangular grid of cells, arranged in rows and columns (see Figure 7 for an example). Each of these cells contains either a constant or a function (which is commonly called a *formula*). This grid is usually represented as a group of stored values, implemented as a matrix of values that is kept as part of a "global state" object. The user interfaces of both commercial [32, 35, 4] and experimental [49, 11, 20, 30, 23, 46, 24] spreadsheets mirror this view. Cells are represented as being in a rectangular grid, displaying their value, which is either their contents (in the case of a "data" cell) or the result of their computation (in the case of a "formula" cell). "Formula" cells can also be viewed and edited as formulas by using facilities provided by the spreadsheet application.

3.3.2 A More Structured View

The way of looking at spreadsheets as presented above, although quite adequate for relatively simple spreadsheets (e.g., keeping track of monthly sales), becomes increasingly unwieldy as the complexity of the spreadsheet increases. This is because intercell dependencies are not always readily visible, and so keeping track of which cells will be affected when a particular formula is modified increases in difficulty with the number of formulas in any given spreadsheet. Isakowitz and Schocken [23] approached this problem by using ideas from the field of databases, and came up with the concept of a *dual* view of a spreadsheet. On one side, there is the *physical* layout. This is what has been discussed above. On the other side there is a *logical* view of the spreadsheet. This

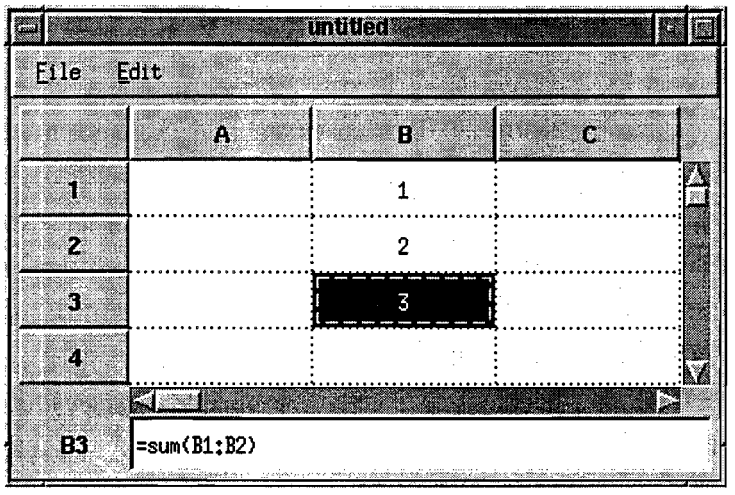


Figure 7: This illustrates a "typical" spreadsheet: It consists of a regular grid of cells, each of which contains a numeric constant, a small caption or a formula. Cells are addressed by their Cartesian coordinates; traditionally, the x coordinate is expressed using letters (base 26) while the y coordinate is expressed using decimal numbers. The text entry control at the bottom displays the contents of the current cell (with its coordinates next to it for reference) so they may be edited.

handles abstract aspects of the spreadsheet, such as data dependencies between cells. The logical view could be likened to a database's schema, and in fact performs many of the same duties. Using an algorithm described in [23], one can break a spreadsheet up into several aspects (called "properties" by Isakowitz), named *schema*, *data*, *editorial* and *binding*. These four aspects, taken as a whole, make up the entire spreadsheet, i.e. *spreadsheet* = *schema* + *data* + *editorial* + *binding*.

- The *schema* stores a formal definition of the spreadsheet's underlying logic, i.e., the formulas and their relationship to each other.
- The *data* is the structured set of constants on which the *schema* property operates.
- The *editorial* is what remains in the spreadsheet after the *schema* and *data* have been extracted, and consists of text strings meant to be comments, etc.
- The *binding* is what defines the logical to physical mapping of the other three properties, using row and column addresses.

The logical structure of a spreadsheet is defined by the *schema* and *data*, while its physical structure is described by the *editorial* and *binding*.

3.3.3 A Functional Look at the Logical Aspects

Isakowitz and Schocken [23] separate the concepts of *schema* and *data*, suggesting that functions and the data they operate on are different. This separation is quite common in current commercial database managers and in procedural languages. Most functional languages, however, treat functions and data as equivalent entities. A re-examination of the logical structure of a spreadsheet, this time from a functional point of view, reveals some interesting properties of spreadsheets.

In spreadsheets, cells can contain either constant values or formulas. A formula can be easily seen to be simply a different representation of a function, since it takes several inputs and only produces one result, without any side-effects. For example consider Figure 7; the formula `=sum(B1:B2)`, which is used to add up the contents of a range of cells, can be formally written as *sum(B1,B2)*, if the function *sum* is appropriately defined (cell ranges can thus be seen to be simply a shorthand way of specifying a group of adjacent cells). Any constant can also be trivially expressed in term of functions, simply by defining a function that always returns the same constant. Therefore it is possible to describe the contents of a spreadsheet cell entirely in terms of functions. From this, it follows that a spreadsheet can be seen as a grid of functions. For the sake of simplicity, each could in turn be called A1, A2, A3, ..., B1, B2, B3, ..., etc. For the example in Figure 7, these functions could be written, using the *scheme* language syntax [26], as shown in Figure 8.

```
(define (B1) 1)
(define (B2) 2)
(define (B3) (+ (B1) (B2)))
```

Figure 8: In this code segment, three *Scheme* functions [26] are defined, named B1, B2 and B3. None of the functions takes any arguments, and functions B1 and B2 return a constant value each (respectively, 1 and 2). Function B3 returns the number obtained by adding the results of calling functions B1 and B2, i.e. *B3()* = *B1()* + *B2()*. Note that in Scheme, expressions are stated in the prefix format, i.e., $a + b \equiv (+\ a\ b)$.

The implications of this observation are not immediately obvious, yet they are quite fundamental. Since a spreadsheet can be completely defined in terms of named functions, it can be adequately described in terms of functions only.

Of course, some information is lost in the process, namely the spatial relationships between the cells. These relationships are used in two ways, namely to make relative references to cells (e.g., "add the contents of the cell just above this one to the contents of the cell just to the left of it"), and to manipulate groups of cells as ranges. Note that both of these are used only when

editing the spreadsheet and do not affect *computations* in any way. However, this functionality can be implemented in other ways, for instance, consider the following implementations of relative references and cell ranges.

Relative References

Relative references can be implemented by applying simple transformations to the formulas as they are moved or copied to other cells. For example, a formula such as `=sum(A1:A3)` would become `=sum(B1:B3)` if copied one cell to the right of its current location. A mechanism would be required to "anchor" some cell references so they are not transformed (for example, the cell reference `$B$4` in *Microsoft Excel* does not change when the cell referring to cell B4 is moved). *This functionality is not a part of the functional part of the spreadsheet, but of the cell editor.*

Cell Ranges

Cell ranges can be implemented as compound types. For example, the cell range named `B2:B4` could be expressed as the Scheme list `(B2 B3 B4)`. Again, since this relies on cell adjacency, it is provided by the cell editor.

3.4 Abstract Definition of Spreadsheets

In the previous section it was observed that a spreadsheet can be described as a rectangular grid of functions, each of which returns either a constant value or a value derived by calling other functions. This means that spreadsheets can be seen as a way to store and manipulate interrelated sets of functions, instead of a mixture of functions and data.

Isakowitz and Schocken's logical view of a spreadsheet [23] is made up of two parts, namely *schema* and *data*. These account for, respectively, spreadsheet formulas and constants. Since we have shown that spreadsheets can be viewed as consisting only of functions, the logical view of a spreadsheet referred to by Isakowitz and Schocken need not be broken down into *schema* and *data*, but can in fact be described as a coherent whole. This unifying of functions and data can be used to provide spreadsheets with the ability to use higher-order functions, which manipulate program code as if it were data (or conversely, can evaluate data as if it were code). A trivial example of this is illustrated in Figure 9. The user types in two numbers into cells A1 and C1, and an operator into cell B1. The formula in cell A2 combines these values into a string that looks like a formula (in this

	A	B	C
1	5	+	3
2	`="="&A1&B1&C1`		`=eval(A2)`

Figure 9: This demonstrates a hypothetical spreadsheet capable of high-order evaluation: the user types in two numbers (in cells A1 and C1) and an operator (in cell B1), and the result of applying that operator to the two numbers is displayed in cell C2. This works by creating a string that looks like the desired formula in cell A2 (in this case, "=5+3") and evaluating that string (in cell C2) as if it were in fact a formula.

case, the string would be `"=5+3"`). This string is then evaluated by the high-order formula in cell A3. The end result is exactly the same as if cell A3 had contained the formula `=5+3`.

In Section 3.3.3 it was shown that if spreadsheet cell B3 contains the formula `=SUM(B1:B2)`, then it could be written, using the *Scheme* programming language, as `(define (B3) (+(B1) (B2)))`. This would create a function (called B3) which when evaluated would call two functions, B1 and B2, and add their results. This means that the inter-cell dependencies that exist between cells B1, B2 and B3 can be completely described using the terminology of function calls. It is not difficult to see that this can be generalised to all intercell dependencies one may find in a spreadsheet.

Since all inter-cell references are calls to named functions, spreadsheets have no semantic need for a grid structure, and the need for a grid organization in spreadsheet cells thus comes into question. It seems that a spreadsheet could be described simply as a set of functions that are ordered on a rectangular grid. However, it should be remembered that the names that are given to functions are completely arbitrary (as long as they are consistent). If one also considers the alternative implementations for relative cell references and cell ranges described in Section 3.3.3, neither of which specifically needs a grid to function, it can be seen that a spreadsheet need not consist of a grid of functions, but merely of an interrelated set of functions.

Isakowitz and Schocken have approached the process of dissecting spreadsheets from a procedural point of view. By tackling this from a functional point of view, as suggested by De Hoon's functional spreadsheet [11] and Eriksson's *Scheme in a Grid* [16], we can fuse Isakowitz and Schocken's *Schema* and *Data* properties into a single entity. Whereas Isakowitz and Schocken separate schema and data, this is not necessary from a functional point of view, and in fact the logical view of a spreadsheet can be described as simply a set of functions which call each other.

This means that the grid layout of spreadsheets is in fact a *user interface consideration*, since it is not necessary for the spreadsheet's proper functioning. Note that this remark is not meant to demean the importance of the grid, but instead to highlight the separation that exists between the user interface of the spreadsheet and the underlying functional mechanism. The physical layout of a

spreadsheet can therefore be seen to be a data visualisation of the set of functions which comprise its logical structure (based on the unique properties of this editing layer, as discussed in Section 3.3.3, we propose to call this the *Adjacency-based Editing Layer* of the spreadsheet).

In Section 2.2, we stated that a dataflow diagram can be seen as a shorthand notation for mathematical functions. If we combine this with our observation that a spreadsheet consists of a usability layer on top of a functional program, then it is apparent that spreadsheets and dataflow programs can be considered as similar in nature (in fact, they are equivalent — see Section 3.5). Given this similarity, we propose to call the underlying computational part of spreadsheets their *Dataflow Logic Layer*. To summarise, we propose that spreadsheets can be seen to consist of two components, which are semantically independent of each other:

- Firstly, there is the *Adjacency-based Editing Layer*, which corresponds to Isakowitz and Schocken's *editorial* and *binding* properties. In addition to handling all the usual editing tasks such as undo, cut-and-paste, etc., this cell editor is what keeps track of cell adjacency. In this way the cell editor can translate cell ranges into sets of cells, and convert relative references to absolute references. This layer also keeps track of cells whose contents have only an informative purpose, i.e., cells used for comments and labels.

- Then, there is the *Dataflow Logic Layer*. This part of the spreadsheet corresponds to the combination of the *schema* and *data* properties (recall that, as we discussed above, these two can be considered to be one and the same), and is what performs the actual calculations. As was shown in Section 3.3.3, the underlying logic is functional in nature. This part of the spreadsheet is completely independent of the spatial relationships between cells.

Therefore, since the physical layout is separate from the underlying logical structure, yet aids in its comprehension and manipulation, it can be classed as a form of data visualisation. An abstract definition of a spreadsheet could therefore be given as:

> A spreadsheet is a finite set of functions (the *Dataflow Logic Layer*) and a grid-based visualisation of them (the *Adjacency-based Editing Layer*), which taken as a whole solve a given problem.

This of course means that the traditional two-dimensional grid is not the only way to manipulate the sort of data normally operated on by spreadsheets. In fact, because of the underlying functional nature of spreadsheets, any mechanism that can be used to visualize and manipulate the flow of data can be used to visualize and manipulate spreadsheets, and *vice versa*.

3.5 Proof of Equivalence of Spreadsheets and Dataflow

In this section, we show that spreadsheets and dataflow systems are, in essence, equivalent to one another. We claim that an arbitrary, correct spreadsheet (i.e., one devoid of circular references) can be transformed into an equivalent pure dataflow diagram (i.e., one devoid of control-flow constructs) and *vice versa.* As proof, we present algorithms to do these transforms.

Spreadsheet to Dataflow Consider a correct spreadsheet S, which consists of a grid containing populated ("full") and unpopulated ("empty") cells, the former being called c_i. The algorithm in Figure 10 describes how the set of cells $c_i \in S$ can be converted into an equivalent pure dataflow diagram D, which consists of a set of nodes n_i joined by a set of directed arcs a_j. We define all n_i as containing functions of the form $f(a_1, a_2, ..., a_n)$, where a_j is the j^{th} arc pumping data into n_i.

input: a correct spreadsheet S consisting of a set of cells c_i
output: a pure dataflow diagram D consisting of a set of nodes n_i joined by arcs a_j

begin
  foreach populated spreadsheet cell c_i:
    create a dataflow node n_i, labelling it with c_i's spreadsheet grid location, e.g., D4.

  foreach populated spreadsheet cell c_i:
  begin
    find the node n_i that corresponds to c_i (n_i will be labelled with c_i's grid location).
    if c_i does not contain a formula:
      write the contents of c_i into n_i.
    else begin
      foreach cell c_r that is referenced by the formula in c_i:
      begin
        find the node n_r that corresponds to cell c_r.
        draw a directed arc a_j from c_r to c_i, and label it in_j.
      end
      write the formula from c_i into n_i, replacing each cell reference c_r by in_r.
    end
  end
end

Figure 10: This algorithm can be used to transform an arbitrary correct spreadsheet into a pure dataflow diagram.

Dataflow to Spreadsheet Consider a pure dataflow diagram D, consisting of a set of nodes n_i joined by directed arcs a_j. The algorithm in Figure 11 describes how D can be converted into an equivalent, correct spreadsheet S consisting of a set of cells c_i. Again, we define all n_i as containing functions of the form $f(a_1, a_2, ..., a_n)$, where a_j is the j^{th} arc pumping data into n_i.

input: a pure dataflow diagram D consisting of a set of nodes n_i joined by arcs a_j
output: a correct spreadsheet S consisting of a set of cells c_i

begin
  let S be an unpopulated spreadsheet.

  foreach dataflow node n_i:
    give n_i a unique label consisting of a letter and a number, e.g., D4.

  foreach dataflow node n_i:
  begin
    let c_i be the spreadsheet cell in S with grid reference equal to n_i's unique label.
    foreach arc a_j pumping data into n_i:
    begin
      let n_j be the dataflow node a_j pumps data from.
      let $cellref_j$ be n_j's unique label.
    end
    write the function from n_i into c_i, replacing each reference to a_j with $cellref_j$.
  end
end

Figure 11: This algorithm can be used to transform an arbitrary pure dataflow diagram into a spreadsheet.

Since these algorithms can be used to convert any correct spreadsheet into an equivalent pure dataflow diagram and vice-versa, they demonstrate that these two notations are in fact equivalent.

3.6 Chapter Summary

Although spreadsheets have, since their invention in the late 1970's, always been represented as two-dimensional grids containing values and formulas, we have shown that this view is not necessarily the most fundamental one. Specifically, we have found that the grid representation of data, although fundamental to the user experience of using a spreadsheet, is a separate entity from the underlying

computational engine.

We base our observations on the work of Isakowitz and Schocken [23], who have stated the need for an underlying structure to spreadsheets, in order to be able to verify their correctness. Borrowing ideas from database theory, they have suggested a way to break down a spreadsheet into four distinct components, namely *schema*, *data*, *editorial* and *binding*.

However, Isakowitz and Schocken have approached the problem from a procedural point of view. By tackling the problem from a functional point of view, as suggested by De Hoon's functional spreadsheet [11], we have found that spreadsheets may be broken down quite naturally into just two components, one logical (the *Dataflow Logical Layer*) and one physical (the *Adjacency-based Editing Layer*). Although both of these layers are necessary for the functioning of the spreadsheet, they are semantically independent of each other. We then arrived at the conclusion that dataflow diagrams (which can be seen as a graphical notation for mathematical functions) can be considered as being similar in nature to spreadsheets (which can be seen as a user interface over functional programs). We further extended this observation by proving that spreadsheets and dataflow diagrams are equivalent to one another.

Chapter 4

An Extended Spreadsheet Paradigm

4.1 Introduction

In Chapter 2 we described instances where spreadsheets have been used for data visualisation, notably work by Levoy [30] and Chi *et al* [9, 10]. Although these programs look and feel very much like traditional, declarative spreadsheets, both programs require spreadsheet formulas to be expressed using the imperative language *Tcl* [45]. Additionally, the use of imperative programming techniques is encouraged, such as having cells directly modify the contents of other cells. In addition to being the source of quite obscure bugs (what happens when two cells store different values into the same cell?), this use of an imperative language creates a paradigm mismatch with the declarative nature of spreadsheets, which we described in Chapter 3.

During the analysis of the spreadsheet paradigm we made in Chapter 3, we showed that a spreadsheet is a visualisation environment for functional programs; therefore it is possible for us to make use of functional language features to extend spreadsheets to meet the demands of data visualisation without needing to use mismatched paradigms.

In this chapter we show the inadequacies of the traditional spreadsheet paradigm when it is applied to data visualisation. We then illustrate how, by the addition of three basic techniques, the spreadsheet paradigm can meet the requirements needed to perform data visualisation tasks on non-trivial datasets.

4.2 Spreadsheets and Data Visualisation

Spreadsheets can be considered to be effective tools for data visualisation because they allow data to be easily input, dynamically updated and readily viewed [34]. They provide a flexible and easy-to-learn environment for user programming [10]. Additionally, spreadsheet developers need not worry about many traditional programming details such as memory management or input/output. The fact that spreadsheets automatically keep track of data dependencies between cells and update relevant cells when necessary frees the spreadsheet modeler from this task, preventing many errors familiar to programmers of imperative languages.

Spreadsheets also elegantly address some of the problems that are met when visualising multidimensional data, such as the mutable nature of visualisations and the need to have different yet simultaneous views of the same data [9].

However, spreadsheets have certain design features which conflict with the needs of data visualisation. These arise mainly from the fact that each spreadsheet cell can contain only a single object. Although the use of *cell ranges* solves the problem where only a few dozen values are dealt with, this breaks down when thousands of values must be manipulated. In that case (common for many data visualisation tasks), the spreadsheet would have to be several thousand cells wide or deep, making navigation a difficult exercise.

4.3 The Extended Spreadsheet Paradigm

To overcome the problems discussed above, we need to extend the spreadsheet paradigm to address the needs of data visualisation systems, while maintaining the simplicity and declarative nature of traditional spreadsheets. This extension can be divided into three logical parts, namely the handling of large datasets; the use of lazy evaluation and the replacement of the traditional formula language by a fully-featured functional programming language.

4.3.1 Handling Large Datasets

Most programming systems deal with the concept of processing multiple data items by using ideas such as recursion or iteration, i.e., taking the elements of a set and processing them one after the other. While these methods have many advantages, they are ill-suited to the spreadsheet paradigm, where looping constructs cannot be adequately expressed. Instead, numbers in traditional spreadsheets are manipulated individually, and where some form of grouping is desired, cell ranges can

easily be formed by the user, and operations such as finding the average of a given set of numbers can be accomplished by idioms such as `=average(B12:B72)`. Although this works quite well for financial statements, data visualisations normally deal with data sets containing tens of thousands of items. At this level, the range paradigm of dealing with datasets breaks down, usually due to the unwieldiness of rows or columns containing thousands of items.

Our way to solve this problem was to extend the traditional spreadsheet paradigm to allow sets of items (e.g., in the form of lists) to be stored in each cell, as opposed to single items. In this way, an entire dataset could be contained in a single spreadsheet cell; in the vast majority of cases, data visualisation is not concerned with the transforming of individual values, but rather of entire datasets. By storing a dataset per cell, large amounts of data can be processed without needing to clutter a spreadsheet with thousands of numbers that have little individual meaning.

Therefore, the concept of ranges is done away with, since it is no longer needed. In effect, each cell could be thought of as containing a range within itself.

4.3.2 Lazy Evaluation

As mentioned above, the problem of storing and manipulating large datasets can be eliminated by giving each spreadsheet cell the ability to store an entire dataset. However, there is still the problem of efficiency. If a spreadsheet that has several hundred thousand values stored in it must be recalculated, that recalculation can take a substantial amount of time. Traditional spreadsheets do not have this problem, since even a large financial statement typically will not have more that a few thousand values stored in it. If we extend the spreadsheet paradigm to allow the storage of hundreds of thousands of values, we must find a way to deal with the associated increase in computational time.

An important part of data visualisation is to eliminate information from a dataset, in order to obtain the underlying patterns in the data. This can take one of two forms, either elimination of detail in order to be able to see patterns in the overall structure of the data or "zooming in" to be able to examine localised properties of the dataset. Both of these techniques sacrifice the amount of data displayed to allow multi-resolution analysis of the physical or mathematical process being visualised. Since these techniques allow researchers to selectively examine any given aspect of the data, eliminating all other aspects which may be considered as distracting, they are quite frequently used when visualising datasets.

This means that, in many cases, data does not make it all the way from the database to the screen. In those cases it would be useful to know what values are not going to be needed by

later calculations, in order to save time by simply not calculating them. Although it is possible to determine which values are not going to be needed and build this into the equations used to transform the data so that it may be visualised, these constraints usually complicate the equations. These complications make the visualisation pipeline harder to debug and can impose a performance penalty if not properly expressed. Users of data visualisation systems are usually more concerned with understanding their data than with learning optimisation tricks; therefore it is important that data visualisation systems have the ability to exploit data access patterns to reduce computation times.

The functional programming technique known as *lazy evaluation* is a simple, yet powerful way of exploiting data access patterns, since it is not necessary to build any constraints into the equations to account for the fact that only a fraction of the dataset is going to be transformed. All equations can be written assuming that the entire dataset will be processed; should any function in the evaluation chain need only a subset of the data being passed to it, only the required data will be passed to it. Because of the lazy evaluation mechanism, the functions supplying the data do not need to be modified to provide only certain data points — the flow of data through the inner levels of the expression is dictated by its outer levels.

The concept of lazy evaluation is probably best explained with an example. Consider the common case of generating an image from a dataset, scaling it down to 25% of its size and displaying it. This could be functionally expressed as *display(scale(render(read())))*. One way to efficiently implement this would be to have functions *read()* and *render()* take as arguments which points need to be rendered; *scale()* would then request the appropriate pixels from *render()*, which would in turn request the appropriate data points from *read()*. It would not be unreasonable to presume that in such an implementation a noticeable percentage of the code would be dedicated to this necessary bookkeeping. On the other hand, a naive implementation could simply read in the entire dataset, pass it to *render()*, which would generate a complete image, which would in turn be given to *scale()*, which would pick the pixels it needs. This implementation would be much simpler — each function would perform its task without regard to how the data it generates is used by the calling function. Unfortunately, this implementation would also be very inefficient: 75% of all data points read in would be discarded, along with 75% of all the calculated pixels.

However, consider what would happen if the naive implementation were evaluated lazily. Superficially, the *scale()* function would call *render()*, which would in turn call *read()*. Recall that all of these functions are written in such a way that they take as arguments and return entire datasets, i.e., there is no explicit way to state which data points are required. The difference between lazy and

eager evaluation lies in *what* is returned by these function calls. Under eager evaluation, as we have seen above, these function calls cause *work* to be done and return *passive* data, i.e., the requested data structures. However, under lazy evaluation, these function calls return immediately — the real work is performed *when the returned data structures are evaluated.* In other words, under lazy evaluation, data structures returned by functions are *not* passive. When *scale()* queries a pixel from the bitmap returned by *render()*, the function *render()* is called *implicitly.* The very act of querying the bitmap is what causes work to be done, in this case calling the function *render()*, since that is the function that returned the bitmap.

Note that all of this is done *transparently* — as far as the programmer is concerned, *render()* generates and returns an *entire* bitmap, and it does so *when it is explicitly called.* The reality of the situation, i.e., that the requested function call never took place, and that evaluating the return value is in fact what triggers the function call, is kept hidden. In the process of calculating the pixels it is asked for, *render()* evaluates individual data points from the dataset returned by *read()*. This evaluation causes *read()* to also be called implicitly, and query the database to obtain only the needed values. To summarise, function calls *do not do work* under lazy evaluation — instead, they return *active* data structures which, *when evaluated,* do the actual work. This means that a more straight-forward, "naive" implementation has the potential to be as efficient as an optimised implementation. The advantage to the user is a program which is easier to read (and hence easier to debug) than a highly-optimised implementation, but which achieves high performance.

Since lazy evaluation defers calculations until the results are used, it is possible to have a function which operates on arbitrarily large lists, including those with an infinite number of items (such as $\mathbb{N}$). An example of the usability of this would be the ability to ray-trace an infinite grid of spheres at any resolution. Since the subset of spheres that is rendered (and hence evaluated) is dependent on the resolution of the display, the only parameters that would need to be changed to obtain a more accurate representation would be the horizontal and vertical resolution of the generated bitmap. The (infinite) scene description would remain unchanged.

4.3.3 Using a Functional Formula Language

In Chapter 3, we showed that a spreadsheet can be viewed as an environment for the development of functional programs. Indirectly, this means that the formula languages normally used inside spreadsheets are functional in nature (since spreadsheet formulas do not have any side effects, they can be considered as special-purpose functional languages).

Therefore the basic spreadsheet paradigm would not be significantly altered if, instead of the

formula languages normally associated with spreadsheets (using idioms such as `sum`, `average`, etc.), a fully-featured functional language were used. Additionally, the use of a fully-featured functional language would give the user the benefits of generality and the extensibility of the language, as opposed to being constrained by the primitives provided by the spreadsheet software. The user could also benefit from functional language constructs (e.g., high-order functions), which can provide much more expressibility than is possible with other programming paradigms [21]. An example of the use of high-order functions could be using the data being visualised as the building blocks for a function, which would be built up and evaluated "on the fly" (See Figure 9 in Chapter 3). This sort of functionality is not directly available in traditional spreadsheet systems, and hence any attempts to perform this type of calculation are doomed to either failure or categorization as "hacks."

4.4 Chapter Summary

In this chapter, we have shown that, although spreadsheets are a desirable medium for the development of data visualisations, the traditional spreadsheet paradigm is unsuitable for data visualisation. Building on the theoretical framework developed in Chapter 3, we have extended the spreadsheet paradigm to address the deficiencies of the traditional spreadsheet with regard to data visualisation. Here, we summarise the differences between the traditional spreadsheet paradigm and the extended spreadsheet paradigm that has evolved from this discussion. The extended spreadsheet paradigm differs from the traditional in three ways:

- Firstly, each cell does not contain a single value, but a list of values. This allows the storage of tens of thousands of values in a spreadsheet, while keeping its size manageable.
- Secondly, the lists of values are manipulated using lazy evaluation, which allows for the efficient manipulation of potentially infinite datasets.
- Finally, the extended spreadsheet uses a fully-featured functional programming language as the formula language. Using a fully-featured functional language (instead of the formula languages used in most commercial spreadsheets) makes sense since we have shown that spreadsheets are already functional in nature, and use of a fully-featured functional language provides the user with several advantages over formulas.

Together, these three additions extend traditional spreadsheets so they can efficiently handle the large datasets that are routinely manipulated during data visualisations. This is done without sacrificing the simplicity and declarative nature of the traditional spreadsheet paradigm.

Chapter 5

ViSSh, a Data Visualisation Spreadsheet

5.1 Introduction

In Chapter 4, we described how the traditional spreadsheet paradigm can be extended to accommodate the demands imposed by data visualisation. We have built a prototype to test this extended paradigm, and verify its usability. The design of this prototype, called ViSSh (short for **Vi**sualisation **S**pread**Sh**eet) is described in this chapter, both from the user's perspective and functionally from our perspective as its designers.

5.2 The User's Perspective

From the user's point of view, ViSSh seems at first quite different from most commercial spreadsheets. However, the differences are mostly superficial, arising mainly from the need to deal with large datasets. A cursory examination of the ViSSh spreadsheet in Figure 12 reveals the following:

1. The spreadsheet is made up of cells that are arranged in a rectangular grid and referenced in the same way as those in "traditional" spreadsheets.

2. The cells look nothing like traditional spreadsheet cells, containing miniature control panels instead of text or numbers.

The first of these two observations underlines the basic similarities between ViSSh and traditional spreadsheets. Users of other spreadsheets will, once they have adapted to the slightly different feel of the system, be able to apply their experience to ViSSh.

The second observation highlights the main differences between traditional spreadsheets and ViSSh, which reflect the extended spreadsheet paradigm, and will be discussed below.

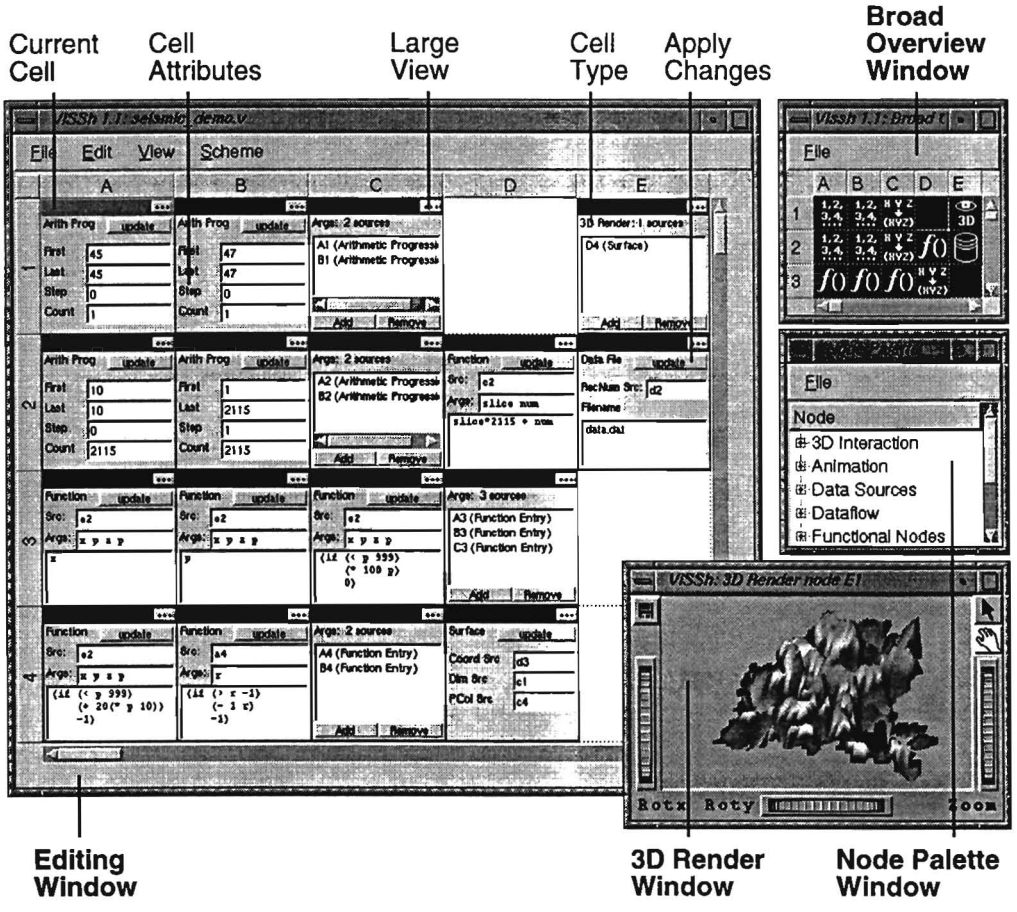


Figure 12: This is ViSSh, our Data Visualisation Spreadsheet, being used to visualise seismic disturbance data. The *Current cell* is the one being edited at any given time. Each cell has a *Cell Type* that governs its behaviour, which is modified by the user-specified *Cell Attributes*. Each cell also has a *Large View*, accessible by clicking on the "..." button, that provides on-line help and a larger editing/display area. Spreadsheets are edited by using the *Editing Window*; new cells are added by dragging them from the *Node Palette Window* and dropping them into a cell slot in the Editing Window. Since spreadsheet cells are physically quite large, users can use the *Broad Overview Window*, with its smaller icons, to view a larger portion of the spreadsheet.

5.2.1 Dealing with Large Datasets

The ability to deal with large datasets is crucial in a viable data visualisation system. ViSSh implements the extended spreadsheet paradigm described in Chapter 4, and as such is capable of handling

potentially infinite datasets. ViSSh places more emphasis on operators than operands. This is both because of the large volumes of data (making the traditional "cells display their own contents" technique impossible in the general case), and because there is no single right way of representing an arbitrary data set stored in a cell. Instead, there are specialised cells that are used to view data in different ways. For example, in Figure 12 the contents of cell D4 are being displayed by cell E1. Although this functionality is partially implemented in some commercial spreadsheets (e.g., *Microsoft Excel's* graphing tools), the amount of interaction that is possible when 3D interaction cells (described below) are added to a scene is far beyond any current commercial implementation.

5.2.2 Functional Programming Model

Whereas most spreadsheets use a fairly rudimentary language to express their formulas in, ViSSh uses the programming language Scheme [26]. This gives users a lot more expressive power, allowing constructs such as high-order functions, which are simply not expressible using traditional spreadsheet formulas. Note, however, that users are not constrained to the use of Scheme — expressions may also be specified in infix form (i.e., a user may equally validly define an expression as "`m * x + c`" or "`(+ (* m x) c)`").

An example of the expressive power of high-order functions is illustrated in Figure 13. This example demonstrates how several functions can be stored in a spreadsheet cell (inside a list), and one of these functions can be selected and applied to other data. In Figure 13, Cell C1 is a "function evaluator" — it selects an item from a list given to it, based on an index passed by a third cell, and evaluates it as a function (if the list item is a list, then it is evaluated as a Scheme expression; if it is a string, then it is evaluated as a "normal" mathematical expression). Cell A1 provides a list of 10 consecutive integers, while cell A2 provides a list of 3 functions. Which function is used to operate on the list of numbers is indicated by cell B1, which in this case is selecting the third function (list items are numbered starting from zero).

ViSSh makes use of lazy evaluation, as described in Section 4.3.2. This gives ViSSh the advantages described in that section, particularly the ability to efficiently deal with very large datasets without the need to explicitly code optimisations into the equations. Evaluation is performed immediately when the spreadsheet is edited; there is no "background processing" and recalculations proceed to completion before the user is allowed to do further editing.

One of the advantages of a functional programming system is its enhanced modularity [21], which from a user's point of view translates into a system that easily allows code reuse. In many cases data visualisation benefits greatly by code reuse, and most data visualisation systems allow

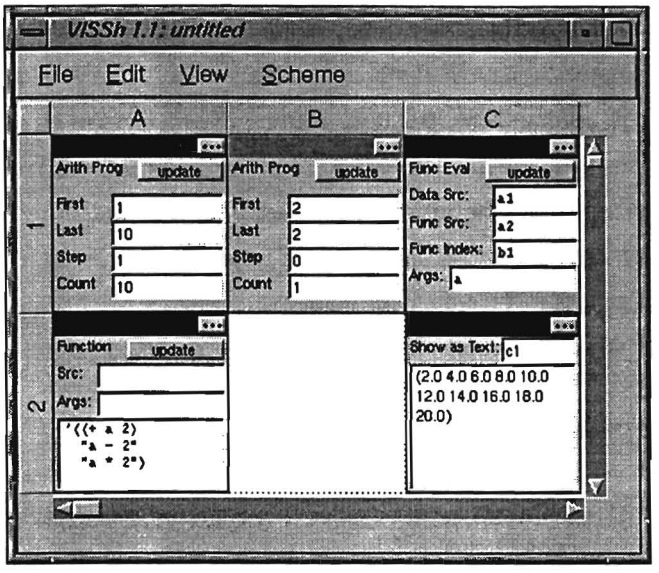


Figure 13: This spreadsheet demonstrates a use of high-order functions — in this case, selecting an item from a list and applying it (as if it were a function) to another list. This spreadsheet generates a list containing the first 10 natural numbers and applies a function, selected from another list by list position, to these numbers. The list of numbers is generated by cell A1, and cell B1 is used to indicate which function must be applied from those given in cell A2 (the functions in inverted commas are infix arithmetic expressions, while those in parentheses are Scheme expressions). The function application is done by cell C1, and the results are displayed by cell C2.

this practice by the use of some form of function libraries. In order to provide this facility without breaking the spreadsheet paradigm, ViSSh allows several spreadsheets to be linked together in a functional manner. This is achieved by the use of three specialised spreadsheet cells which, as a group, implement the idea of function calls in the context of spreadsheets (see Figure 14).

The user's view of functional linkage between two spreadsheets is quite simple: the "helper" spreadsheet has an "Argument" cell, which collects the arguments that the function implemented by the spreadsheet takes (recall that each cell generates a list). It also has a single "Result" cell, which exposes the final result calculated by the spreadsheet. The master spreadsheet has a "Subsheet" cell which behaves much like a function evaluation cell, but which "calls" the helper spreadsheet by evaluating its Result cell. This causes the normal evaluation process to propagate through the helper spreadsheet, until the Argument cell gets evaluated. This cell pulls in the arguments passed from the master spreadsheet, and the results then propagate forward until they reach the Result cell. The results then are given to the SubSheet cell, which returns them as its own results. With this simple mechanism, several spreadsheets can be linked together in a functional manner, providing the user

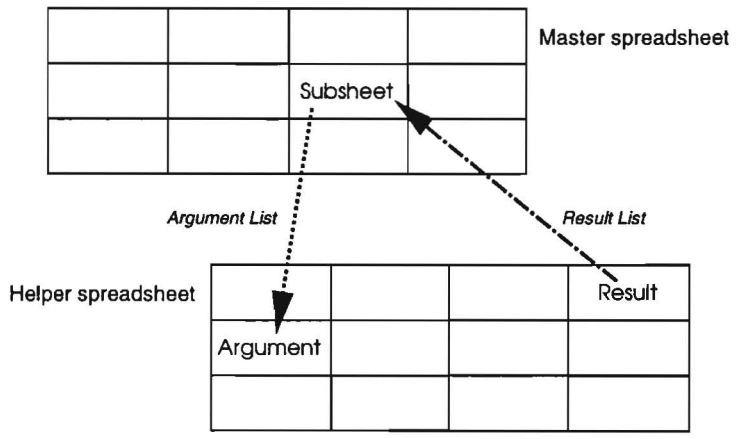


Figure 14: This illustrates the mechanism whereby several spreadsheets may "call" each other. The "Master" spreadsheet contains a Subsheet cell, which forwards all data sent to it to the "Helper" spreadsheet. This data arrives into the Helper via an Argument cell. When the computation is complete, the result is sent back via a Result cell.

with a simple way of partitioning the problem being solved, and allowing code reuse. The inter-sheet communication is two-way: when a cell in the subsheet is modified, this information is sent to the master spreadsheet, which then behaves as if the SubSheet cell linking the two spreadsheets had been modified. The client-server model also allows for future implementations of the system to be built using a distributed model, possibly even allowing for multi-user interaction. Section 8.4 further discusses these possibilities.

Users may also extend the system by writing new functions in Scheme, which are dynamically loaded into the spreadsheet at runtime and can be used by spreadsheet cells.

5.2.3 3D Interaction

Users may also interact with the 3D environment that is displayed in 3D Render windows. This is accomplished via the use of specialised cells which are data sources, and also have a 3D representation. This form of direct interaction can be more useful that textual editing of parameters, especially when qualitative measurements are desired, and the exact setting is not as important as a feel of how "large" or "small" a given value is. An example of this type of cell is the "1D Dragger," which is represented by a double-headed arrow and which returns a single scalar whose value depends on how far along the longitudinal axis the arrow has been dragged by the user (see Figure 15). Another example is the *3D pick node*, which allows the user to select any given 3D object in a 3D render window. These implement level-4 liveness [6] using the model described by Burnett as *LazyLM* [6], with the expiration time of any event-driven data point being set to the time the next event arrives.

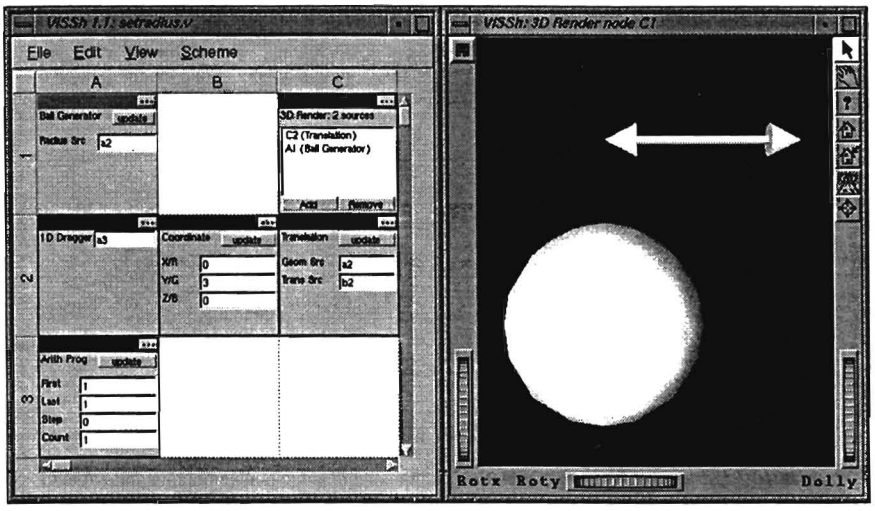


Figure 15: This spreadsheet illustrates the *1D Dragger* 3D interaction cell. Here, a 3D sphere is displayed, together with a 1D dragger. Dragging the double-headed arrow in the 3D display forwards and backwards changes the radius of the 3D sphere in real-time. The spreadsheet is quite simple: the sphere is generated by cell A1, which takes its radius from cell A2 (the 1D dragger). Cell A2, in turn, takes its initial value from cell A3; this means that the sphere always starts with a radius of 1 unit. The dragger is translated to (0,3,0) by cell C2 before being displayed, together with the sphere generated by cell A1, by cell C1.

5.2.4 Debugging Aids

A common problem with spreadsheets is that users can become "lost" when navigating through large spreadsheets which contain many similar subsections. Also, Nardi [41] has found experimentally that users prefer to be able to view as much of a spreadsheet as possible without scrolling. This problem is especially acute in the case of ViSSh, since each cell is rather large and hence the number of them that can be displayed at any one time is fairly small (typically 7×5 cells in a 1024×768 display). This problem is tackled using two different tools, one for each aspect of the problem: grid navigation and dataflow.

Grid Navigation

In order to find out which part of the spreadsheet grid is being viewed though the editing window, as well as locating the current cell cluster in relation to the rest of the spreadsheet, the user can call up a window which contains a miniature version of the spreadsheet (Figure 16 illustrates this "Broad Overview" window, which in this case corresponds to the spreadsheet in Figure 12). Each cell of this contains a small (32×32 pixel) icon describing the function of the cell in the larger spreadsheet,

wherein the area of the larger spreadsheet that is visible at any time is shaded in the small-scale spreadsheet. Since this can display a much larger portion of the spreadsheet (about 30×20 cells in a 1024×768 display), the spreadsheet navigation problem is greatly aided by the use of this window. The "Broad Overview" window can be scrolled independently of the main spreadsheet window, and the cells which are currently visible in the main window are shown in a highlighted manner. Clicking on one of these cells scrolls the main spreadsheet so that the cell becomes visible. In this way, the "Broad Overview" window acts as a road-map, with the icons representing the function of each cell being used as landmarks. Although we could have used a lenticular interface such as the one employed by *TableLens* [50], we decided to instead use a separate "map" window to aid navigation because in this way users can maintain the current view in the main editing window while they search for other cells in the independently-scrolling Broad Overview window. The main editing window could in fact be considered to be a lens over the "Broad Overview" window.

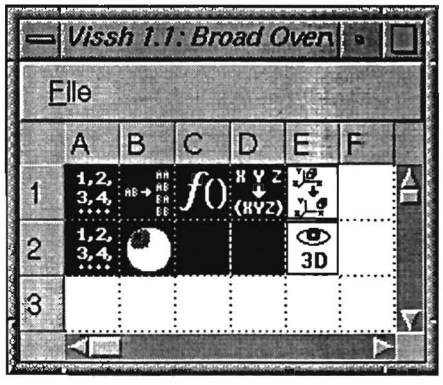


Figure 16: This illustrates the window that is obtained by using the "Broad Overview" feature of ViSSh when the spreadsheet in Figure 19 is being edited. The white-on-black cells represent the portion of the spreadsheet that is visible through the editing window.

Dataflow

The other aspect of spreadsheets that can cause much frustration is keeping track of which cells depend on what other cells. As was discussed in Section 3.5 of Chapter 3, spreadsheets and dataflow systems are equivalent in nature. ViSSh makes direct use of this fact by providing the user with a window which contains a data flow diagram in which the nodes are iconic representations of the spreadsheet cells (Figure 17 illustrates the "Show Dependencies" window which corresponds to the spreadsheet in Figure 19). This idea, in itself, is not new (for example, *Microsoft Excel* [35] has an

option to trace references to individual cells). However, we believe that this particular implementation is more useful than previous attempts (e.g., *Microsoft Excel* and *Nattospread* [55]), since it is not limited to the subset of cells visible through the editing window; all the cells in the spreadsheet are included in the diagram. This prevents dependencies from being "hidden" by the fact that not all dependent cells fit in the editing window. The model we have followed can be described as a combination of Igarashi's *Static Global* and *Animated Global* views [22]. The main difference between out work and Igarashi's is that the view is displayed in a different window, so as to minimize clutter in the main editing window. This allows us to convey all the information in both of Igarashi's views in a single display, without needing to resort to animation.

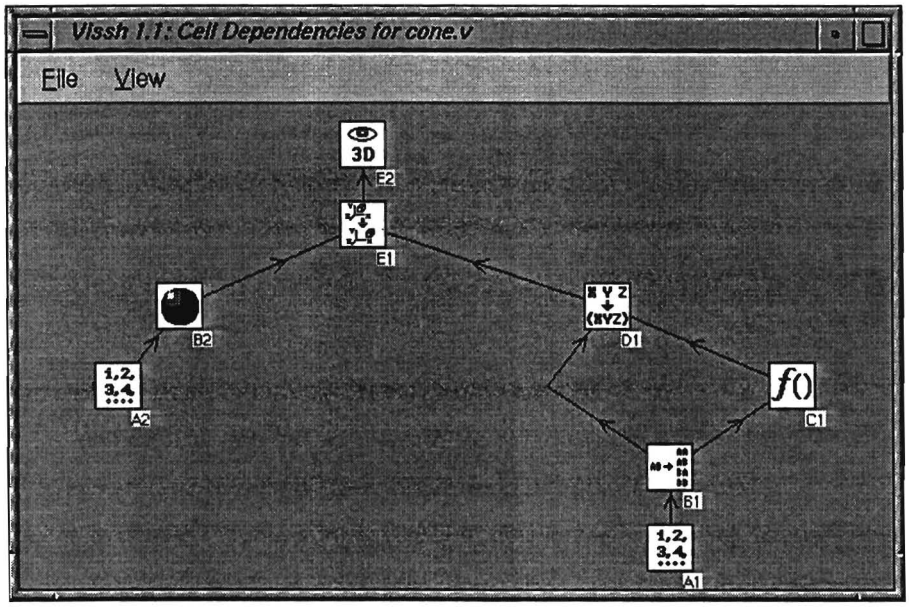


Figure 17: This illustrates the intercell dependencies of the spreadsheet in Figure 19.

We have used the same glyphs to describe spreadsheet cells as the ones used in the "Broad Overview" window, in order to reduce the cognitive load on the user. The dependency graph is editable, and is kept constantly synchronised to the main spreadsheet window so that the current state of the spreadsheet is always visible. Editing is accomplished by selecting a cell on the dataflow diagram, this selects it in the spreadsheet view so it may be edited. This dataflow diagram provides users with the advantages of the dataflow metaphor, while the spreadsheet editing environment shields them from the cluttering associated with medium to large dataflow editing environments.

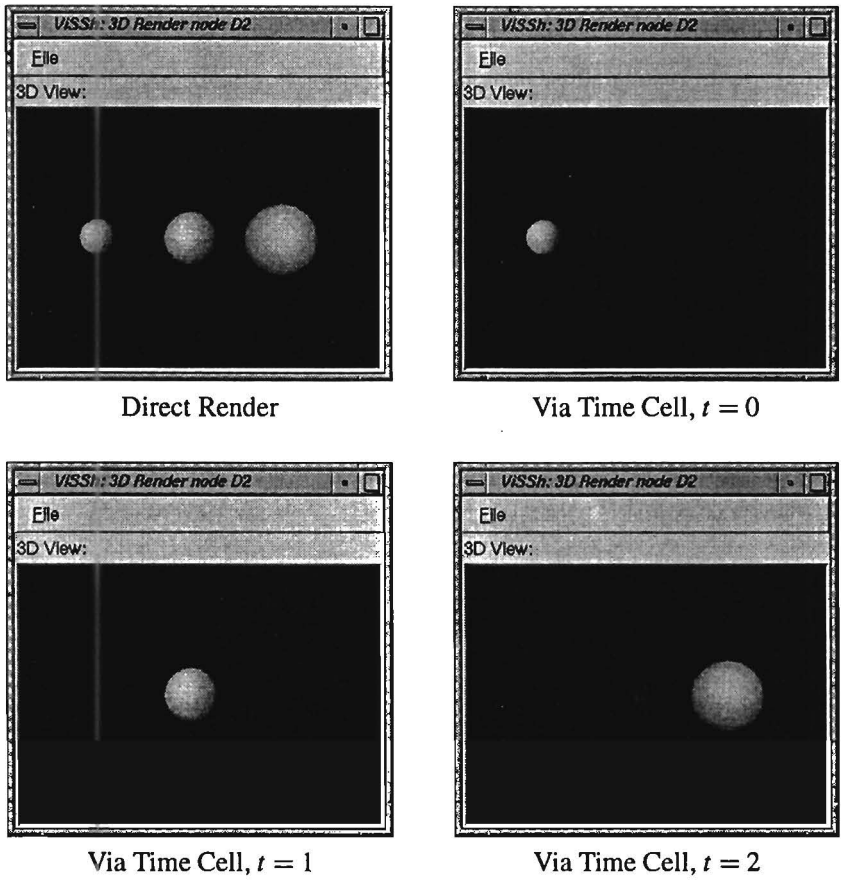


Figure 18: This shows the effect of adding a Time Cell to a simple visualisation. The image on the top left represents the scene rendered "as is," while the remaining three images show how the 3D objects are "let through" as time passes.

5.2.5 Animation

As was mentioned in Section 5.2.1, ViSSh has no concept of looping or iterating through a dataset. ViSSh instead has the ability to store entire datasets in a single cell, and to perform operations on these datasets. Since the system is functional in nature, the order of evaluation of single items in a dataset is essentially arbitrary (with hardware support, data items could even be operated on in parallel).

The concept of animation is brought into this mechanism by simply imposing an order on the evaluation of items in a dataset, and inserting a known delay between the evaluation of successive

items. Since, as described above, ordering is unimportant, the imposition of order on the evaluation mechanism has no bearing on the correctness of the result. Similarly, the introduction of a time delay has no effect on the result of the calculation. Therefore, this mechanism implements animation by overlaying order of evaluation and time delays on the purely functional spreadsheet paradigm, without affecting the functional nature of the paradigm itself.

The implementation is deliberately primitive, in keeping with the "basic building block" philosophy of ViSSh. A specialised cell (called the "time cell") takes a set of items, and outputs each item in succession, each one being output after a given time delay has elapsed. Figure 18 illustrates the effect of adding a time cell to a simple visualisation (in this case, a sphere whose radius depends on its position along the x axis).

5.3 Spreadsheet Cell Taxonomy

Before we begin our in-depth discussion of ViSSh, we should take a closer look at the different types of spreadsheet cells that are available (see Section A.4 of Appendix A for a complete cell reference). Generally speaking, there are three major types of spreadsheet cell: data sources; data sinks and operators. Additionally, there are two minor types associated with each major type: computational and scene. Computational cells operate on data values that can be manipulated algorithmically (such as numbers or strings), while scene cells operate on visual data (such as cones and cubes).

Data sources are the cells that bring data into the spreadsheet. These include such cells as the ball generator and the data file reader. The ball generator creates a set of spheres that fit the constraints given by the spreadsheet programmer, while the data file reader imports data obtained from an external source. Data sources do not usually depend on any data already in the spreadsheet.

Data sinks are the complement of data sources: data that has flowed through the spreadsheet is displayed by these cells. Data sinks generate no data. An example data sink is the 3D Renderer cell. Because of the lazy evaluation used by ViSSh, Data sinks are responsible for initiating all recalculations.

Operator cells are the largest category of spreadsheet cell. They are responsible for all computations and geometrical transforms. An example of a computational operator cell is the List Length cell — this cell counts the elements in the list given to it by the cell it depends on and returns to its dependent cells that number. A scene operator cell would be a Translator cell; this cell retrieves a scene containing 3D objects from the cell it depends on and returns to its dependent cells a scene like the one it retrieved, except that all the objects in the scene have been translated (moved) by the

given amount.

All cells in the spreadsheet belong to one of these three categories, which naturally reflect the flow of data through the spreadsheet. Data arrives into the spreadsheet through one or more data sources, flows through and is transformed any number of times by operator cells, and eventually is displayed by a data sink; the flow of data stops there.

New cells are added to a spreadsheet by using the *Node Palette* (see Figure 19), which allows users to drag cells (indexed by functionality and sorted alphabetically) from itself into spreadsheet cells, overwriting any cell already present in the cell being dropped into. New cell types can be added by modifying the ViSSh executable, as described in Appendix B.

5.4 A Brief Demonstration

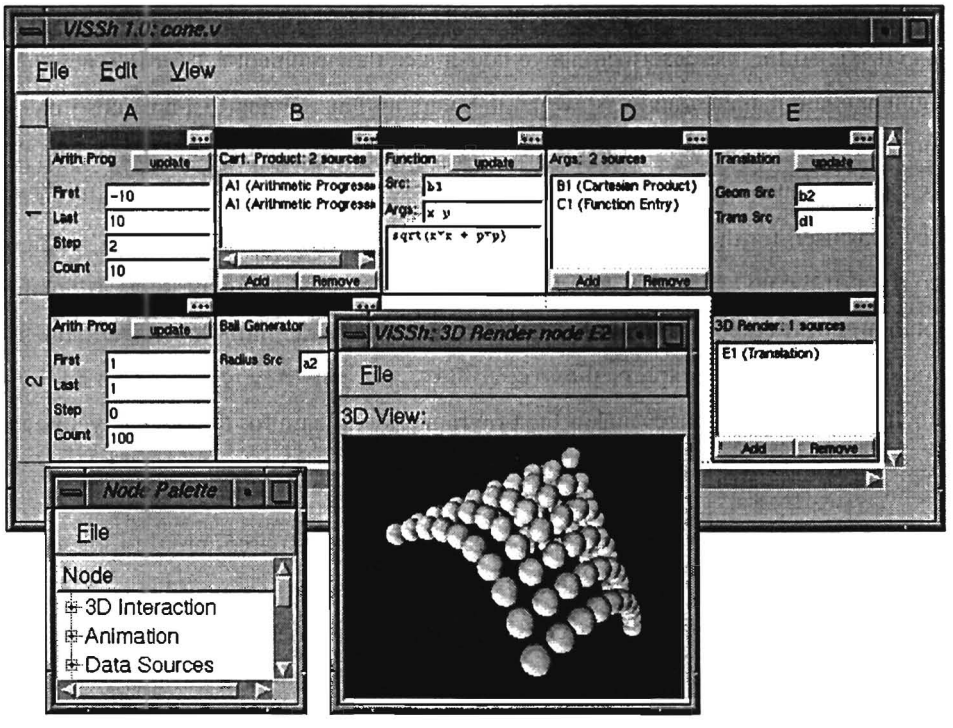


Figure 19: This is a spreadsheet used to visualise the parametric cone equation. The large, rectangular window contains the actual spreadsheet, while the square window in the foreground is a rendering of the cone itself, and the small window on the bottom left is the "node palette" used to add new cells to the spreadsheet.

In order to demonstrate the "feel" of ViSSh, we present a simple example. The aim is to make a cone out of spheres (using the parametric cone equation) and display it in three dimensions (see

Chapter 6 for more complex visualisations). Figure 19 demonstrates what the completed spreadsheet looks like. The key to editing with ViSSh is the "Node Palette," illustrated in Figures 12 and 19. From here the different nodes are selected from their groupings, dragged and dropped into the spreadsheet cell where the user wishes them to be. The spreadsheet was constructed as follows:

1. An "Arithmetic Progression" node was dragged from the node palette into cell A1, and the relevant parameters were entered into it. In this case, we want a set of numbers running between -10 and 10, in steps of 2.

2. A "Cartesian Product" node was dragged from the node palette into cell B1. Cell A1 was added twice as a source, so the output of this node will be the list {(-10 -10) (-10 -8) ... (10 8) (10 10)}.

3. A new "Function" node was dragged into cell C1. The source of the arguments was specified as cell B1, so the list described above constitutes the argument set for this new node. The arguments are named x and y, so when the pairs in the argument list are substituted into the equation, the first item in the pair will be substituted for all the x's and the second for all the y's. The actual function is then entered (since the actual cell is rather small, an expanded view was used for this. The expanded view was activated by clicking on the "..." icon in the top right of the node). In this case the function was entered using the standard mathematical notation, but it could also have been entered in the programming language Scheme [26] (it would then have been expressed as `(sqrt (+ (* x x) (* y y)))`). This ability to enter functions in both languages makes the program easy to learn for the novice, yet powerful for the seasoned user.

4. An "Argument" node is then dragged into cell D1, and its sources are set to be cells B1 and C1. This node collates lists of arguments to generate a new argument list. As an example, if two lists {(1 2) (3 4)} and {(11 12) (13 14)} were passed to an Argument node, the resulting list would be {(1 2 11 12) (3 4 13 14)}. This is needed in order to generate the required arguments for the node in the next step.

5. Another "Arithmetic Progression" node is dragged into the spreadsheet, this time into cell A2. This time, the "step" field is set to 0 (indicating that the same number will always be generated) and the "count" field is set to 100 (indicating that a list containing one hundred "1"s will be generated).

6. A "Ball" node is now dragged into cell B2. Its source is then set to cell A2. This node takes each of the arguments generated by its source cell and uses them as the radius of a new sphere that it creates and adds to its output list. Therefore, the number of spheres generated by this node equals the size of the list that contains the ball radii.

7. A "Translation" node is dragged into cell E1. This node will be used to move the balls from position $(0,0,0)$, where they appear by default, to their proper positions. The balls are taken from the Ball node we just placed in cell B2, while their positions are taken from the Argument node in cell D1.

8. Finally, a "Render" node is dragged into cell E2, and its source is set to the Translate node in cell E1. When the expanded view for this node is opened by clicking on the "..." icon in its top right corner, the cone of spheres can be viewed and manipulated by the user.

5.5 Internal Structure

Internally, ViSSh makes extensive use of the Scheme programming language [26]. Although ViSSh itself was written in C++ [58], the MzScheme embedded interpreter [51] is used as the actual computational engine. C++ was used as the "main" development language since it was deemed easier to combine interpreted Scheme with the OpenInventor [64] data visualisation library than to write OpenInventor bindings for Scheme. MzScheme provides access to Scheme objects via opaque types and helper functions, providing an efficient interface between the computational and display layers of the system.

5.5.1 Layered Design

ViSSh consists of several layers of software, communicating via a standardised interface. The layers are the following:

- Dataflow Manager
- Data Source Layer
- Computation Layer
- Mapping Layer
- Output Layer

The relationship is more completely illustrated in Figure 20. The dataflow manager controls all aspects of the system, propagating dependency change messages backwards and data forwards. Data is inserted into the system by the Data Sources layer, which can either generate data on the fly (e.g., where a portion of an arithmetic series is calculated) or import it (e.g., where a data file is read from disk). The dataflow manager then routes the data either to the Computation or Scene layers, where the data is then prepared for display. Data will typically travel several times from the computation layer to the dataflow manager layer and back, as data flows from one spreadsheet cell to the next. The Mapping layer will at some stage be handed data by the Dataflow Manager to create something the output layer can render. This can be plain text, polygon meshes, etc.

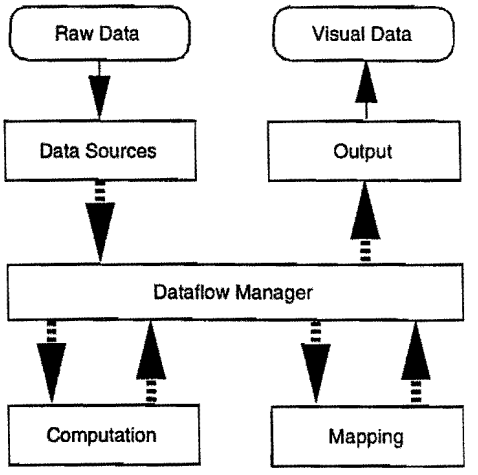


Figure 20: This illustrates the layered design employed by ViSSh. Raw data enters the system via the Data Source layer, and is passed along by the Dataflow Manager to cells in the Computation and Mapping layers. Eventually, it is displayed to the user by the Output layer.

5.5.2 The Dataflow Layer

This is the most complex layer, as it keeps track of all intercell dependencies and manages the flow of data between cells. Data is represented internally in two forms, namely Scheme objects and OpenInventor scene graphs [64]. All data items are stored as Scheme objects, except for the graphical objects manipulated by OpenInventor. Scheme objects are used for general data storage and manipulation since a single opaque type can be used to manipulate any object in the Scheme environment. This means that when data is passed between cells, the dataflow layer need not know whether numbers, strings, lists or any other objects are being passed. This type transparency considerably simplifies the layer. OpenInventor scene graphs are used to transfer geometrical information

generated by the mapping layer to the output layer.

The Mechanics of Data Flow

Although some form of pointer-to-integer fudging could have been performed to pass scene graph pointers as Scheme objects, this was avoided because direct passing of pointers is faster than conversion to and from the internal Scheme representation, and there is no need to anyway since computations cannot be performed on the scene graph pointers.

Instead, two separate channels are used by the dataflow layer to transfer data, one for data on which computations can be performed, and another for scene data. The first channel transfers Scheme objects and the second pointers to OpenInventor scene graphs.

Since the system must deal with potentially enormous lists of items, extensive use is made of lazy evaluation, meaning that calculations are not performed unless it is absolutely necessary (in the hope that they will not be required at all). Unfortunately the Scheme programming language has no provision for lazy evaluation, so this functionality must instead be handled by the dataflow layer. Although all lists that the system deals with look identical to the user, there are actually two distinct types of lists: real lists and lazy lists. A real list is an actual Scheme list, which is stored in its entirety and manipulated as a single Scheme object. Real lists are used when it is known that the lists will be short, typically less than 50 items in length. In contrast, lazy lists are used when the potential length of the list could run into the tens of thousands of items. A lazy list is obtained by repeatedly calling a function, with the list index of the required object as an argument. This works because ViSSh operates on only one list item at a time, and so the entire list is not needed at any one time. This not only greatly reduces storage requirements, but also speeds up recalculations by several orders of magnitude (see Section 4.3.2 of Chapter 4). All data that is moved by the dataflow layer between cells is transferred in the form of lazy lists.

As was mentioned above, scene data is transported separately from Scheme objects, since there is no overlap between the two. However, since potentially thousands of graphical objects can be called into existence, they are all manipulated as lazy lists of OpenInventor scene graphs.

Keeping Track of Dependencies

Intercell dependencies are kept track of by two sets of pointers to spreadsheet cells. The first set keeps track of "depends on" relationships, and the second keeps track of "is depended on by" relationships. When a cell changes value, either because of user interaction, or because a cell that it

depends on changed its value, it signals all the cells that depend on it. Eventually, a cell from the output layer will be notified of a change, and it will then request, one by one, all the elements of the lazy lists that are output by the cells that it depends on. This will trigger a chain reaction of back propagations where the actual values are recalculated and sent back to the output layer cell. Cells that are not providing data to the display cell are not queried at all, and only those values that are required for successful completion of the recalculation are actually recalculated. Note that only the output layer cell has the power to request data, so as long as there are no display cells in the spreadsheet, no calculations are performed. This allows for very responsive editing of the spreadsheet without having to worry about whether recalculation is enabled or not. Most spreadsheet systems have a mechanism for enabling and disabling recalculations, in order to speed up the editing process (when recalculations are disabled, there is no need to wait for them to complete after each cell is edited). However, it would be easy for users to forget that they have disabled recalculations, edit the spreadsheet and read incorrect (i.e., outdated) results from the spreadsheet; for example, to find out whether automatic recalculation is enabled in *Microsoft Excel*, users have to select the "Calculation" tab of a dialog box that opens when the "Options" item of the "Tools" menu is selected — not exactly in plain sight. In ViSSh, however, if there are no display cells, then there is no recalculation; otherwise, there will be recalculation. It is easy to find out if there are any display cells in the spreadsheet — they are all listed in the "View" menu for easy access.

Dependency data is updated whenever cells are deleted or edited. If a given cell is deleted, then this fact is broadcast to all cells in the deleted cell's "is depended on by" list. This causes all these other cells to remove the deleted cells from their respective "depends on" lists. Whenever a cell is edited, the "depends on" list of that cell and "is depended on" lists of the cells that are either newly depended on (or no longer depended on) are updated accordingly. It is during this stage that circular dependencies are checked for, using a simple algorithm that tags cells with a unique value and follows "depends on" pointers until either a cell with no dependencies is found (a data source) or a cell that has already been tagged is discovered (a circular dependence).

The Sheet Call Mechanism

The dataflow layer is also responsible for the inter-sheet function call mechanism (see Section 5.2.2). This is implemented in a client-server fashion, anticipating a network implementation in a future version of ViSSh. When a SubSheet cell is added to a spreadsheet, it scans the subspreadsheet whose name is given to the SubSheet cell, looking for Argument and Result cells and storing pointers to them. When a SubSheet cell is recalculated, it first indicates to the argument cell in the subsheet that

it is the sheet the argument cell must communicate with. The reason behind this is that a subsheet may be simultaneously in use by several SubSheet cells in several spreadsheets. Once this happens, the SubSheet cell then tells the Result cell it must recalculate itself. This will cause a chain reaction of "recalculate" messages to propagate all the way to the Argument cells, followed by a flow of data in the opposite direction. When the Argument cells need data, they request it from their current "server," i.e. the SubSheet cell the Argument last received a "connect" message from. This in turn causes the SubSheet cell to request the same data item from the cell it depends on and pass it on to the Argument cell. When the calculated data has eventually propagated to the Result cell, it is passed on to the SubSheet cell, which then forwards it to the cell that requested it from the SubSheet cell itself. In a sense, the combination of the three cells, SubSheet (on the master sheet), Argument (in the subsheet) and Result (also in the subsheet) form two "data bridges" between the two communicating spreadsheets, one from the master sheet to the subsheet and another in the return direction.

5.5.3 The Data Source Layer

The data source layer is responsible for insertion of data into a spreadsheet. Cells from this layer either generate data, as in the case of the Arithmetic Progression cell, or pull it in from external sources, such as the 1D Dragger cell, which collects interaction data from the user. Cells from the data source layer do not "push" data into the system, rather they signal when their data has changed and wait for it to be "pulled" from them. This "pulling" originates from cells in the output layer, and eventually propagates back to the Data Source layer cells.

5.5.4 The Computation Layer

The computation layer is responsible for all arithmetic and logical transforms to data. All computation is performed via the Scheme programming language [26], using the MZScheme embedded Scheme interpreter [51].

Although the use of an interpreter is not as fast as compiled object code (e.g., loaded in as a shared library), it does have many advantages; the main one is flexibility — it is possible to dynamically extend the program while it is running, without need to call external compilers and linkers. The use of an interpreter also allows for the inclusion of a Scheme interpreter window, which gives users immediate feedback and so can be used to prototype new functions. These functions, which have been defined in the interpreter window, can then be called by function evaluation cells in the

spreadsheet. This ability to create *ad-hoc* functions is an example of the level of flexibility inherent in interpreted systems which would be difficult to duplicate using compiled code. Since the emphasis of a data visualisation system is on exploration and experimentation, we felt that flexibility was more important than runtime performance. Note that this does not mean that the system is sluggish: in our testing we have found that reading 34,000 numbers from an ASCII comma-delimited file, transforming them, making a 3D surface out of them and displaying this surface took 20 seconds on an SGI O2 (R10000 processor running at 195MHz).

All function operands are retrieved from source cells via the Dataflow layer, which also sends the function results to the cells that depend on the computation layer cell.

5.5.5 The Mapping Layer

Cells in this layer work together to map abstract numbers into concrete representations. Currently, this means converting the numbers into geometric data ready for display by cells in the Output layer. This geometric data are represented by OpenInventor [64] `SoNodes`, which are manipulated similarly to data in the Computation layer. Mapping layer cells are grouped into two major categories: scene generators and scene transformers. Scene generators are used to make new objects that can be viewed using a 3D Viewer cell. In this way they are quite similar to cells from the Data Source layer, since data source cells also "generate" new data. Scene transformers, on the other hand, are quite similar to Computation layer cells. They are used to modify a scene, in the same way that Computation layer cells modify data. Examples include the Translate, Rotate and Scale cells. The way these are implemented makes use of the lazy evaluation Scheme discussed in the Dataflow Layer section above.

Consider, for example, translating a sphere (generated by a Ball cell, and implemented as an Inventor `SoSphere` node) by (-10,0,0), as illustrated in Figure 21.

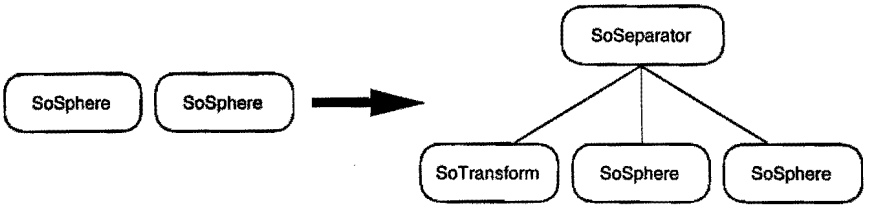


Figure 21: This figure illustrates what happens when a transform is applied to a Mapping Layer cell. Basically, a scene graph is created which contains an `SoTransform` Inventor [64] node, followed by the data that was output by the Scene Layer cell.

The Translation cell, when asked to recalculate itself, first creates a new `SoSeparator` node, then adds to it an `SoTransform` node. This node takes its parameters from the Translation cell. Then it retrieves all the `SoSpheres` generated by its source and adds them to the `SoSeparator` in order. Note that although a (lazy) list of several `SoSpheres` was used as input to the Translate cell, a lazy list containing a single scene graph was generated by the cell. Although in this example `SoSpheres` were added, ViSSh allows for arbitrary scene graphs to be transformed in this way. The key to this is the Group cell, which is very similar in nature to the Translate cell just described, but which simply adds its inputs to a new `SoSeparator` node. The groups generated this way can then be translated, rotated, etc. as a single unit.

5.5.6 The Output Layer

Cells that belong to the output layer are responsible for displaying information to the user, as well as being the data sinks that cause data to be generated from the data source cells (see Section 5.5.1). In many cases these cells also form an environment for user interaction, although the interaction itself is performed by Data Source layer cells. An example of this relationship would be the 1D Dragger cell and the 3D Viewer cell. The latter belongs to the output layer, while the former is a Data Source cell. The 3D Viewer cell displays OpenInventor scene graphs [64], which may include the 3D representation of 1D Dragger cells. When a user interacts with the 3D form of the 1D Dragger, this happens in the 3D Viewer window, yet the actual changes triggered by the user manipulating the 1D Dragger are communicated by the 1D Dragger cell, not the 3D Viewer cell.

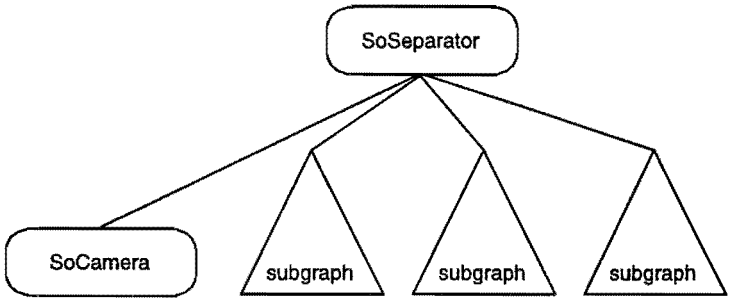


Figure 22: This is the scene graph that is built by Render cells. It is simply an Inventor `SoCamera` [64] node followed by the scene graphs generated by all the inputs to the Render cell.

Output layer cells receive their data in the scene channel of the Dataflow layer (see above). This data, in the form of OpenInventor scene graphs [64], is combined with an `SoSeparator` node and an `SoCamera` node to form a large scene graph, which can then be rendered by a subclass of

`SoXtViewer`. Figure 22 illustrates what the scene graph looks like.

5.6 Chapter Summary

In this chapter we have described an implementation of the extended spreadsheet paradigm described in Chapter 4. This implementation, called ViSSh (short for **Visualisation SpreadSheet**), was built for the purposes of testing the usability of the extended spreadsheet paradigm, as well as creating a tool that could be used by others.

We started by looking at ViSSh from a user's perspective, beginning with a description of how ViSSh implements the extended spreadsheet model described in Chapter 4. We also also described other features of ViSSh that are useful for data visualisation but are not covered by the extended spreadsheet metaphor, such as interaction with the 3D environment and support for animation. At this point we introduced a direct use of the correspondence that exists between spreadsheets and the dataflow paradigm, as described in Chapter 3: the *Cell Dependencies* window, which generates and renders a dataflow diagram corresponding to the current spreadsheet. This is a useful debugging aid, since it makes explicit all the inter-cell dependencies that exist in the spreadsheet.

We then described the three different types of spreadsheet that exist: *Data Sources*, which import data into the spreadsheet (such as the ball generator cell), *Operator Cells*, which perform operations on this data (such as the function evaluator cell), and *Data Sinks*, which output the data in human-readable form (such as the 3D Render cell). This was followed by a simple example use of ViSSh (making a cone out of balls), which illustrated the procedure usually used to build a spreadsheet.

We then discussed the internal structure of ViSSh, beginning with an overview of its layered structure (namely, the *Dataflow, Data Source, Computation, Mapping* and *Output* layers). This was followed by a discussion of how each layer is structured and how it interacts with the layers adjacent to it.

In the next chapter, we shall describe two example visualisations developed with ViSSh.

Chapter 6

Practical Applications

6.1 Introduction

In this chapter, real-world examples will be used to demonstrate and analyse the practical use of ViSSh (and hence, tne extended spreadsheet paradigm).

The process followed is simple: first some actual visualisation problems were obtained that involved non-trivial amounts of data. These problems were obtained from other researchers and come from different areas, namely geology and ATM networks. The data was then visualised by the author, and the visualisation process recorded. That process will be described here, together with the reasons why certain decisions were made during the visualisation process.

The results described in this chapter will be used in Chapter 7 to analyse both the ViSSh prototype and the extended spreadsheet paradigm, using Green's Cognitive Dimensions Framework [18, 17].

Note that to enhance readability, all spreadsheet cell names in this chapter are typeset in a sans-serif font.

6.2 Example 1: Seismic Disturbance Analysis

The data used for this visualisation was collected by Dr. David James of the Carnegie Institute of Washington, in the United States. It comes from the KaapVaal Seismic Experiment [13], in which teleseismic events were recorded from Southern Africa in a SW-NE diagonal roughly 3300Km in length (see Figure 23). The data represent disturbances in the earth's crust generated by seismic waves that propagated through the mantle after being generated by earthquakes in other parts of

the world, such as China or South America. This data is useful because it can be used in a way analogous to medical ultrasound to obtain imaging of the earth. The actual measurements are of velocity perturbations, measured in Km/s.

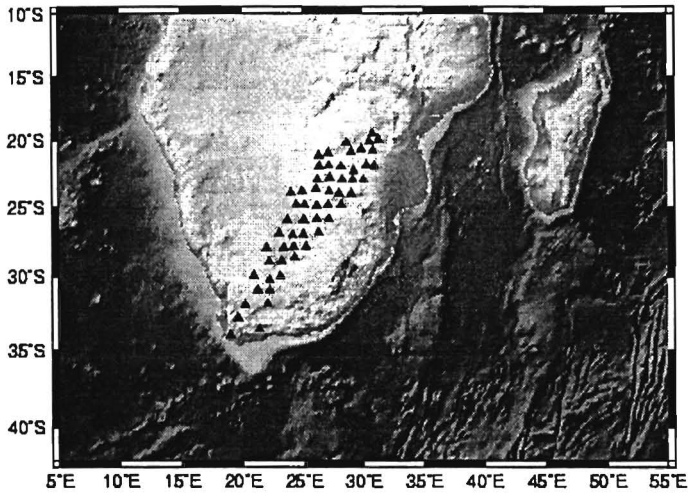


Figure 23: Sampling Stations for the KaapVaal Experiment. These stations lie on a diagonal roughly 3300Km in length, and are used to measure teleseismic events, or "echoes" of earthquakes that occur on other parts of the world.

This particular test aims to demonstrate several basic properties of ViSSh:

- The ability to import and handle large datasets.
- The ability to perform arbitrary transforms on a dataset to prepare it for visualisation.
- The ability to generate meaningful pictorial representations of the dataset.
- The usefulness of the *Cell Dependencies* window.
- The compactness of the spreadsheet visualisation programs.

6.2.1 The Dataset

For this visualisation, the entire dataset (76141 data points) was used. This is organised as a set of rectangular "slices," which measure 25 by 24 degrees (i.e., 1500×1440 nautical miles), and were taken at 50Km depth intervals through the earth's crust. Each slice consists of 47×45 data points, or 2115 points. Each data point consists of a tuple $\{x,y,z,p\}$, where x and y are the longitude and

latitude of the point, measured in degrees, z is the depth below sea-level measured in Km, and p is the actual measured value. This value is measured as the difference from average velocity of the traveling seismic waves, in Km/s. All points which share the same z value belong to the same "slice."

Data Format

The data used for this visualisation was stored in text format, using the popular "comma-delimited" format. The dataset consists of a number of newline-terminated lines, each of which being made up of four decimal numbers separated by commas. A small section of the data is shown in Figure 24; this represents part of a slice that lies 250Km below the surface. Where no data was available, this fact was represented by a p value of 999, as illustrated by the final column of the first row of Figure 24. Although this format is not very space-efficient, it does have the definite advantages that it is very portable, easy to parse and output by most commercial spreadsheet and database programs.

```
-24.000000, 21.500000, 250.000000, 999.000000
-24.000000, 22.000000, 250.000000, 0.021240
-24.000000, 22.500000, 250.000000, 0.025710
-24.000000, 23.000000, 250.000000, 0.023950
-24.000000, 23.500000, 250.000000, 0.008470
-24.000000, 24.000000, 250.000000, -0.007190
-24.000000, 24.500000, 250.000000, -0.012570
-24.000000, 25.000000, 250.000000, -0.015080
```

Figure 24: This illustrates the data format used to store the seismic disturbance data to be visualised. Each line is a field, and the records are separated by commas. Where no data was recorded for a given point, the last field contains a 999 (e.g., the first line of this sample).

6.2.2 Objective of the Visualisation

Before any programming task can take place, it is important to have a clear idea of what must be done. In this instance, the task is to build a simple browser, which may be used to visualise any given "slice" of the data as a height field, with the x and y coordinates being taken as-is from the dataset and the z coordinate being a scaled-up version of the p value from the dataset. Additionally, the displayed 3D surface will be coloured according to the p value at each point. This will allow for more accurate analysis of slices, since overhead views (along the z axis) can be used to find patterns

in the seismic disturbances. Using this browser, the user will be able to see where the disturbances are and how these disturbances change with increasing depth (by changing the slice parameter and watching how the rendered slice changes).

6.2.3 Logical Structure

Figure 25 illustrates the intercell dependencies for the seismic disturbance spreadsheet, as generated by the *Show Dependencies* function of ViSSh (this can be seen as a dataflow diagram representation of the spreadsheet). The actual spreadsheet can be seen in Figure 26.

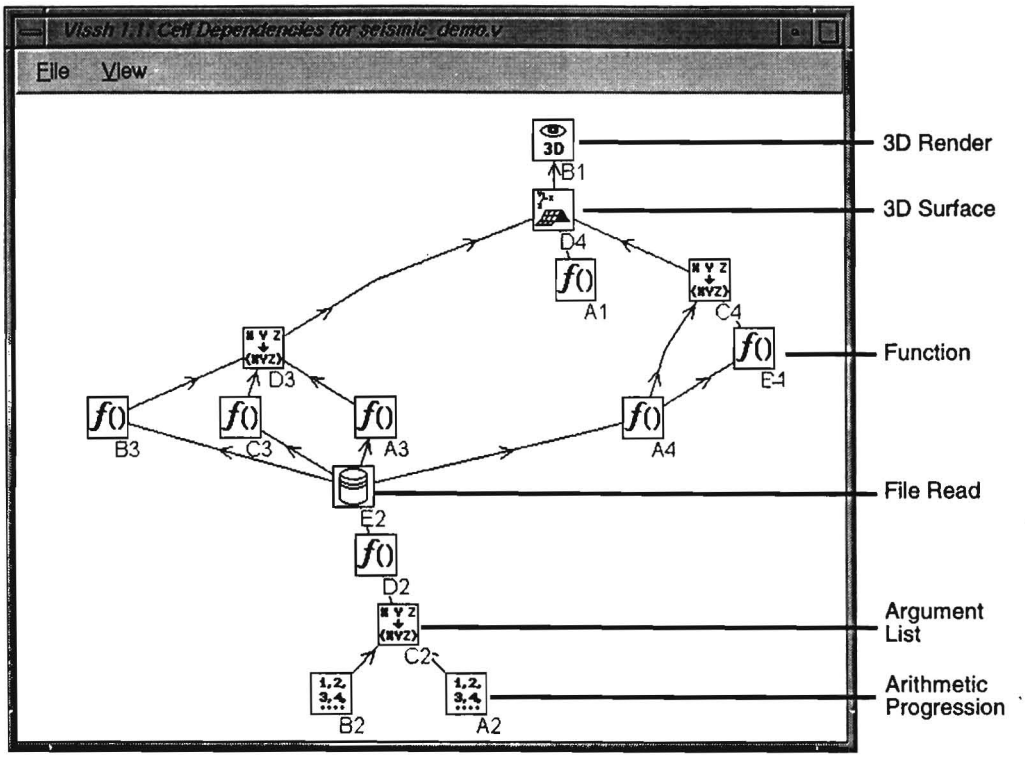


Figure 25: These are the dependencies of all the cells of the Seismic Disturbance spreadsheet. This diagram was generated by the "Show Dependencies" feature of ViSSh.

The *Show Dependencies* view lets the user quickly discover any flaws in the spreadsheet, as well as gaining an understanding of the (sometimes rather complex) dependencies that can develop in spreadsheets. Consider, for example, the path taken by the data retrieved from the database. This originates in cell E2, and it is plainly visible that it then goes through B3, C3 and A3 before

being recombined into tuples by cell D3. These tuples are then fed into cell D4 and form part of the surface-generation process. This flow of data is not clearly visible by simple examination of the spreadsheet (i.e., some effort must be expended into searching the spreadsheet for which cells depend on which) and therefore the dataflow representation can be clearly seen to make up for that particular deficiency in the spreadsheet paradigm.

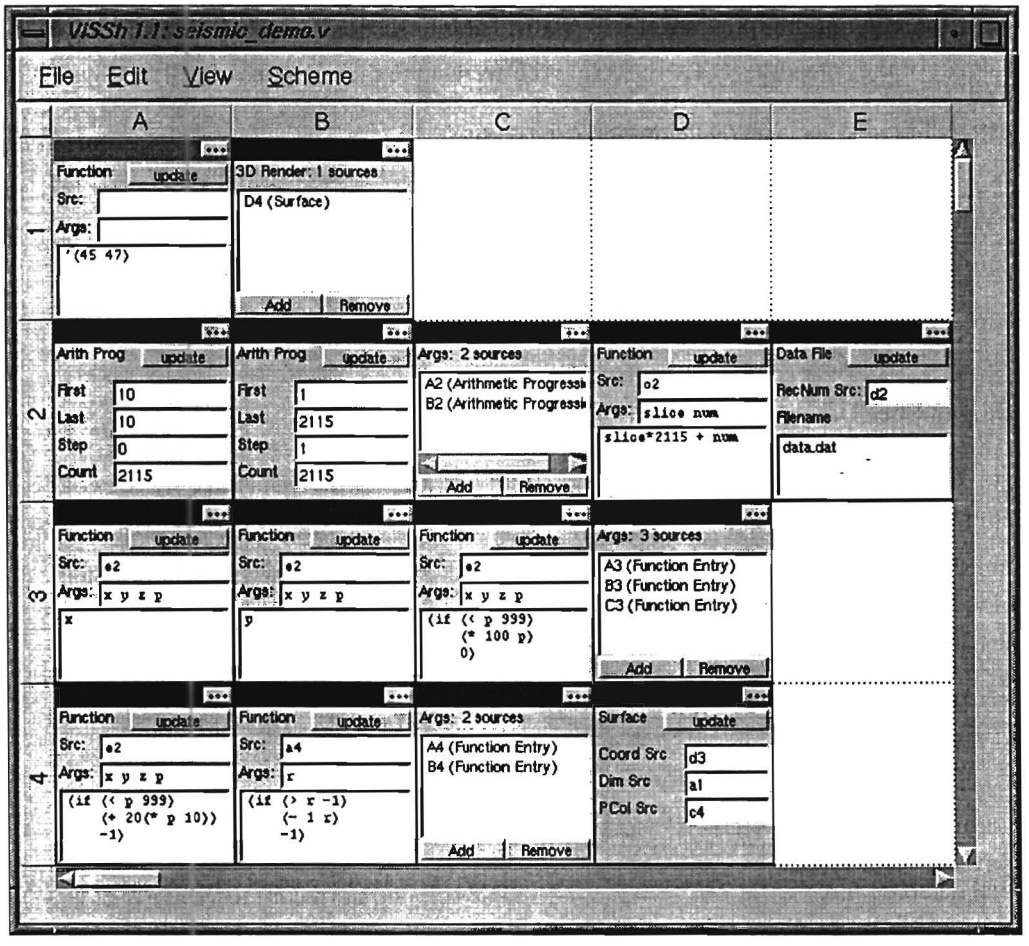


Figure 26: This is the actual spreadsheet used to visualise the seismic disturbance data.

Before beginning the analysis of the seismic disturbance spreadsheet, a few things mentioned in Chapter 5 are worth repeating:

- The spreadsheet is made up of cells that are arranged in a rectangular grid and referenced in

the same way as those in "traditional" spreadsheets, i.e., the first cell in the top-left is called A1, the cell to its right is called B1, the one below it A2, and so on.

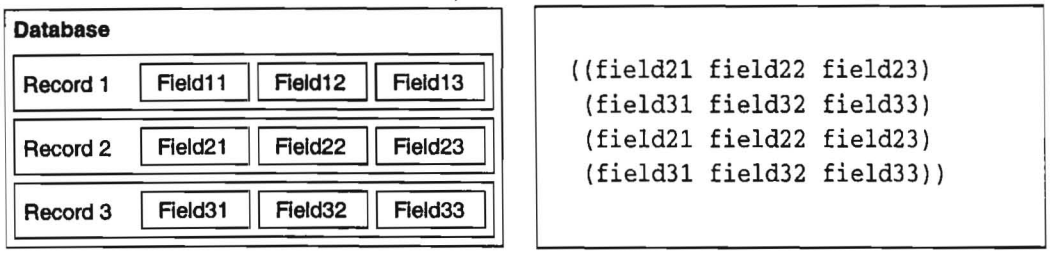
- The cells look nothing like traditional spreadsheet cells, containing miniature control panels instead of text or numbers.
- Each cell contains a list of values, not a single number or string.

The discussion of the spreadsheet will be divided into sections, each of which corresponds to a logical part of the spreadsheet. These sections correspond to these groups of cells in Figure 25:

1. Reading in Slice Data (cells A2, B2, C2, D2, E2)
2. Transforming the Input (cells A3, B3, C3, D3)
3. Building the 3D Surface (cells A4, B4, C4, A1, D4)
4. Rendering (cell B1)

Reading in Slice Data

The job of actually reading in the data points from the database file is handled by cells A2, B2, C2, D2 and E2. Although this may seem like quite a complex operation, in essence what is being done is quite simple: a list of record numbers is built up, and then used to retrieve data from a file.



Actual database file. File cell output (record list was `(2 3 2 3)`).

Figure 27: This illustrates how a File cell interprets a database file. The database's contents are shown on the left, while the list generated by the File cell is shown on the right. Note that the generated list depends on the given list of record numbers.

Figure 27 illustrates how a database file is interpreted by a File cell; records are selected from the database based on a list of record numbers, and returned as a list of records. The record numbers used to retrieve data items from the database are calculated using the fact that the data are structured as "slices" of 45×47 data points each. These cells operate as follows:

- **Cell A2** indicates the slice number that we wish to examine (settable by the user); in the spreadsheet in Figure 26 the slice being examined is the eleventh one (slices are numbered starting with 0). The reason behind the fact that the "count" field of cell A2 contains "2115" instead of "1" is related to the way ViSSh handles list operations. When an operation is performed on several lists, the resulting list is always defined to be as long as the shortest of all the input lists; should the "count" field of cell A2 have been "1", then only one data point would have been read in, instead of 2115.

- **Cell B2** generates record offsets used to retrieve data from the disk file. It is an arithmetic progression cell, configured to generate a list containing all integers between 1 and 2115 (as mentioned above, each slice consists of 47×45, or 2115 data points).

- **Cell C2** is used to prepare function arguments for the function evaluator cell D2. It interleaves the lists generated by cells A2 (the slice number) and B2 (the point number within the slice), generating a list of lists which reads `((10 1)(10 2)...(10 2115))`.

- **Cell D2** calculates the record number to read off the database file. It uses the list containing the slice and record numbers passed by cell C2 to calculate the record number of each data point in the data file. The formula is quite simple, $f(slice,offset) = \{slice \times 2115 + offset\}$.

- **Cell E2** is what actually reads data off the database. It treats the input file (in this case, "data.dat") as a database made up of comma-delimited records, which are in turn separated by newline characters. This format is quite popular for exporting data, and as such most database engines can be configured to output their data in this way. At this moment in time, this is the only data format that can be imported by ViSSh. Each newline-terminated line of the data file is a record, and the numbers of the records that must be fetched from the data file are given to Datafile cell E2, in this case by cell D2. The Datafile cell outputs each record as a list, the items of which being the fields. As an example, if the last two lines of the data in Figure 24 were read in by a Datafile cell, its output would be the following list: `((-24 24.5 250 -0.01257)(-24 25 250 -0.01508))`.

A look at Figure 25 will clarify the role of these four cells: two number sequences, generated by cells A2 and B2, flow into cell C2, which combines them into a single list. This list is then forwarded to cell D2, which applies a function to them and gives the result of that function (a record number) to cell E2, which queries the database.

Transforming the Input

Now that cell E2 can read in the data points from the database file, we need to display them. Before this can happen, the data need to be transformed slightly to aid comprehension by the user. In this case, we choose to display the seismic disturbances as a three-dimensional plot, with the x coordinate being longitude, the y coordinate being latitude, and the z coordinate being the actual disturbance value. Although we can use the x and y values directly from the data file, the z value presents some problems. Firstly, there are many "missing" data points, represented by a disturbance value of 999 (see Figure 24). Also, the seismic disturbances are about four orders of magnitude smaller than the longitude and latitude.

The cells in range A3:D3 take the $\{x,y,z,p\}$ tuple generated by cell E2 and convert it into an $\{x,y,z\}$ tuple suitable for rendering in three dimensions. Cells A3 and B3 simply extract the x and y components of the $\{x,y,z,p\}$ tuple by means of a simple function, in A3's case $f(x,y,z,p) = \{x\}$, B3 performs a similar task for the y coordinate.

The actual solving of the problems mentioned in the first paragraph of this section (namely, missing values and the discrepancy in order of magnitude) is handled by cell C3. This cell makes use of the fact, as mentioned in Chapter 5, that the underlying computational engine used by ViSSh is the Scheme programming language [26]. Although, as we saw in the case of cell D2, functions may be entered in the "standard" mathematical infix format, they may also be entered in the native Scheme, in order to make fuller use of the language's expressive power. Since the scheme function entered in the cell is not clearly visible in Figure 26, that function is reproduced in Figure 28, together with a translation of it into C [27], for easier comprehension.

```
(if (< p 999)
    (* 100 p)
    0
)
```

Original Scheme Expression

```
if (p < 999)
  return (100 * p);
else
  return 0;
```

Expression translated to C

Figure 28: This function is used to scale the seismic disturbance reading in order to make it noticeable, as well as to account for the fact that "missing" values are represented as 999. The function, as it appears in the spreadsheet, is in Scheme, but it is illustrated here next to its C [27] translation to ease comprehension.

This example serves well to illustrate the advantage of using a functional programming language like Scheme instead of the more primitive formula languages used in traditional spreadsheet packages. Its function is quite simple: it takes each 4-item list generated by cell E2, i.e., the $\{x,y,z,p\}$

tuple read from the database, and calculates the z value to be plotted. This is simply the p database field multiplied by 100, unless the stored value was 999, in which case it is forced to 0 (another cell will be tasked with making these "missing" values invisible).

Referring to Figure 25 will clarify the above paragraph. The data read off the database by cell E2 is sent to cells A3 and B3, which extract, respectively, the x and y values from the record read in. It is also sent to Cell C3, which extracts the p value (see Section 6.2.1) and applies a function to it. These three values are combined into one tuple by cell D3, which outputs a 3D point.

Building the 3D Surface

Cells A4, B4, C4 and D4 build the 3D surface to be rendered. Cells A4, B4 and C4 calculate the colour to be used at each point (these colours will be interpolated by the rendering engine), and cell D4 is responsible for the actual generation of the 3D mesh.

Cell A4 calculates the red component of the colour of the mesh at any given point. It works similarly to cell C3, as illustrated in Figure 29, which is again translated into C for the convenience of the reader. It basically spreads the p argument passed to it in the range $\{-10, 10\}$, unless it is 999, in which case an illegal value of -1 is returned. This illegal value is what forces the "missing" data points to be invisible. Cell B4 does something similar to calculate the green component of the colour, which is simply one minus the red component (unless the red component is -1, in which case the green component will also be -1).

```
(if (< p 999)
    (+ 20 (* p 10))
    -1
)
```

Original Scheme Expression

```
if (p < 999)
  return 20 + p * 10;
else
  return -1;
```

Expression translated to C

Figure 29: This function calculates the Green component of the colour of the 3D surface at the given point, by mapping the value into the range [-10,10]. The function, as it appears in the spreadsheet, is in Scheme, but it is illustrated here next to its C [27] translation to ease comprehension.

Cell C4 combines the red and green components, which are then passed to cell D4 (the missing blue component will be filled in by cell D4, the 3D surface generator).

Cell A1 expresses the dimensions of a single slice. It is a function entry cell, which is configured to always return the same value: a two-element list containing the dimensions of a slice: `(45 47)`. This value is also passed to cell D4, which uses it to convert the 1D list generated by cell E3 (which

we described above as the cell that generates the list of points used to build the surface) into a 2D surface.

Referring to Figure 25 further illustrates the operation of building the 3D surface. Cell A4 retrieves the database record in the same way as cells A3, B3 and C3, extracts the p attribute (see Section 6.2.1) and uses it to generate the red component of the vertex colour for that data point. This is then passed to cell B4, which generates the green component. These are then combined to form a colour triplet by cell C4 (the missing blue component defaults to 0). These colours are then passed, together with the 3D data generated by cell D3 and the slice dimensions in cell A1, to cell D4, which generates the actual 3D surface.

Rendering

Rendering of the generated surface is handled by cell E1. This cell reads the 3D mesh generated by cell D4, as illustrated in Figure 25, and renders it to a separate window. This window is independent of the spreadsheet editing window and can be resized and moved to other parts of the screen. The actual geometry can be manipulated by the user, so it can be rotated, zoomed, etc. Figure 30 illustrates this window for slice 10. If the user wishes to view any other slices, they need only change the number in cell A2, and the view in the 3D window will be automatically updated. Clicking on the "Disk" icon on the top left corner of the 3D render window will output the current scene (together with the current view parameters) to an Inventor format 3D file [64]. This format is understood by many 3D packages, and can be easily converted to the popular VRML format.

6.2.4 Conclusion

This first test was designed to demonstrate the following properties of ViSSh:

A. The ability to import and handle large datasets.

B. The ability to perform arbitrary transforms on a dataset to prepare it for visualisation.

C. The ability to generate meaningful pictorial representations of the dataset.

D. The usefulness of the *Cell Dependencies* window.

E. The compactness of the spreadsheet visualisation programs.

We believe we have demonstrated all of the above; our justification for this statement follows.

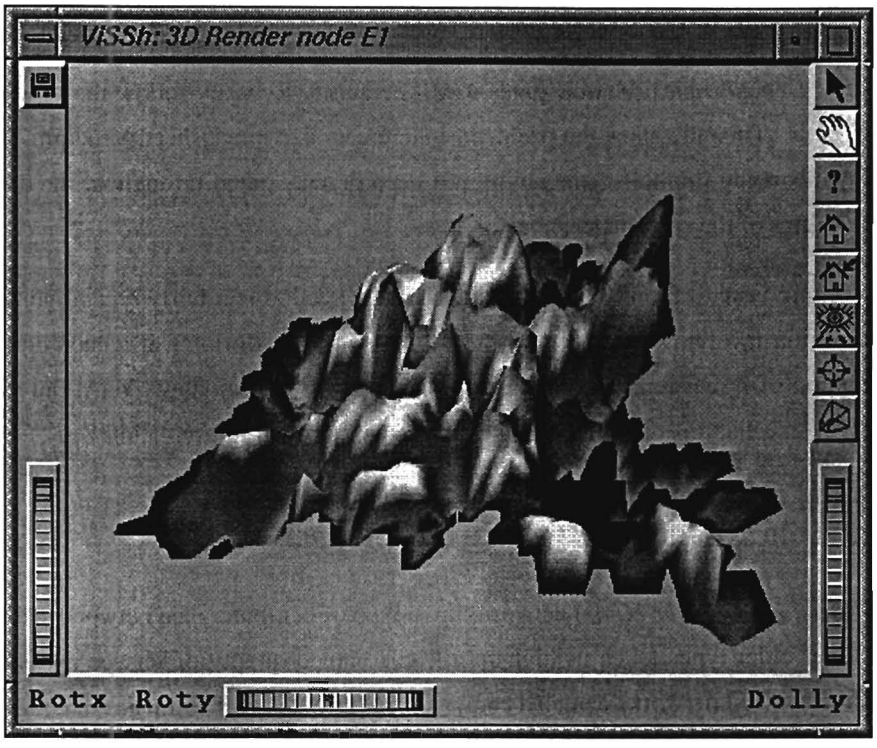


Figure 30: This is a rendering of slice 10 of the seismic disturbance data, visualised by plotting the disturbance value both as a vertical displacement and a vertex colour. Clicking on the "Disk" icon on the top left of the 3D render window will save the 3D scene, including the current view parameters, to a file.

A. ViSSh's database model, in which a database is conceptualised as a set of numbered records implemented as lists, allows databases of arbitrary size to be accessed, since only one record is accessed at a time. Although the data format is currently limited to comma-delimited ASCII, adding specialised import modules should pose no significant challenge.

B. As shown in Section 6.2.3, ViSSh's function entry cells, optionally coupled with the use of the Scheme programming language (common arithmetic expressions can be stated in infix form), gives ViSSh spreadsheets a great deal of expressive power.

C. ViSSh was used to generate a 3D surface from the given data, where the x and y coordinates of the 3D surface mapped 1 to 1 to those of the dataset, and the z coordinate of the 3D surface was a non-degenerate linear function of the p attribute of the dataset (see Section 6.2.1). This means that, since each distinct dataset will have a distinct pictorial representation that is linearly dependent on the dataset and nothing else, users should be able to extract meaning

from the data by only looking at the generated image.

D. The *Cell Dependencies* window gives users the ability to easily follow the flow of data between cells. This alleviates the problem, inherent in the spreadsheet paradigm, of links between cells being implicit. This ability has been demonstrated throughout the discussion of the workings of the spreadsheet above.

E. The seismic visualisation example outlined in this section is a fairly typical non-trivial data visualisation involving reading data from a text file, transforming it, generating a 3D representation of it and rendering this onto an interactive 3D window. Yet it only fills fifteen spreadsheet cells; this demonstrates the conciseness of the representation.

6.3 Example 2: Network Routing Visualisation

ATM (*Asynchronous Transfer Mode*) networks are a type of computer data network that has several advantages over older types (such as high speed, guaranteed quality of service and simplicity of implementation). ATM networks can also be dynamically routed, which means that the exact route that a given data packet takes to get from point A to point B can be determined "on the fly." This allows ATM networks to quickly adapt to changes such as surges in network traffic or damage to routers. This dynamic routing is a complex operation, and there exist many algorithms to manage it. To identify which algorithms are most appropriate for a given network, or what parameters need to be fine-tuned, the network must be visualised in such a way that the effects of these changes can be quickly and easily appraised.

The motivation behind this visualisation exercise was to demonstrate ViSSh's ability to interact with the user at the 3D representation level (which provides users with the ability to steer computations at a more concrete level than spreadsheet formulas), as well as the usability of the *Broad Overview* window (see Section 5.2.4 for more details).

6.3.1 Task Outline

As was mentioned above, the fine-tuning of an ATM network can be a complex operation involving a number of parameters. This visualisation aids the professional involved in fine-tuning by plotting the relationship between *physical links* (the actual network nodes), *logical routes* (the path taken by data packets when moving between physical links) and *capacity* (the amount of data that can travel down a particular logical route).

The visualisation was performed by doing a 3D rendering of several stacked 2D plots (each containing a *logical-route vs. capacity* plot). These plots are aligned so that network nodes can be compared by having a fixed *physical-link* and variable *logical-route*, or vice versa.

6.3.2 Description of the Visualisation

Each of the *logical-route vs. capacity* plots is located at the required point along the *physical-link* axis. Each data point was represented by a small cube in the 3D subspace spanned by the three axes *physical-link*, *logical-route* and *capacity* (See Figure 31). To enhance visibility, a line was plotted between each cube and the horizontal (*physical-link/logical-route*) plane. This way the *physical-link vs. logical-route* relationship between different data points could be determined without having to rotate the view to an overhead position.

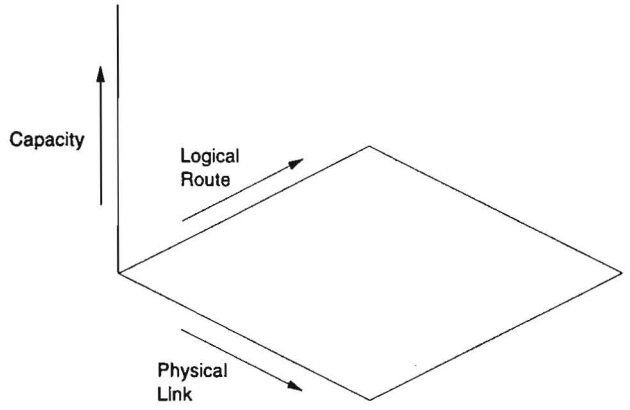


Figure 31: These are the axes used to represent the three different dimensions of the network routing data. "Physical Link" refers to the actual network node (e.g., 5), while "Logical Route" is the route that a data packet will travel to get from origin to destination (e.g., 5-6-7-8). "Capacity" is the maximum number of packets that can travel through a given physical link that is part of a given logical route.

Clicking on any of the cubes representing a data point displays, in a spreadsheet cell, the {*physical-link logical-route capacity*} tuple corresponding to that particular point. In this way a broad overview of the data can be obtained by simple inspection, while more detailed analysis can be obtained by clicking on the relevant points.

6.3.3 The Spreadsheet

Figure 32 illustrates the broad overview of the spreadsheet (see Section 5.2.4 for an explanation of this feature of the ViSSh program).

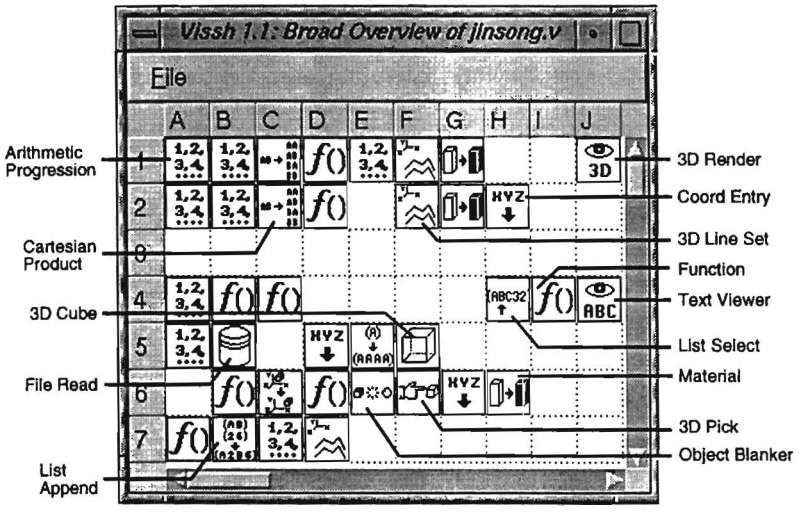


Figure 32: Overview of the Network Routing Visualisation Spreadsheet, as generated by ViSSh's *Broad Overview* function (see Section 5.2.4.)

The spreadsheet cells are grouped into several logical clusters, each performing one of the following tasks:

- Generating the 3D geometry of the *xy* (*physical-link/logical-route*) plane (cells A1:H2).
- Reading the data to be visualised off the disk file (cells A4:C4 and A5:B5).
- Generating the 3D geometry of the data cubes (cells D5:F5, B6:C6 and D6:E6).
- Generating the 3D geometry of the lines projecting below the data cubes (cells A7:D7).
- Outputting all the 3D geometry to an interactive 3D Render window (cell J1)

Generating the x-y Plane

This is taken care of by the cells in the range A1:H2 (see Figure 33). These cells are grouped into four groups, namely:

1. Generating coordinates for the horizontal lines (cells A1:D1).
2. Generating geometry for the horizontal lines (cells E1,F1,G1 and H2).
3. Generating coordinates for the vertical lines (cells A2:D2).
4. Generating geometry for the vertical lines (cells E1,F2,G2 and H2).

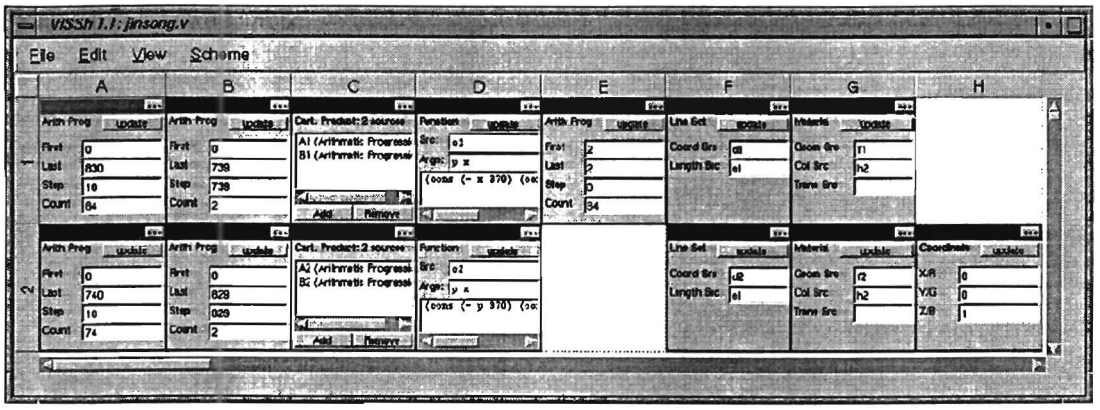


Figure 33: Closeup of cell range A1:H2 — these cells generate the grid which lies under the data points.

There is a degree of symmetry between these four groups, reflecting the similarity in their functions. The cells which calculate coordinates (Groups 1 and 3 above) do so in similar fashions: first, the untranslated end coordinates of the lines are generated by arithmetic progression cells. These are made into (x,y) pairs by Cartesian product cells, which are then properly translated and made into 3D coordinates by function entry cells. For the horizontal lines of the grid, this is done by cell D1, while cell D2 takes care of the vertical lines.

The coordinates generated by cells D1 and D2 (described above) are now used to generate the actual 3D lines making up the representation of the $x-y$ plane; this is done by the cells lying in the range E1:H2, which form groups 2 and 4 mentioned above. The actual 3D lines are created by cells F1, G1 (horizontal) and F2, G2 (vertical). These cells generate arbitrary line sets, and must be given more information before they can interpret the list of coordinates given to them, namely the number of coordinates each line segment is made up of. This information is generated by cell E1, which generates a list containing eighty-four 2's, indicating that the line sets being generated consist of eighty-four segments spanning two points each.

In order to make the $x-y$ plane less obtrusive, it was decided to make it blue. This is done by cells G1 and G2, which apply the RGB triplet $\{0,0,1\}$ (i.e., blue) to the horizontal and vertical lines, respectively. The RGB triplet was generated by cell H2.

Reading the Data Off the Disk File

This operation is performed by two groups of cells (See Figure 34 for a close-up of the actual cells): cells A4:C4 are responsible for generating the first and last record indices to read off the data file, and A5:B5, which respectively generate all the consecutive record numbers corresponding to

the records that must be read in, and read in the actual data records (see Section 6.2.3 for a more complex example of database access). The data itself is stored in a comma-delimited text file. This format was chosen since most database/spreadsheet programs can export data in this format. Note that the "First" and "Last" indices of cell A5 are cell names enclosed in square brackets: this means that the actual values are extracted off those cells, instead of being supplied by the user when the spreadsheet is edited.

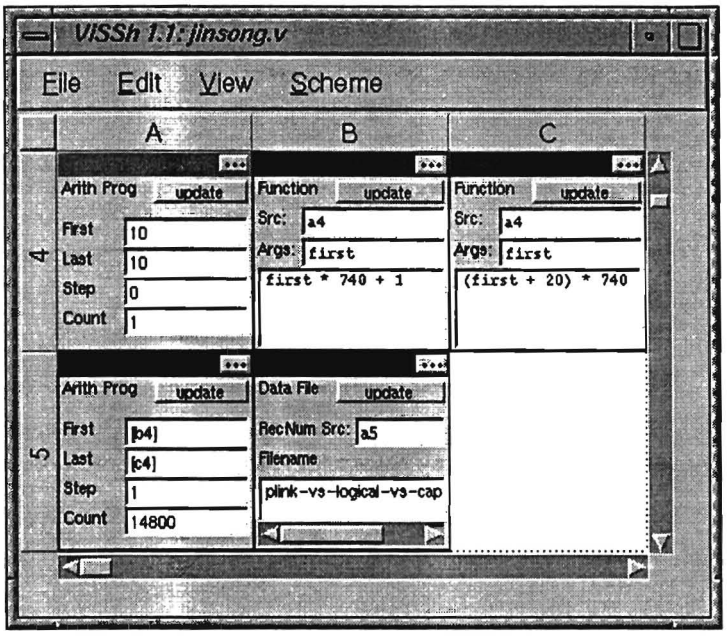


Figure 34: Closeup of cell range A4:C5 — these cells calculate which records must be read from the data file, and read these records in.

Cell A4 is special since it specifies the first of the twenty physical links to be plotted, and hence forms part of the "user interface" of the spreadsheet.

Generating the 3D geometry of the data cubes

An important part of the generated scene is the set of cubes that represent the data points. Each of these cubes has a vertical line projecting underneath it, which extends until it meets the xy plane. This line is used to compare the position of cubes when the scene is viewed at an angle. Additionally, these cubes can be clicked on to obtain specific information about the corresponding data point. These factors complicate the generation of the scene displayed in the 3D view.

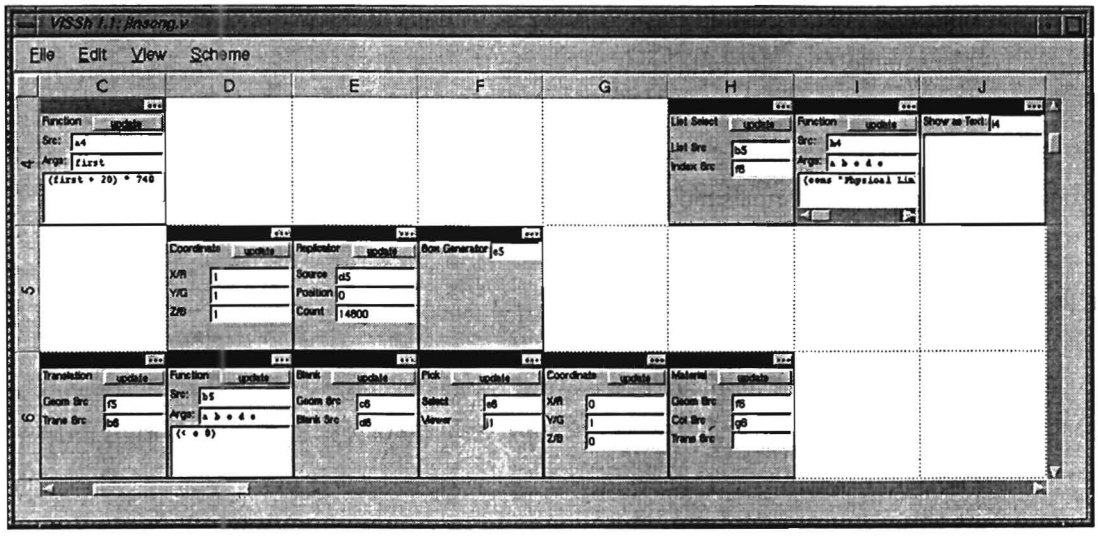


Figure 35: Closeup of cell range C4:J6 — these cells generate and transform the 3D geometry used to visualise the network, as well as managing interactive 3D queries.

Generating the actual cubes is the responsibility of 3 groups of cells, namely the ones spanned by ranges D5:F5, B6:C6 and D6:E6 (see Figure 35).

- Firstly, cells D5:F5 generate a list of 14800 cubes of size $1 \times 1 \times 1$.
- These cubes are then translated to their proper positions by cells B6:C6.
- The data in the source file is sparse, that is, not every possible data point contains data. These "data holes" are represented in the dataset by data points having a "capacity" value of -999. Cells D6:E6 take care of this, by blanking out any cube that has a negative capacity.

As mentioned above, the generated 3D scene is interactive, that is, the user may select cubes with the mouse in order to get detailed information on the network nodes represented by the cubes. This functionality is implemented by *Pick cells*. 3D objects that are eligible for picking are passed through a Pick cell, which updates itself automatically when one of these objects is selected (the 3D render window where the interaction must happen is also specified — this allows the spreadsheet developer to display the same 3D scene in two different windows, one with interaction and one without). The value generated by the cell is a list index, corresponding to the position of the selected 3D object in the list that passed through the Pick cell. Since ViSSh's computational model is such that a list of n 3D objects can only be created by a list of n data values, retrieving information about

the i^{th} 3D object in a list is as simple as retrieving the i^{th} item of the list used to generate that 3D object. As an example, consider cells F6 and H4:J4 in Figure 35. Cell F6 reads in the 3D geometry generated by cell E6 (and passes it on to cell H6), and when the user clicks on one of the cubes, the list offset of the cube is sent to cell H4, which uses it to retrieve the data used to generate the cube from cell B5. Cell I4 then formats the data into a message, which is displayed by cell J4.

Generating the 3D geometry of the lines projecting below the data cubes

As mentioned above, it can be difficult to see exactly where over the xy plane a particular cube is, especially when the scene is not viewed from the top (i.e., along the z axis). To remedy this problem, a series of vertical lines are drawn projecting below each cube, stopping at the xy plane. These lines allow the user to more easily compare several network nodes that share either the same physical link (x coordinate) or logical route (y coordinate). Cells A7, B7, C7 and D7 implement these lines (see Figure 36). The coordinates used for the base of each line (where it meets the $x-y$ plane) are generated by cell A7. These coordinates are then paired with the location of the corresponding 3D cube (taken from cell B6) by cell B7. The result of this is a set of coordinate pairs, with the first one being the top of the line (where it meets the corresponding cube) and the second being the bottom of the line. The actual line segments are generated by cell D7, in a similar way to the line segments used to make up the $x-y$ plane.

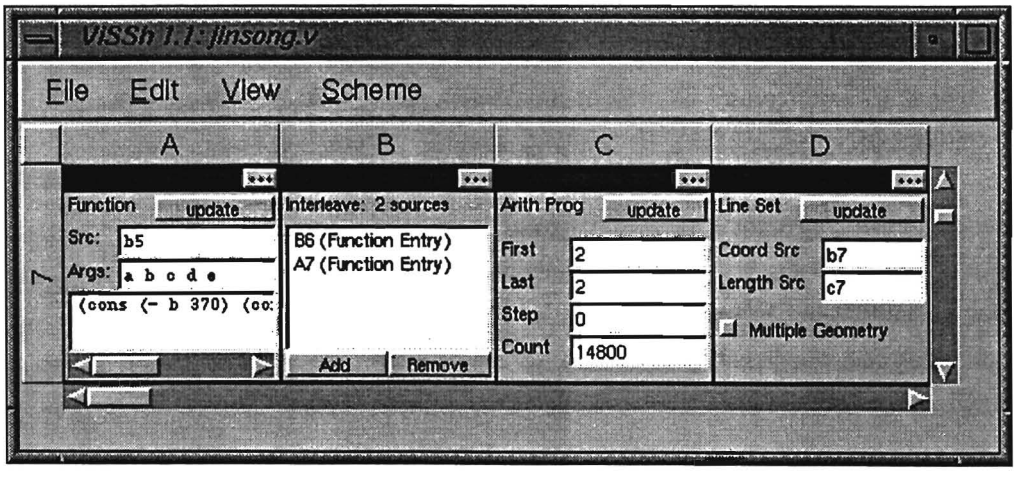


Figure 36: Closeup of cell range A7:D7 — these cells generate and transform the 3D geometry used by the vertical lines under each data cube.

Outputting all the 3D geometry to an interactive 3D Render window

Once all the 3D geometry has been generated, it is displayed by cell J1. Figure 37 illustrates what a typical generated scene looks like, together with the enlarged view of the text display cell J4. This displays the information relevant to the selected data point.

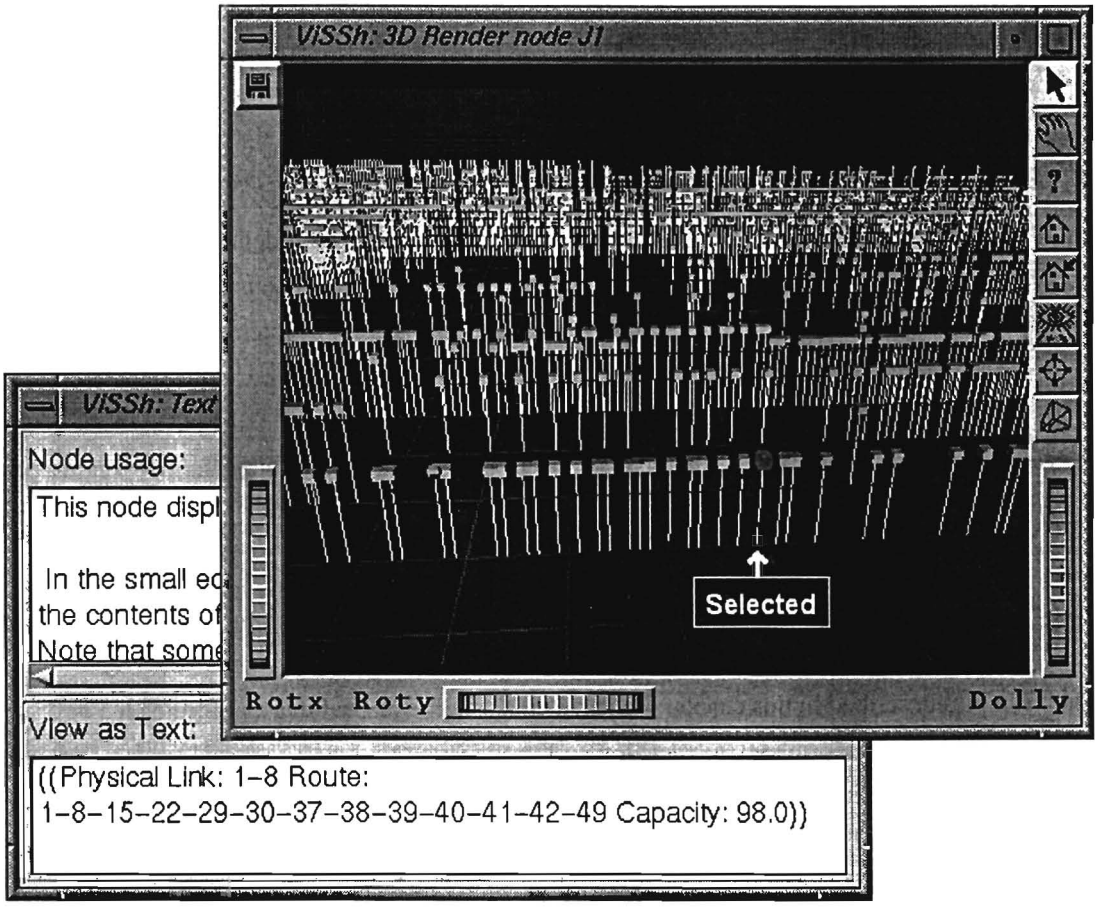


Figure 37: This is an example image generated by the Network Routing Visualisation Spreadsheet. Note that one of the data points has been clicked on, and is highlighted in red (the "Selected" caption is not part of the screenshot). The text display window (opened by clicking on the "..." button of cell J4, see Figure 12 in Chapter 5) shows the data associated with the selected data point.

6.3.4 Conclusion

This second test was designed to illustrate the usefulness of the *Broad Overview* window, as well as ViSSh's 3D interaction capabilities.

Spreadsheets are normally built as sets of functional cell clusters, as we have demonstrated in our discussions of both the seismic and the ATM network spreadsheets. The *Broad Overview* window lets the user find these clusters in an easier way than is the case with the main editing window, since more cells are visible in the former. Since cells are displayed in the *Broad Overview* window in iconic form, users can quickly spot the cell cluster they are looking for; clicking on one of the cells in that cluster will scroll the main editing window so that the cluster can be edited. The fact that the "Broad Overview" window scrolls independently of the main editing window lets users compare the area being edited with the one being scanned.

We have shown the general operational model behind 3D interaction, as well as a specific example. Generally, 3D interaction works by causing a cell to update its "changed" status when the user interacts with the 3D display. This makes 3D interaction semantically identical to editing values stored in spreadsheet cells. The method used for extracting information about an object by selecting it in the 3D display is quite simple. In order to implement this form of interactivity, Pick cells are used. Lists of 3D objects to be rendered are passed through Pick cells before being rendered. When a user clicks on a 3D object in the render window, the pick cell outputs the list index of the object that was clicked. This index can then be used by other cells to obtain the data that was used to generate that particular 3D object.

6.4 User Experiences

The examples described in this chapter are part of the set of visualisations that formed the basis for the user testing we performed while developing the ViSSh prototype. The actual testing was done by the author and others, mainly with the objective of iteratively improving the design of the system as it was being built. User feedback was also used to improve the user manual (see Appendix A).

We have compared ViSSh to Khoral Research's *Cantata* [28], and found that ViSSh was generally easier to use. We believe this is mainly because the spreadsheet layout made the visual program less cluttered than was the case with Cantata's dataflow model, thus easing the editing process. Additionally, the Scheme programming language made the formulas, especially those involving conditionals, compact and concise. Using the "Broad overview" and Dependencies windows (see Section 5.2.4 for details) also eased the visualisation task. The latter in particular gives users the advantages normally associated with dataflow diagrams without the clutter problems mentioned above (since the dataflow diagram is generated algorithmically).

We have also analysed ViSSh using Green's Cognitive Dimensions Framework [18, 17] (see

Chapter 7) and found that it compares favourably with dataflow systems.

6.5 Chapter Summary

In this chapter we have shown the real-world applicability of ViSSh, the data visualisation spreadsheet that was described in Chapter 5, as well as demonstrating the usefulness of its major features. Two very different data sets were obtained (seismic disturbances and ATM network topology) and visualised. The spreadsheets used to visualise both sets of data were described in detail, in order to provide insight into how ViSSh programs are built. The following features were demonstrated:

- ViSSh's ability to import and handle large datasets.
- ViSSh's ability to apply arbitrary transforms to this data.
- ViSSh's ability to generate meaningful pictorial representations of this data.
- The compactness of ViSSh visual programs.
- The usefulness of the *Cell Dependencies* and *Broad Overview* windows.
- ViSSh's 3D interaction capabilities.

ViSSh relies on these features to provide users with an efficient data visualisation system.

To conclude, we briefly discussed the user testing we performed during the building of ViSSh.

We have demonstrated the efficiency of the system in this chapter, the only remaining question is its usability. This will be addressed in the next chapter.

Chapter 7

Cognitive Dimensions Analysis

7.1 Introduction

In this chapter, the usability of the extended spreadsheet paradigm will be analysed. This will be achieved by measuring the usability of the ViSSh prototype, using Green's Cognitive Dimensions Framework [18, 17] as the analytical tool.

The process to be followed is simple: first some real-world visualisation problems were obtained, which involved non-trivial amounts of data. These exercises were obtained from the research interests of fellow MSc students, and come from different areas of research (geology and ATM networks). Then the data was visualised, and observations were made about the process used to visualise the data; this process is discussed in detail in Chapter 6. Finally, Green's Cognitive Dimensions Framework was used to quantify these observations.

The results obtained from the analysis were then compared to those obtained by Green from two dataflow systems [18], and this comparison was used as the basis for a comparison of the underlying paradigms, namely our enhanced spreadsheet paradigm versus dataflow.

7.2 The Cognitive Dimensions Framework

As we saw in Chapter 2, Green and Petre [18] propose using a "broad brush" analysis method, as opposed to the fine-grained analysis of simple tasks traditionally used by HCI researchers. Their analysis tool, called a "cognitive dimensions framework," is task-specific, concentrating on process rather than content. It provides us with a set of mutually-orthogonal cognitive dimensions, which in essence allow the broad description of any visual programming environment by conceptually

plotting points in a 13-dimensional set of axes. Although the dimensions are meant to be orthogonal, Green admits that there is a certain amount of interaction between them, such that changing one aspect of a visual programming environment to improve its position along a given axis is likely to affect the environment's position along several other axes. Using the Cognitive Dimensions Framework it is possible to quantify the differences between existing programming environments, so they can be objectively compared and conclusions drawn about the suitability of these environments for any given task. These cognitive dimensions are summarised, together with a simple example of their application, in Section 2.8.1.

7.3 A Cognitive Dimensions Analysis of ViSSh

In this section, we shall use the Cognitive Dimensions Framework to analyse the usability of the ViSSh data visualisation spreadsheet that was described in Chapter 5. Since this program closely implements the extended spreadsheet paradigm we developed in Chapter 4, in this way we are demonstrating the usability of the extended paradigm itself.

Below, we list each of the cognitive dimensions, followed by a brief discussion of how ViSSh can be rated according to the criteria associated with that particular dimension.

7.3.1 Abstraction Gradient

Although traditional spreadsheets are *abstraction-hating* [17], the extended spreadsheet paradigm described in Chapter 4 prescribes the use of functional programming languages instead of the commonly used formula languages. Since functional programs consist mostly of function definitions (i.e., abstractions), functional programming languages can be described as *abstraction-hungry.* Hence in ViSSh we have a curious combination of two systems on opposite sides of abstraction argument (since user-defined functions can either be loaded from a file or created as needed, ViSSh supports both persistent and transient abstractions).

Since the ViSSh user does not *have to* create new functions in order to use the system, but can *if he or she wants to*, ViSSh can be said to be abstraction-tolerant. This makes ViSSh a useful tool both for the novice programmer (who usually has trouble with abstractions [40]) as well as the experienced programmer (who can take advantage of the conciseness conferred by the use of abstractions). Additionally, since no abstractions are needed to use ViSSh, it has a practically non-existent *abstraction barrier* [17].

Use of the functional language is not the only way to introduce abstractions into a ViSSh spreadsheet — the *subsheet* mechanism (see Section 5.2.2 for an explanation of this) allows for abstractions to be created by embedding entire spreadsheets into single cells.

7.3.2 Closeness of Mapping

Closeness of mapping deals with how much the syntax of the language separates the problem being solved from the program being used to solve it. ViSSh, being based on spreadsheets, has no need for "finicky syntax rules" [18] like the placing of semicolons; this gives it a better closeness of mapping than most textual programming languages. Its underlying basis in spreadsheets means that in order to use the system, the user must master only two concepts: cells as variables, and formulas as relations between variables [40]. Also, spreadsheets are devoid of control constructs (since the emphasis is on the flow of data, as opposed to the flow of control associated with imperative programming languages), which further separate the program from the problem being solved.

On the negative side, the need to build the visual representation of a dataset from graphics primitives which are transformed in a stepwise manner, semantically separates that part of the system from the actual visualisation. However, the stepwise building of the scene makes this operation simple and easy to debug, so the tradeoff was deemed worthwhile; further, since the actual data modeling and/or transformations are performed separately to the setting up of the scene, this does not negatively affect the overall closeness of mapping of the system.

The Scheme programming language, with its associated problems with regard to the overabundance of brackets, also subtracts from the closeness of mapping of the system. However, since ViSSh constrains the use of Scheme to single arithmetic expressions inside spreadsheet cells, the negative effects of Scheme are localised. Additionally, the use of Scheme is not mandatory: for example, in Figure 26 of Chapter 6, cell D2 has its formula expressed as an infix mathematical expression. The use of Scheme is mostly constrained to "power" users wishing to use advanced language features of Scheme. These users are less likely to be affected by the quirks of the language.

Hence, because ViSSh programs lack control structures and have small amounts "syntactic clutter," which is both optional and localised, they can be said to have a high closeness of mapping to the data being visualised.

7.3.3 Consistency

This is also known as the "principle of least astonishment," which states that once a part of the system has been learned, the user should be able to infer the remaining parts of it.

Like most spreadsheets, ViSSh has a high degree of consistency, since all spreadsheet cells are added, deleted and edited in the same way (see Section 7.3.12), and all inter-cell dependencies are represented in the same way, i.e., in the traditional spreadsheet cell name form (e.g. B5).

7.3.4 Diffuseness

This cognitive dimension deals with how "verbose" a given language is, i.e., whether many primitives are required on the average to express any given basic concept.

ViSSh, being based on spreadsheets, is quite concise: for example, the spreadsheet shown in Figure 26 of Chapter 6, which consists of only thirteen cells, generates user-selected 3D representations of a dataset stored in a file and displays them on an interactive window which allows zooming, panning and rotation of the scene.

Additionally, the use of functional programming languages in the extended spreadsheet paradigm ViSSh is based on (see Chapter 4) gives the system the potential of being much more concise: potentially, an entire program could fit into a single spreadsheet cell. This is, however, a degenerate case: so much functionality would be lost in the process that the resulting program could no longer be called a spreadsheet. It is, however, interesting that it *is* possible to go so far.

The inherent conciseness of spreadsheets, optionally combined with the use of a general-purpose functional language, keeps ViSSh programs small. This places less strain on the user trying to debug a spreadsheet.

7.3.5 Error-proneness

This cognitive dimension asks, "How easy is it to make mistakes with this system?"

ViSSh has no control constructs and no need for variable to contain intermediate results — both of which are non-task-related and tiresome to users [40]. According to the Yerkes-Dodson Law [68, pp. 431–432], users with a moderate level of motivation asked to perform boring, tiresome tasks are more likely to make mistakes than users faced with more interesting tasks. The use of a functional programming also reduces certain types of programming errors [21, 63], mainly due to the "cleaner" programming style imposed by these languages. This said, Scheme's syntax does leave a lot to be desired as far as brackets are concerned. ViSSh alleviates this by the use of bracket-matching text input controls, which cause the cursor to briefly jump to the corresponding opening bracket when a closing bracket is typed in.

Additionally, ViSSh's high levels of consistency and conciseness (see above) also help reduce

the propensity of errors.

7.3.6 Hard Mental Operations

If the user needs to result to external aids such as pencil and paper to aid his mental processes while using a program, that program is said to have hard mental operations.

According to Green and Petre [18], visual programming languages have a number of hard mental operations, all related to control constructs. Since ViSSh does not have any control constructs, being instead based on a pure dataflow model, it is exempt from these particular difficulties. This is because ViSSh spreadsheet cells only deal with complete datasets. For example, if some transform needs to be applied to all data items smaller than zero, then an entire dataset will be analysed by the relevant spreadsheet cells, and the resulting dataset will consist of the union of all values greater than or equal to zero (which will remain untouched) and those values which were smaller than zero (and will have been appropriately transformed).

There are, however, two sources of hard mental operations in ViSSh: keeping track of "is depended on by" relationships (the opposite of "depends on" relationships) and the idiosyncrasies of the Scheme programming language.

Keeping track of "is depended on by" relationships. This is a hard mental operation since it is not immediately obvious which cells are depended on by any given cell (in contrast, it is easy to see which cells any given cells depends on, since those cells are explicitly stated in the cell that depends on the others). However, the *Show Dependencies* window (see Section 5.2.4 for an explanation of this feature) greatly simplifies this process.

Scheme. This has two main areas as far as hard mental operations are concerned: keeping track of brackets and the fact that expressions are specified in the prefix form. Keeping track of brackets is a hard mental operation because every operation must be surrounded by brackets, e.g., the inline expression $a*b+c*d$ is expressed as `(+ (* a b) (* c d))`. This example also illustrates the problems posed by the prefix notation. While this notation is semantically cleaner (everything is treated as a function application), deciphering a complex expression can be a taxing experience. However, the use of Scheme is optional. The typical user that chooses to use Scheme expressions will be a "power" user, who will have the necessary mental tools to deal with this complexity. Ordinary users need not be exposed to this, since a software layer exists that converts expressions from the infix format commonly used by mathematicians into the internal Scheme representation.

Hence ViSSh can be said to have few hard mental operations, with a few caveats. Firstly, it is possible to create such an obfuscated spreadsheet that it is difficult to keep track of the flow of data. Secondly, the use of the Scheme programming language introduces hard mental operations, but this should only be attempted by seasoned programmers who wish to highly tune their programs; ordinary users need not resort to such measures.

7.3.7 Hidden Dependencies

A hidden dependency is a relationship between two parts of a program such that the one depends on the other, but this dependency is not fully visible.

Although the spreadsheet model that ViSSh is based on suffers from hidden dependencies (one can see what cells any given cell depends on, but not what cells depend on any given cell), the *show dependencies* window implemented in ViSSh circumvents this problem (see Section 5.2.4 for a discussion of this feature). This window contains a dataflow diagram representation of the program implemented by the spreadsheet, which can be examined to find out the data dependencies between individual spreadsheet cells. This diagram can be used to find out both which cells depend directly on which other cells, as well as finding out all cells that ultimately depend on any given cell (since circular references are not allowed, all cells that are somehow affected by any given cell will be all the nodes of the tree rooted at that cell).

Since the dataflow diagram is built algorithmically, it is always at least as readable as a hand-edited data flow diagram, and does not need constant maintenance.

Therefore, ViSSh can be said to have the best of both worlds: the simplicity afforded by the one-way links provided by the spreadsheet paradigm [17], without the inherent hidden dependencies; this is because all data dependencies can always be examined by the use of the *Show Dependencies* window.

7.3.8 Premature Commitment

This occurs when the user is forced to make a decision before all necessary information is available.

ViSSh can be considered to have a certain amount of premature commitment for the simple reason that spreadsheets have boundaries — the user must decide, before starting, in which directions a spreadsheet will grow. Guessing wrong will mean that the user will find him or herself in a position where a cell must be inserted to the left of a cell that is in column A, for example. However, ViSSh has a low viscosity (see Section 7.3.12 below) and hence correcting problems of this nature is not a

big undertaking.

Premature commitment due to ordering constraints is not present in ViSSh, since a spreadsheet can be built by placing cells in any order the user wishes.

7.3.9 Progressive Evaluation

If a system supports progressive evaluation, then any partially-completed program can be tested at any time. This contrasts with most text-based languages, where a program must be syntactically complete before it can be compiled and tested.

ViSSh, being based on spreadsheets, has a high degree of support for progressive evaluation. This is a necessary feature for a system used for exploratory programming (such as data visualisation). This means that the user can develop his visualisation in stages (since the system automatically keeps track of data dependencies and recalculates the spreadsheet when necessary) which can be individually tested.

7.3.10 Role-expressiveness

This cognitive dimension is used to find out how difficult it is to answer the question "what does this part of the program do?"

ViSSh has no explicit support for role-expressiveness, since it was felt that adding something like a separate comment layer would add to the viscosity of the system (see below) and hence would detract from its usefulness as an exploratory-programming system. Users can, however, use text-entry cells to comment on the functions performed by nearby cells.

Therefore ViSSh can be considered to have a low score for role-expressiveness.

7.3.11 Secondary Notation

This is extra information carried by other means than the official syntax. An example from text-based programming languages could be the indenting of loop bodies relative to loop-delimiting statements.

ViSSh has some direct support for secondary notation by giving the user the ability to write a block of text that is associated with any given spreadsheet. The users may use this facility in any way they see fit; the text is stored in the same file as the spreadsheet and pops up when the spreadsheet is opened. This commenting is not on a cell-by-cell basis; it was felt that this would interfere with the rapid development cycle typical of a data visualisation tool. Instead, the comments apply to the

spreadsheet as a whole, and allow users to comment their programs in a way that does not interfere with the frequent editing associated with data visualization.

Additionally, users may use the standard secondary notations associated with spreadsheets, such as clustering cells into functional units.

7.3.12 Viscosity

Viscosity is defined as *the cost of making small changes to an existing program.* In ViSSh, this is quite small; to back up this claim we submit that all editing (in the context of spreadsheets in general, and ViSSh in particular) can be seen to belong to one of two categories: cell modification and insertion.

- **Modification** consists of changing the attributes of a cell (such as the cell it gets its data from), replacing the cell with one of a different type or deleting a cell completely. The first case, editing a cell's attributes has no effect whatsoever on neighbouring cells (recall that no cell may directly set the state of another cell). Replacing and deleting have no effect either because each cell fits exactly into a grid position, so all editing changes are by definition completely localised.

- **Insertion** of cells into a spreadsheet does not affect neighbouring cells either, simply because a cell can be inserted anywhere in the spreadsheet regardless of where the cells that logically "belong" with that cell are. In other words, if the user wishes to add a cell that gets its data from cell C5, the new cell does not need to reside anywhere near cell C5. Since all cell connectivity is handled by textual cell references, the "spaghetti" problem commonly associated with dataflow systems simply does not appear.

Since low viscosity is a desirable property of exploratory systems such as data visualisation systems, we believe that this low viscosity is one of the most important properties of ViSSh.

7.3.13 Visibility and Juxtaposability

Visibility is defined as *the ability to view components easily*, while juxtaposability is defined as *the ability to place components side by side.* These two are related in that, combined, they imply the ease, or difficulty, of seeing how any two parts of a program relate to each other.

Spreadsheets traditionally have asymmetric visibility and juxtaposability — this is because formula cells contain two separate sets of information: the formula and the formula's result. As a

result, when the spreadsheet is in "results mode," it is difficult to compare formulas (since only one formula at a time can be viewed in this mode), while when it is in "formula mode," it is difficult to compare results (since only one result can be viewed at a time in this mode). ViSSh circumvents this problem by having dedicated output cells; in this way, all formulas *and* all results can be viewed and compared simultaneously.

The large cells used by ViSSh cause a problem with visibility, since not many of them can be displayed simultaneously. This problem is alleviated by the use of the *Broad Overview* and *show dependencies* windows (see Section 5.2.4 for a discussion of these features). These windows display miniature versions of the cells in the spreadsheet, greatly increasing the number of cells that can be viewed simultaneously and hence visibility. The fact that the *Broad Overview* window can be scrolled independently of the main editing window also enhances visibility. The *Show Dependencies* window, additionally, exposes the relationships between spreadsheet cells, further enhancing the visibility of the system.

ViSSh has a high amount of juxtaposability in that there is no limit to how many windows can be open at any given time, or on what these windows are displaying. For example, it is possible to have three 3D render windows looking at the same object, one each for top, left and front view. Any changes made to the object will be visible in the three views simultaneously.

Therefore ViSSh can be said to have both high visibility and high juxtaposability; both qualities are important in a data visualisation system.

7.4 Summary of Results

In Section 7.3 we described the results of our Cognitive Dimensions analysis of ViSSh. This analysis reveals that ViSSh has a low level of diffuseness (i.e., programs tend to be concise), allows progressive evaluation (i.e., a partially-finished program can be run) and has low viscosity (i.e., making any small change to a program is easy). These properties show ViSSh to be very suitable for exploratory programming, in a similar way to the spreadsheets that it is based on; this means that the extensions that we have made to the spreadsheet paradigm to increase its suitability for data visualisation have not adversely affected this central feature of the spreadsheet paradigm.

In Table 1 we list the results of the cognitive dimensions analysis, together with those given by Green for two other visual programming languages, ProGraph and LabVIEW [18]. This comparison is interesting because both LabVIEW and ProGraph use the dataflow paradigm, which is currently the most common with data visualisation systems. Although the two systems are fairly different in

Cognitive Dimensions Axis	ViSSh	LabVIEW	ProGraph
Abstraction	tolerant	tolerant	hungry
Closeness of Mapping	high	high	high
Consistency	high	high	high
Diffuseness	low	med	high
Error-proneness	low	low	low
Hard Mental Operations	few	some	some
Hidden Dependencies	low	low	low
Premature Commitment	med	high	high
Progressive Evaluation	good	none	good
Role-expressiveness	low	low	low
Secondary Notation	med	low	low
Viscosity	low	high	low
Visibility	high	high	medium

Table 1: Cognitive Dimensions comparison of ViSSh, LabVIEW and ProGraph (values for LabVIEW and ProGraph obtained from Green and Petre [18])

appearance (LabVIEW is a pure dataflow system while ProGraph uses electronic circuit diagrams as its user interface metaphor), their cognitive dimensions analysis probes deeper and reveals that they are, in reality, quite similar in nature.

By comparing the three systems in Table 1, it is possible to list the advantages the extended spreadsheet paradigm (as implemented in ViSSh) has over the dataflow diagrams normally used in data visualisation (as implemented in LabVIEW and ProGraph):

- ViSSh has less diffuseness than LabVIEW or ProGraph. This is because the extended spreadsheet paradigm ViSSh is based on (see Chapter 4) combines spreadsheets with functional programming. Spreadsheets are inherently concise, and when combined with text-based functional languages, the resulting system has the ability of being more concise than the purely graphical dataflow diagrams.

- ViSSh has fewer hard mental operations than the other two systems. This is as a result of the fact that users of dataflow systems have to perform backtracking when analysing the flow of data as it goes through conditional nodes. Since ViSSh nodes (i.e., spreadsheet cells) deal only with complete datasets (the effects of conditionals are entirely hidden by the cells performing the conditional operations), users are spared the pencil-and-paper tricks commonly used by dataflow system users. Although the use of the Scheme programming language imposes some

cognitive load on users, this will be limited to "power users" who can deal with it; most users will make use of the ability to enter expressions in the infix form.

- ViSSh users are less vulnerable to premature commitment than users of dataflow systems, since the connections between ViSSh cells are implicit (although they can be made explicit). Therefore ViSSh is immune to the "spaghetti syndrome" commonly associated with heavily-edited dataflow diagrams.

- ViSSh has slightly more support for secondary notation than ProGraph and LabVIEW, mainly due to the fact that users have the ability to comment a spreadsheet, in addition to using physical grouping of cells according to functionality. LabVIEW and ProGraph are quite limited in this area [18].

Although in this analysis we seem to downplay the role of functional languages (one of the three extensions to the spreadsheet paradigm described in Chapter 4) and suggest that they be used only by "power users," this is not the case. Any software system has two types of user: casual/learner users and power users. ViSSh addresses the needs of both of these users by making the most complex of the spreadsheet paradigm extensions, i.e., using a functional language, an optional user interface feature. Inexperienced users are still making use of a functional programming system, but in a way that is familiar to them (i.e., infix mathematical expressions). As they become more experienced, users will want to abandon the "training wheels" provided by the infix expression parser in favour of the full power provided by the functional programming language Scheme. By comparison, users of traditional spreadsheets only have one choice of formula language. While this is a comfort to beginners, expert users looking for more expressive power when building their spreadsheets are often forced to work outside the spreadsheet paradigm, by using embedded BASIC routines, ActiveX controls, etc.

7.5 Chapter Summary

In this chapter we have used the experience gained while building the real-world visualisations described in Chapter 6 to perform a Cognitive Dimensions Analysis of ViSSh (and so, indirectly, of the extended spreadsheet paradigm that ViSSh is based on). Once we had obtained the results afforded by this analysis, we proceeded to compare ViSSh to two dataflow systems, based on the analysis of these systems performed by Green [18]. This comparison was relevant because the

two systems compared against, LabVIEW and ProGraph, are representative of the class of dataflow systems that ViSSh is meant to improve upon.

In our comparison of ViSSh, LabVIEW and ProGraph we found that the extended spreadsheet paradigm underlying ViSSh does indeed improve upon the dataflow paradigm underlying LabVIEW and ProGraph, while not introducing any traps for the unwary user. The implication of this observation is that data visualisation systems based on the extended spreadsheet paradigm should be at least as usable as the current dataflow systems, with the possibility of being better tools.

Chapter 8

Conclusion

8.1 Introduction

In this dissertation we have examined the usefulness of spreadsheets for data visualisation applications. We have also outlined the shortcomings inherent in spreadsheets with regard to data visualisation, and described an extension to the basic spreadsheet paradigm that remedies these. Finally, we have described a software prototype that implements this extended paradigm and our evaluation of this prototype. This chapter gives our findings at both the theoretical and implementation level, and concludes with possible future work in this direction.

Our main contribution lies in the discovery of an extended spreadsheet paradigm that allows spreadsheets to be efficiently used as data visualisation systems. This was based on our novel functional analysis of the spreadsheets paradigm and the discovery, presented together with an algorithmic proof, of the fact that spreadsheets and dataflow are equivalent in nature. We have also created a novel data visualisation system based on spreadsheets that allows arbitrary multidimensional datasets to be manipulated and visualised.

8.2 Analytical Results

We have analysed spreadsheets at their most basic level, and discovered several interesting properties. Firstly, we have found that spreadsheets can be described as *Applicative State Transition* systems, as described in Section 2.5. This means that spreadsheets are functional in nature. Based on this demonstration, we have also found that spreadsheets can be described as consisting of an editing system layered on top of a functional computational engine (see Section 3.3.3). Both of

these discoveries led to the finding that spreadsheets are equivalent to dataflow systems. Section 3.5 has an algorithmic proof for this finding.

With regard to data visualisation, we have found that spreadsheets are indeed useful tools for this task, as reported by Levoy [30] and others. We have also found them to be superior to dataflow visualisation systems due to the lack of clutter normally associated with heavily-edited dataflow diagrams. However, we have found that spreadsheets in their current form have problems with regard to data visualisation, due mostly to the difference in the volumes of data that spreadsheets were originally designed to handle (hundreds of data values), versus those normally handled by data visualisation tools (tens of thousands to millions of data values).

We have further found that this deficiency can be further classified into three categories, namely *volume of data*, *flexibility of computation* and *computation time*. We have found that the following techniques address each of these:

- The problem with the *volume of data* that needs to be processed is that spreadsheets would need to be tens of thousands of cells wide or tall to contain these datasets (since datasets can normally only be processed as cell ranges that are horizontally or vertically arranged). We have found that allowing entire datasets to be stored and manipulated in each spreadsheet cell, as described in Section 4.3.1, drastically reduces the size of spreadsheets, since the majority of spreadsheet cells then contain formulas instead of raw data.

- *Computation time* is a problem related to *volume of data*: the more data that is processed, the longer it will take to process. However, we have found that in the context of data visualisation a solution exists. In most data visualisation exercises part of the data is not displayed, either as a result of decimation (in broad overviews) or clipping (in zoomed-in views). We have found that in these cases the technique of lazy evaluation, described in Section 4.3.2, can reduce computation times; this is because lazy evaluation avoids performing calculations whose results will not be made use of.

- Spreadsheets use domain-specific formula languages; this results in a reduced *flexibility of computation*, since these formula languages are typically both highly specialised and not extensible. Based on our finding that spreadsheets are functional in nature, as discussed above, it is possible to alleviate this problem by using a standard functional programming language (see Section 4.3.3). This gives casual users the benefit of pre-built function libraries which take over the role of the specialised formula languages, while simultaneously permitting experienced users to extend the formula language to fit their needs.

8.3 Experimental Results

In order to test and demonstrate the extended spreadsheet paradigm described in Section 4.3, we have built a software prototype, *ViSSh*; this prototype is described in detail in Chapter 5.

8.3.1 User Interface Issues

During the building and testing of this program we found some general usability problems with spreadsheets in general and implemented solutions for them. We have not made these part of our extended spreadsheet paradigm since none of them is directly related to data visualisation.

Firstly, we found that the lack of explicit links between spreadsheet cells is a source of problems when debugging spreadsheets; this is in spite of the fact that this lack of links is largely the reason that spreadsheets are neater than dataflow diagrams, even after heavy editing. The reason is that although it is easy to see which cells any given spreadsheet cell depends on (simply read the formula), the reverse relationship (i.e., which cells are affected by any given cell) is not as obvious; in a moderately complex spreadsheet, the relationship is effectively hidden. The solution we implemented for this problem is to have a window containing an equivalent dataflow representation of the spreadsheet (derived using the algorithms in Section 3.5). This dataflow representation is algorithmically generated and so is less cluttered than a hand-edited dataflow diagram, even after the spreadsheet has been subjected to heavy editing.

Secondly, we found that editing spreadsheets was frustrating because of the small number of cells visible at any given time. This agrees with Nardi's study [41], which reports that "users have a strong preference for being able to view and access as much data as possible without scrolling." Our solution to this problem was to allow users to summon a view of the spreadsheet that substitutes a small icon for each spreadsheet cell. Each of these icons is much smaller than a "normal" spreadsheet cell, and so many more spreadsheet cells can be viewed at any time. Since this "broad overview" window may be scrolled independently of the main editing window, it can be used to navigate the spreadsheet without losing track of the current position.

8.3.2 SubSheets

An interesting offshoot of the discovery that spreadsheets are functional in nature is that, if spreadsheet cells are allowed to evaluate cells in other spreadsheets, a functional link between the two spreadsheets is formed. This link is implemented in the SubSheet facility of ViSSh, which allows developers to create encapsulated, reusable spreadsheets that can be used by other spreadsheets

in the same way that function libraries are used by programming languages (the current model of spreadsheet reuse is more akin to macro expansion). This encapsulation arises from the fact that the master spreadsheet and the subsheet can only communicate via well-defined points, with all other information being inaccessible (see Section 5.2.2 for more information). SubSheets provide ViSSh with a hierarchical abstraction mechanism that promotes code sharing and reuse, both of which are highly desirable properties of a data visualisation system. This mechanism can be used to build large, user-contributed libraries of useful spreadsheets, similar to the script libraries in existence for most established data visualisation systems. The functional nature and data hiding inherent in the SubSheet mechanism also assist with the maintenance of function libraries, since the coupling between master and subsheets is quite loose, reminiscent of shared libraries. We believe that SubSheets have a lot of research potential, and outline some possible areas of research in Section 8.4.

8.3.3 Cognitive Analysis

We have used Green's Cognitive Dimensions Framework [18] to analyse our software prototype, and compared it to two typical dataflow systems analysed by Green (ProGraph and LabVIEW) [18, 17]. We have found that ViSSh is generally as usable as these two systems, with several areas being improved on — these include the ability to create more concise visual programs (i.e., "smaller" programs both in terms of number of primitives and needed screen area) and lower cognitive load on the user when editing visualisations; Chapter 7 contains the usability analysis as well as our conclusions. The cognitive dimensions framework has also shown that the ViSSh prototype, and hence the extended spreadsheet paradigm, is well-suited to exploratory programming. This shows that our extensions to the spreadsheet paradigm have not negatively affected this useful trait of the spreadsheet paradigm. The overall implication of the results of the cognitive dimensions analysis is that, from a cognitive point of view, systems based on the extended spreadsheet paradigm can be used to replace the dataflow-based visualisation systems currently in use.

8.4 Future Work

Although the extended spreadsheet paradigm solves the problems inherent in spreadsheets with regard to data visualisation, we believe that more work needs to be done to address the needs of the novice user. The very flexibility provided by the use of a functional language can be confusing to a newcomer to the system, and the choice of the Scheme programming language is not ideal as far as

non-programmers are concerned. We have partially remedied this by allowing mathematical expressions (i.e., the vast majority of spreadsheet formulas) to be expressed as infix, but list manipulations must still be expressed in Scheme. We believe that there are enough computer languages already in existence, and hence will search for a more "beginner-friendly" functional language. Additionally, the set of available graphics primitives could be enlarged: extending the graphical vocabulary of ViSSh beyond what was needed for testing was considered to be "non-interesting" and hence postponed.

8.4.1 The SubSheet Mechanism

Several avenues of research arise from the SubSheet mechanism discussed in Section 5.2.2. This mechanism allows spreadsheets to evaluate cells from other spreadsheets in a functional manner, thereby providing spreadsheets with the ability to make use of encapsulated spreadsheet "function libraries." However, if the mechanism were modified so that the flow of data between spreadsheets used unix sockets, it would be possible to distribute spreadsheet computations. A "super-spreadsheet" could be built that links several computers together to solve a computationally-intensive problem. Since ViSSh makes use of lists for its inter-cell communication, data could be "streamed" between spreadsheets one item at a time. If this were combined with a multi-threaded spreadsheet recalculation mechanism (where cells that do not depend on each other are evaluated simultaneously), a highly efficient yet simple distributed computation mechanism could result. Since the SubSheet mechanism is functional in nature and has no side-effects, parallelism is easy to achieve.

8.4.2 Multiuser Spreadsheets

Another interesting avenue of research opened up by internetworked spreadsheets would be that of collaborative work. If several users work on separate spreadsheets that are linked to each other in this way, many of the problems that present themselves with remote shared workspaces would simply not exist. This would be because each user would work on their own local spreadsheet, and only the results of recalculations would be sent over the network. This would eliminate the need to keep several copies of the same spreadsheet synchronised over the network; instead of there being only one virtual spreadsheet shared by many users, there would be many spreadsheets communicating with each other via the SubSheet mechanism. When any user modifies their spreadsheet, due to the purely functional nature of the SubSheet mechanism, only the end result of that modification need be transmitted — the modification itself remains local, thereby reducing network traffic (this

is because some modifications, for example adding of comment cells, will not affect the end result) and eliminating the need for complex procedures aimed at keeping all copies of the spreadsheet synchronised.

8.4.3 Alternative Data Storage Models

Finally, our extended spreadsheet paradigm stipulates storing multiple values in each spreadsheet cell. We have opted to use lists to implement this, but believe that other storage mechanisms merit research. Consider, for example, the possibilities opened by storing a relational database table in each spreadsheet cell; database query languages such as SQL are most probably declarative in nature and so would be compatible with the functional spreadsheet paradigm. Formulas could be expressed as database queries on other cells, such as `=select name from B7 where age>21`. This would however comprise a different extension of the spreadsheet paradigm that that outlined in this dissertation, since a database query language would be used instead of a functional programming language. We believe that functional programming languages maximise flexibility in this case, and hence made use of them; further research may reinforce or disprove this view.

8.5 Conclusion

In this dissertation we have shown that the spreadsheet paradigm, suitably extended, can be used for data visualisation purposes. We have presented one such extension, devised by us, together with a supporting theoretical framework. This theoretical framework consists of two facts, which we have demonstrated in this document to be true: firstly, spreadsheets consist of a functional computational engine underlying a cell editor; secondly, spreadsheets and dataflow systems are equivalent in nature.

The extension we have made to the spreadsheet paradigm is three-fold: firstly, we store lists of items, instead of single items, in each spreadsheet cell; secondly, we use a functional language instead of a traditional formula language; finally, we make use of lazy evaluation in recalculations. These extensions to the spreadsheet paradigm allow a spreadsheet user interface to be efficiently used for data visualisation systems, without losing the "feel" of a spreadsheet.

We have built a novel data visualisation system based on this extended paradigm, and used Green's *Cognitive Dimensions Framework* [18] analytical tool to demonstrate that it is at least as usable as current visualisation systems. By demonstrating the usability of the prototype, we have

in effect demonstrated the usability of the extended paradigm. We have thus created a user interaction paradigm for data visualisation systems that is based on spreadsheets, backed by a theoretical framework that states that it is as expressive as current dataflow systems (and at least as usable as these) but which is better suited for data visualisation due to the better support for exploratory programming that is inherent in spreadsheets.

Appendix A

ViSSh 1.x User's Manual

A.1 Introduction

A.1.1 Conventions

Throughout this manual, the following conventions will be used to aid in understanding of the text:

- Shell commands will be typeset in a **`courier bold`** font.
- Menus and menu items will be typeset in **Times Bold**.
- Keystrokes will be typeset in **Helvetica bold**.
- The default number base is decimal. Hexadecimal numbers will be prefixed with `0x`, e.g., `0xffd0`.
- Scheme lists and code will be typeset in `courier`.
- Names of controls in spreadsheet cells will be typeset in Helvetica.

A.1.2 ViSSh Is a Work in Progress

Although ViSSh has been deemed stable and feature-complete enough to merit "Release 1.0" status, data visualisation packages are never "finished." Because data visualisation is such a fluid discipline, the tools that we use should be equally fluid. ViSSh has been designed with this in mind, and can be extended in three ways, ordered according to decreasing ease and increasing run-time performance:

1. Encapsulating spreadsheets to use them as "subroutines"

2. Scheme functions can be loaded at run-time and used in spreadsheets.

3. If run-time performance is of the essence, new "builtin" cell types can be (reasonably easily) written in C++ and linked into the binary.

Therefore, if there is something that you feel ViSSh is lacking, write it yourself — I'm busy right now.

A.1.3 ViSSh and the Spreadsheet Paradigm

ViSSh is a data visualisation package that is based on the spreadsheet paradigm. This contrasts with most other data visualisation systems, which are instead based on the dataflow paradigm. The reasons behind the use of a different paradigm are beyond the scope of this manual, but can be summarised into the following points:

- Most computer-literate users are familiar with the spreadsheet paradigm. This includes the use of declarative spreadsheet formulas and the `A1` (column-row) method of referring to locations of the spreadsheet. The same cannot usually be said of the dataflow paradigm.

- Dataflow visual programs tend to become cluttered when more than a few nodes are present in the dataflow graph, especially when these are moved around often as is the case with exploratory programs.

This manual assumes a basic working knowledge of spreadsheets. If you understand what the phrase *"Put a formula in B76 that will add up the range B5:B75"* means, you can start using ViSSh.

A.2 The Basics

A.2.1 Introduction

This section will introduce the basic concepts underlying ViSSh. Here the basic concepts underlying ViSSh will be explained, and a short, not quite useful example (*Hello ViSSh*, if you will) will be explained to give the reader a feel of how to work with ViSSh. Some experimentation at this early stage is highly recommended. Once you have experimented with this, you should read Section A.3 to be able to make full use of ViSSh.

A.2.2 Basic Concepts

The Declarative Paradigm

ViSSh is a declarative programming environment. This means that instead of describing *how* data must be transformed by programs, the user must describe *what* transforms must be applied to the data. Although the difference seems merely semantic, it is fundamental. It can be explained by comparing the code fragments shown in Figure 38. The code fragment on the left is written using the *imperative* programming paradigm, where each stage of the calculation must be described in full. The code fragment on the right, on the other hand, uses the *declarative* programming paradigm, where only the desired result is specified. Although the imperative programming paradigm is quite commonly used, the declarative paradigm is more concise, letting the programmer concentrate on solving the problem in a more abstract fashion.

```
double result = 0;

for (i = 1;i <= 72;i++)
   result += B[i];

return result;
```

```
=SUM(B1:B72)
```

Figure 38: The code fragment on the left was written in the C programming language (imperative), while the one on the right was written using a typical spreadsheet formula language (declarative).

Spreadsheet Cells as Functions

A ViSSh spreadsheet is made up of a set of cells. Each of these cells performs a specialised task, such as applying a function to a list of values or generating a 3D ball.

Each cell can be though of as a function which takes some arguments and returns a result, for example a "Ball" cell can be though of as a function that takes radii (numbers) as arguments and returns balls (having the given radius) as a result. Cells read their arguments as lists, and return their results in lists as well. Consider a hypothetical cell that multiplied its input by two. If the input to this cell were the list `(1 2 3 4)`, then its output would be the list `(2 4 6 8)`.

Not all cells behave like this though — this is just the most general case. There are two other types of cell, known as *data sources* and *data sinks*. Data sources either generate or retrieve information, while data sinks dispose of it. Data sources implement functions that take no arguments, and are used to do things like generating lists of numbers or collecting data about user interaction. Data sinks, on the other hand, implement functions that return no values. These are used mainly to display information to the user.

A.2.3 Starting and Exiting ViSSh

Starting ViSSh is done via the command line. Type **`vissh`** at the command prompt, and a few seconds later you will be presented with an empty spreadsheet window and the ViSSh *Node Palette*, which is used to add or replace spreadsheet cells. Figure 39 illustrates the startup appearance of ViSSh.

Exiting ViSSh can be done in one of two ways — either selecting **Exit** from the **File** menu, or pressing **Ctrl-C** in the shell ViSSh was started from. The former method is the recommended one, since you will be asked to save the current spreadsheet and the shutdown will be more orderly. The latter method should only be used in case of emergency.

A.2.4 Editing the Spreadsheet

The larger of the two windows in Figure 39 is the main spreadsheet editing window. Figure 40 illustrates this. You may have noticed that the cell in the top left (cell A1) is different — it is black instead of white, and in the screenshot in Figure 40 it shows up in a different colour. That is because that cell is the *current cell*. The current cell is the one that can be operated on, and which has the keyboard focus. To change which cell is the current cell, either click on the cell (if the cell is occupied, you must click on the coloured bar on the upper part of the cell) or use the **arrow keys**.

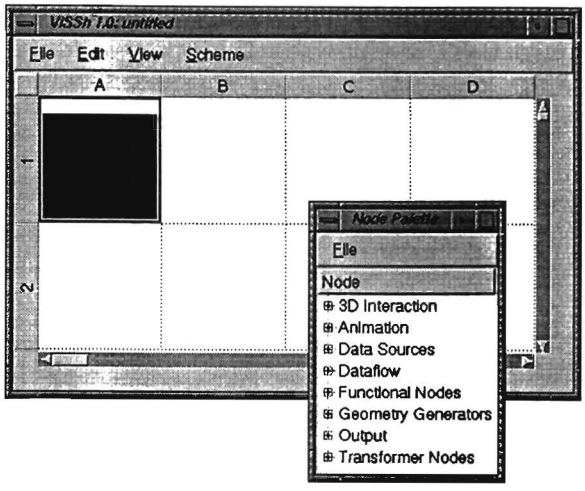


Figure 39: This is the initial appearance of ViSSh. The larger window is the spreadsheet editing window, while the smaller one is the node palette used to add new cells into the spreadsheet.

Adding a new cell to the spreadsheet is quite simple: first click on the listview in the node palette to expose the desired cell type (they are grouped according to functional category), and then drag and drop the cell into the desired grid position (the cursor will change shape to indicate that a cell is being dragged). Overwriting an existing cell with another is done in exactly the same way.

To delete a cell, first make it the current cell as described above, and press **Ctrl-Delete** (the **Delete Current** option in the **Edit** menu will perform the same action). The cell will be deleted and the grid position will become vacant. To undo accidental deletions (or any other changes to the spreadsheet grid), press **Ctrl-Z** and the spreadsheet will be restored to its previous state. There is no limit to the number of undo actions one can perform — the undo buffer extends back in time until the time the spreadsheet was created or loaded in (whichever was more recent). ViSSh also supports the clipboard — one can cut, copy and paste cells by using the standard keystrokes **Ctrl-X** (cut), **Ctrl-C** (copy) and **Ctrl-V** (paste). These operations are all also available in the **Edit** menu.

Spreadsheets are loaded in and saved using the relevant options in the **File** menu.

Area Selections

In order to be able to perform operations on groups of cells, ViSSh allows for the creation of rectangular selections of cells. There are three ways of creating a selection: selecting with the mouse; selecting with the keyboard and selecting entire rows or columns.

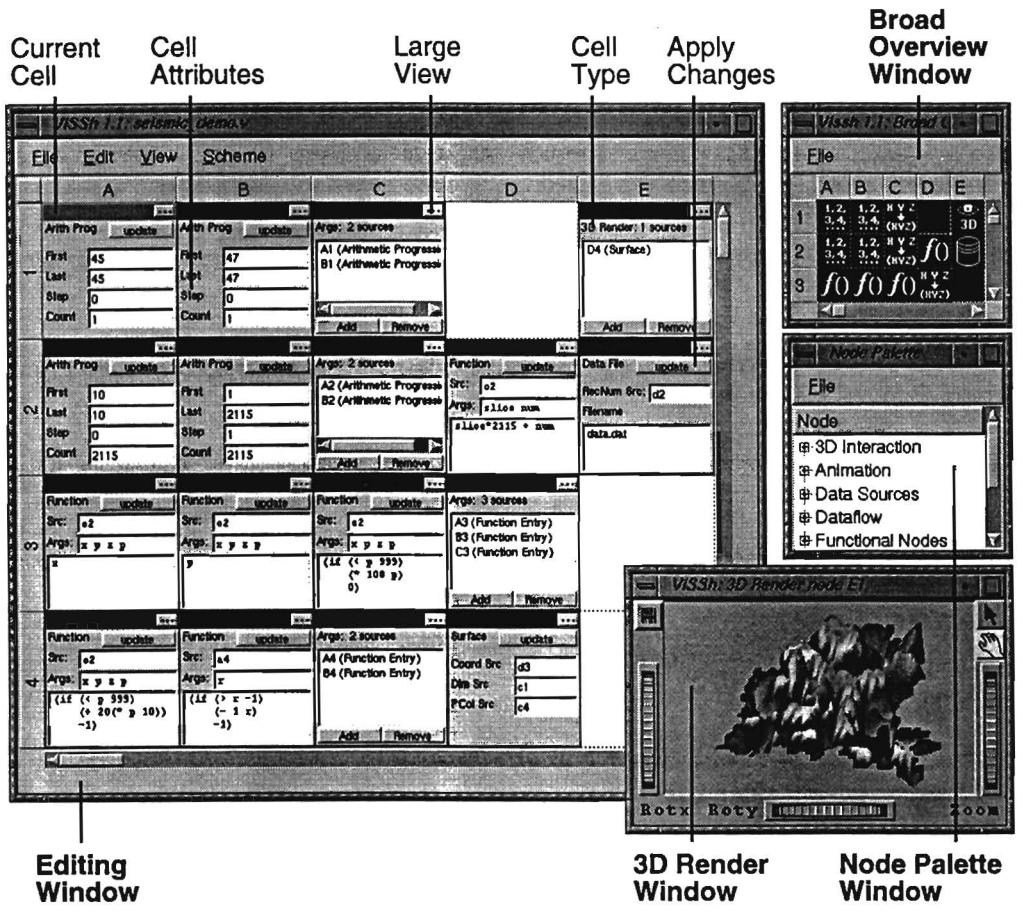


Figure 40: This is the main editing window, demonstrating what a typical ViSSh session looks like. The spreadsheet depicted here visualises seismic data read from a file.

Selecting with the Mouse This is the easiest way to select an area. First left-click the mouse on the cell that will be on the top left corner of the rectangular selection (if the cell is populated, click on the coloured bar at the top of the cell). This will also have the effect of making that cell be the current cell. Then drag to mouse to the cell that will be at the bottom right corner of the selection. When you let go of the mouse button, the area will be selected.

Selecting with the Keyboard When selecting with the keyboard, the current spreadsheet cell is the one that will be at the top left corner of the selection. To start creating the selection, hold down the **Shift** key and use the **arrow keys** to grow the selected area. When you release the **Shift** key, the area will be selected.

Selecting Entire Rows or Columns To select an entire row or column, just left-click the mouse on the row or column header. The entire row or column will then be selected.

Once an area has been selected, cut (**Ctrl-X**), copy (**Ctrl-C**) and delete operations will use the selected area instead of the current cell. Note that, to avoid accidents, there is no hot-key for the deletion of selected areas. Instead, the **Delete Selection** option in the **Edit** menu must be used.

When pasting (**Ctrl-V**) an area from the clipboard, the current cell indicates where the top left cell of the pasted rectangular area will reside. The pasted cells will overwrite whatever was in the cells they occupy.

To clear the current selection (so that nothing is selected), left-click the mouse on the button on the top left of the grid, between the row and column headers.

A.2.5 A Simple Example

The aim of this example is to make a cone out of spheres (using the well-known parametric cone equation) and display it in three dimensions. Figure 42 demonstrates the "finished product." The key to editing with ViSSh is the "Node Palette," illustrated in Figure 41.

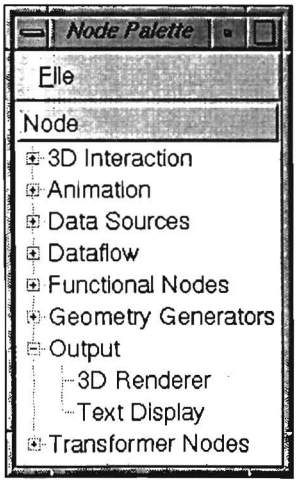


Figure 41: This is the ViSSh Node Palette. All editing is done with it, by dragging cell names from the palette (which is organised as a "tree view" of cell categories) and dropping them into the actual spreadsheet.

From the node palette the different nodes are selected from their groupings, and dragged into the spreadsheet cell where the user wishes them to be. The spreadsheet is constructed as follows:

1. An "Arithmetic Progression" cell is dragged from the node palette into cell A1, and the relevant parameters are entered into it. In this case, we want a set of numbers running between

-10 and 10, in steps of 2.

2. A "Cartesian Product" cell is dragged from the node palette into cell B1. Cell A1 is then added twice as a source, so the output of this node will be {(-10 -10) (-10 -8) ... (10 8) (10 10)}.

3. A new "Function" cell is now dragged into cell C1. The source of the arguments was specified as cell B1, so the list described above constitutes the argument set for this new node. The arguments are named x and y, so when the pairs in the argument list are substituted into the equation, the first item in the pair will be substituted for all the x's and the second for all the y's. The actual function is then entered (since the actual cell is rather small, an expanded view is used for this. The expanded view is activated by clicking on the "..." icon in the top right of the node).

4. An "Argument List" cell is then dragged into cell D1, and its sources are set to be cells B1 and C1. This node collates lists of arguments to generate a new argument list. As an example, if two lists {(1 2) (3 4)} and {(11 12) (13 14)} were passed to an Argument node, the resulting list would be {(1 2 11 12) (3 4 13 14)}. This is needed in order to generate the required arguments for the node in the next step (cell D2 in Figure 42, not a part of this exercise, shows part of the list output by this node).

5. Another "Arithmetic Progression" cell is dragged into the spreadsheet, this time into cell A2. This time, the "step" field is set to 0 (indicating that the same number will always be generated) and the "count" field is set to 100 (indicating that a list containing one hundred "1"s will be generated).

6. A "3D Ball" cell is now dragged into cell B2. Its source is then set to cell A2. This node takes each of the arguments generated by its source cell and uses them as the radius of a new sphere that it creates and adds to its output list. Therefore, the number of spheres generated by this node equals the size of the list that contains the ball radii.

7. A "3D Translation" cell is dragged into cell E1. This node will be used to move the balls from position (0,0,0), where they appear by default, to their proper positions. The balls are taken from the Ball node we just placed in cell B2, while their positions are taken from the Argument node in cell D1.

8. Finally, a "3D Renderer" node is dragged into cell E2, and its source is set to the 3D Translation cell in cell E1. When the expanded view for this node is opened by clicking on the "..." icon in its top right corner, the cone of spheres can be viewed and manipulated by the user.

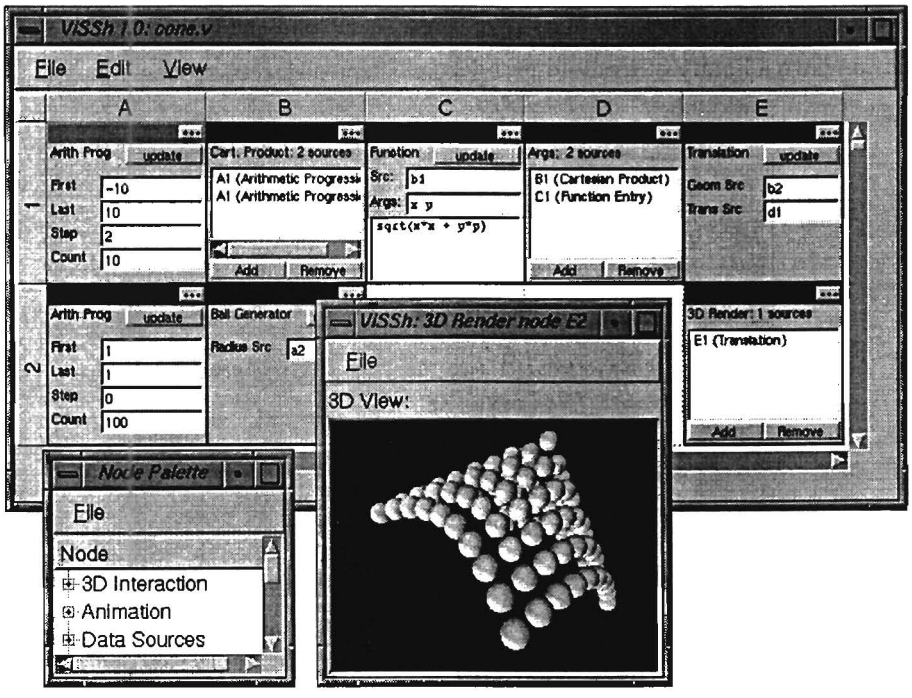


Figure 42: This is the end result after following the steps described in Section A.2.5

A.3 Advanced Techniques

A.3.1 Subsheets

Scheme allows for code reuse by means of subsheets. This system allows entire spreadsheets to behave as though they are contained in a single spreadsheet cell. The main advantage of subsheets is that one can have a library of ready-to-use, debugged spreadsheets, reducing the time it takes to create a new data visualisation.

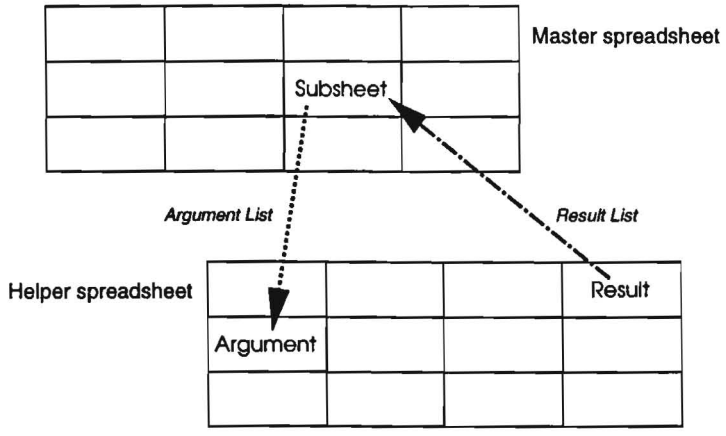


Figure 43: This illustrates the mechanism whereby several spreadsheets may "call" each other. The "Master" spreadsheet contains a Subsheet cell, which forwards all data sent to it to the "Helper" spreadsheet. This data arrives into the Helper via an Argument cell. When the computation is complete, the result is sent back via a Result cell.

The subsheet mechanism relies on three cells, namely the *SubSheet cell*, the *Argument cell* and the *Result cell*. If one considers the spreadsheet that contains the subsheet cell to be the "master" spreadsheet, and the other spreadsheet as the "helper" spreadsheet, then the use of these cells can be summarised as follows (see also Figure 43):

- The "helper" spreadsheet has an "Argument" cell, which collects the arguments that the function implemented by the spreadsheet takes.
- The "helper" spreadsheet also has a single "Result" cell, which exposes the final result calculated by the spreadsheet.
- The "master" spreadsheet has a "Subsheet" cell which behaves much like a function evaluation cell, but which "calls" the helper spreadsheet.

The concept is quite simple, and best demonstrated with an example. In Figure 44, spreadsheet "mul.v" implements a multiplication function. Data (in the form of `(a b)` pairs) enters the spreadsheet via cell A1, the pairs get multiplied by cell B1 and the resulting numbers leave via cell C1. As far as spreadsheet "mul.v" is concerned, the pairs generated by cell B1 are entering cell C1, being processed, and the results are displayed by cell D1.

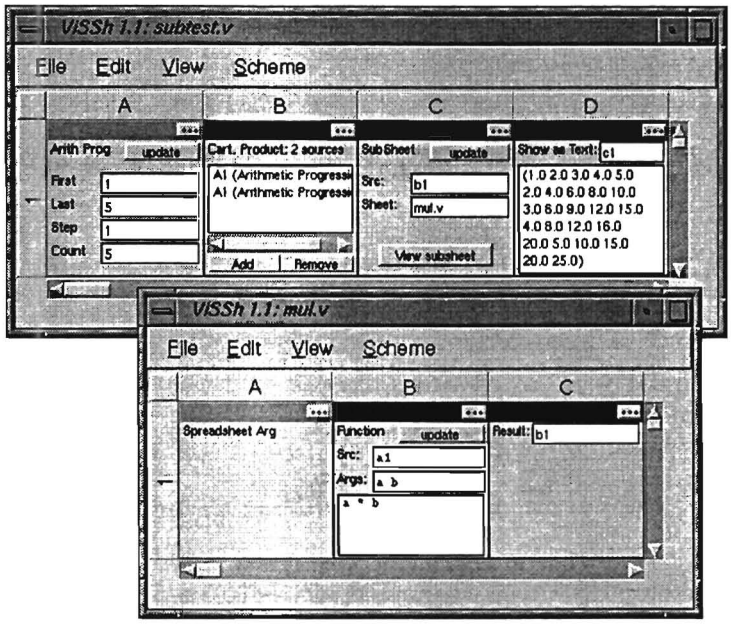


Figure 44: This is a very simple example of how to use subsheets. Cell C1 of spreadsheet "subtest.v" calls spreadsheet "mul.v"; Data flows into "mul.v" via cell A1, and leaves via cell C1.

A.3.2 Scheme Libraries

Sometimes (as in the example in Figure 44), subsheets are overkill. This happens when all users want to do is to add new functions to those already available for use in functional cells. This can be easily achieved by the use of the scheme programming language. Functions can be added in two ways: adding a transient function or a function library.

Adding a Transient Function

Transient functions are functions which are usable from anywhere in a spreadsheet, but which vanish when the spreadsheet is closed. These functions are typically created *ad hoc*. Adding transient functions is done via the **Scheme→Console** menu. When this menu option is selected, an interactive

scheme interpreter window pops up, and new functions may be defined there. See Figure 45 for an example, where a new function (`veclen`) has been defined in the scheme console and used in a functional cell. Note that the scope of the scheme console is the current spreadsheet, i.e. if there are two spreadsheet windows open when a function is defined, then that function will only be available to the spreadsheet it was defined in.

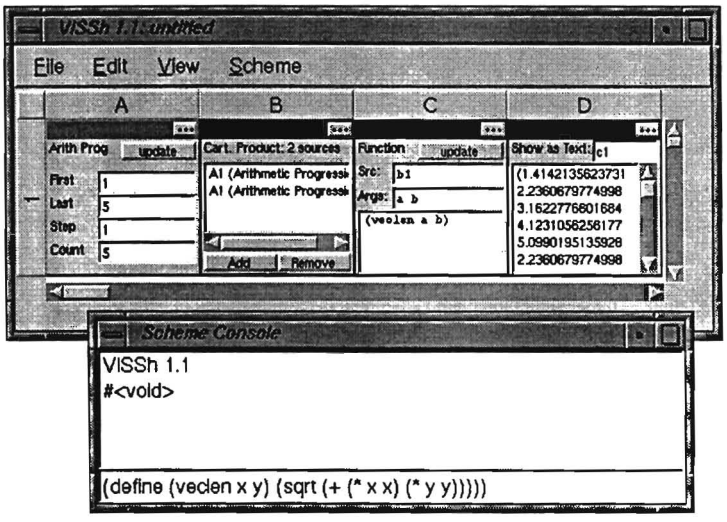


Figure 45: This is an example of how to add a transient function to a spreadsheet. The function is defined in the interactive Scheme interpreter window, and can afterwards be used anywhere in the spreadsheet.

Adding a Scheme Function Library

If the user wishes to have a function that is always available, then that function should be put in a function library, which will be loaded by ViSSh. There are two ways a function library written in Scheme can be added: automatically and explicitly.

To add a function library automatically, simply write it in a file called `autoexec.scm`. This file, which resides in the same directory as the ViSSh executable, is loaded automatically when a new spreadsheet is created or loaded. This means that any functions in `autoexec.scm` are automatically available.

Adding a function explicitly is done via the **Scheme→Libraries** menu. The window which opens when this menu is selected lists all libraries loaded by the current spreadsheet (explicitly loaded functions, like transient functions, are visible only to the current spreadsheet). This list of functions will be saved with the spreadsheet, therefore the user does not have to remember which

libraries a particular spreadsheet needs. To add a new library to the list, simply type its name in the text entry control at the bottom of the window.

A.4 Cell Reference

A.4.1 Dataflow

SubSheet

This node calls another spreadsheet.

It behaves much like a Function node. The Src field must contain the address of a cell that generates argument lists, while the Sheet field must contain the filename of the spreadsheet that will calculate the result. The arguments will be sent to the Argument nodes of the called spreadsheet, and the Result node of that spreadsheet will collect the result, which will be echoed by this node.

SubSheet Arguments

This node is used to pass arguments to a subsheet.

SubSheet Result

This node is used to return values to a caller spreadsheet.

In the small editbox you must enter the name of the spreadsheet cell that is providing the values being returned.

A.4.2 Data Sources

Text Entry

This node is used to enter text.

Unfortunately at this time only US-ASCII characters can be entered, in a future version unicode will be supported.

Arithmetic Progression

This node generates arithmetic progressions.

The numbers are generated as a list, with the values in the fields having the following meanings:

- The first field sets the first number in the progression.
- The last field sets the last number of the progression.

- The step field sets the distance between any two adjacent numbers in the series. If this number is zero, then only the number in the first field will be output
- The count field indicates how many numbers must be generated, in other words, the length of the generated list.

The First and Last fields may be indirectly specified by writing them as cell names in [square brackets]. In that case, the actual values will be first elements of the lists generated by the named nodes.

Geometric Progression

This node generates geometric progressions.

The numbers are generated as a list, with the values in the fields having the following meanings:

- The first field sets the first number in the progression.
- The last field sets the last number of the progression.
- The step field sets the distance between any two adjacent numbers in the series. If this number is zero, then only the number in the first field will be output.
- The count field indicates how many numbers must be generated, in other words, the length of the generated list.

Coordinate Entry

This node is used to enter 3D coordinates.

The coords are stored in a list of 3 elements, namely X, Y and Z.

Rotation Entry

This node is used to enter rotations.

The rots are stored as quaternions, in a list of 4 elements.

Data File

This node reads data off a file.

The file must be made of ASCII characters, consisting of a number of fields and records. The field separator is a comma, and the record separator a newline character. In other words, the file consists of lines containing comma-delimited items. Note that all records must contain the same number of fields. Fields are allowed to be empty.

To retrieve data from the file, set the Filename field to the name of the data file and the Recnum Src field to the address of a node that generates record numbers. Note that records are numbered starting from 1. The output of this node will be a list of lists, each sublist containing the actual record.

Data File Info

This node reads a data file's layout.

The file must be made of ASCII characters, consisting of a number of fields and records. The field separator is a comma, and the record separator a newline character. In other words, the file consists of lines containing comma-delimited items. Note that all records must contain the same number of fields. Fields are allowed to be empty.

To retrieve data from the file, set the Filename field to the name of the data file. The output of this node will be a single `(numrecords numfields)` pair.

A.4.3 Functional Nodes

Cartesian Product

This node generates Cartesian products of lists.

It takes an element at a time from each of the specified lists and combines the elements into a list of lists, with each of the sublists containing one of all the possible permutations obtainable from the elements of the source lists.

As an example, consider two lists, `(1 2 3)` and `(a b)`. The Cartesian product of these lists will be `((1 a) (1 b) (2 a) (2 b) (3 a) (3 b))`.

To add a source, select the source node in the spreadsheet grid and click on the Add button. To remove a source, select it from the list of sources and click on the Remove button.

Running Total

This node adds up list elements.

The n^{th} element of the list returned by this node is the sum of nodes 1 to n of the source node. For example, if the source list contains `(1 2 3 4)`, then this node's output will be `(1 3 6 10)`. This is useful since it may be used to implement integration.

Note that if the source contains non-numeric data, the results are undefined.

Argument List

This node creates argument lists.

It takes an element at a time from each of the specified lists and combines the elements into a list of lists, with each of the sublists containing one element from each of the source lists. These generated lists can be used to pass arguments to function nodes.

As an example, consider two lists, `(1 2 3)` and `(a b c)`. The result of collating these two lists will be `((1 a) (2 b) (3 c))`.

If the source lists are not of the same length, then the generated list will be as long as the shortest of all the source lists.

To add a source, select the source node in the spreadsheet grid and click on the Add button. To remove a source, select it from the list of sources and click on the Remove button.

Function

This node evaluates functions passed to it.

The Src node must pass a list of lists, with each sublist containing the arguments to the function, in order. The Args field must contain the names of the arguments to the function, separated by spaces. Finally, the larger field must be filled in with the body of the function itself, either a simple mathematical expression (e.g., `m*x + c`) or a Scheme function.

Note that the Src field may be left blank, in which case the result will be a list containing only one item, the evaluated constant expression.

Data Replicator

This node is used to replicate data.

It returns a list containing count copies of the positionth element of the source node (0 based). e.g., if the source node outputs the list `(1 2 3 4)`, then if count is 5 and position is 2, then the output will be `(3 3 3 3 3)`.

List Length

This node returns the length of the list passed to it.

List Sum

This node returns the total obtained after adding all elements of a list.

List Average

This node returns the average obtained after adding all elements of a list and dividing by the number of items.

List Selection

This node is used to retrieve single items from a list.

The List Src field contains the name of the cell that generates the source list. The Index Src field contains the name of a cell which generates a single number, which is used as an index into the source list. This cell returns the indexed value from the source list. This node is useful in association with the 3D Pick node.

String Concatenation

This node is used to concatenate strings.

The Left String and Right String fields must contains the names of cells generating lists of values. These values will be converted into strings and concatenated, with the output list being as long as the shortest of the input lists, e.g., if the inputs are `("a" 10 20)` and `("b" "c")`, then the output will be `("ab" "10c")`. If the Add Space checkbox is on, then a space will be added between the left and right strings.

Function Evaluator

This cell evaluates expressions.

This works in a similar way to the Function cell, but the actual function used is passed from another cell. The Func Src field must contain the address of a cell that provides a list of functions. These may be scheme expressions, e.g., `(+ x y)` or infix expressions, in which case they must be strings, e.g., `"x + y"`. The Func Index field must contain the address of a cell which generates a single number, which will be the index of the function used for the calculation. This index is zero-based. The Data Src field specifies the address of the cell supplying the arguments, while the Args field supplies the names of the arguments, separated by spaces.

A.4.4 Geometry Generators

3D Ball

This node constructs balls for display by a Render Node.

The cell specified in the Radius Src field provides a list of numbers, each of which is used as the radius for its corresponding ball. Note that the number of generated balls will be the same as the length of the radius list.

3D Line Set

This node is used to generate line sets.

The Coord Src field must contain the address of a cell that outputs lists of 3D coordinates, and the Length Src field must contain the address of a cell that generates a list of numbers, each of which is the number of points from the first list that the corresponding line is made of. The number of lines that will be generated equals the length of the second list. Each of the generated lines will get its coordinates from the first list, starting at the corresponding offset.

3D Box

This node constructs boxes for display by a Render Node.

It should be passed a list of lists, each of the sublists containing a `(width height depth)` triplet containing the dimensions of each successive box.

3D Surface

This node is used to generate 3D surfaces.

The Coord Src field must contain the address of a cell that outputs lists of 3D coordinates, and the Info Src field must contain the address of a cell that generates a single pair of the form (rows cols), where rows and cols are the number of rows and columns that the generated surface will consist of. The generated surface will get its coordinates from the first list. The PCol Src field is optional, and if filled in it must contain the name of a cell that provides (R G B) triplets for each point in the generated grid. These will override whatever colour was set using Material cells.

3D Text

This node constructs 3D text for display by a Render Node.

The cell specified in the String Src field provides a list of strings, each of which is used as the text to be used to build the corresponding 3D text object.

A.4.5 Transformers

3D Translation

This node is used to translate 3D objects.

The Geom Src field must contain the address of a cell that outputs 3D Geometry, such as a Ball node, and the Trans Src field must contain the address of a cell that generates a list of (dx dy dz) triples. Note that the output list will be as long as the shorter of the two source lists.

3D Rotation

This node is used to rotate 3D objects.

The Geom Src field must contain the address of a cell that outputs 3D Geometry, such as a Ball node, and the Rot Src field must contain the address of a cell that generates a list of `(i j k a)` tuples (a quaternion). Note that the output list will be as long as the shorter of the two source lists.

3D Scaling

This node is used to scale 3D objects.

The Geom Src field must contain the address of a cell that outputs 3D Geometry, such as a Ball node, and the Scale Src field must contain the address of a cell that generates a list of `(sx sy sz)` triples. Note that the output list will be as long as the shorter of the two source lists.

Group

This node groups several 3D objects together, so they can be manipulated as a whole.

Material

This node is used to set the material of 3D objects.

The Geom Src field must contain the address of a cell that outputs 3D Geometry, such as a Ball node, and the Col Src field must contain the name of a cell that generates a list of (red green blue) triples ($0 <= c <= 1$). The Trans Src field must contain the name of a cell that generates transparency values, ranging between 0 and 1. Either of the Col Src or Trans Src fields may be left blank. Note that the output list will be as long as the shorter of the three source lists.

A.4.6 Output

Text Display

This node displays values passed to it as text.

In the small edit box you must enter a cell reference, e.g. A3 and the contents of that cell will be displayed in a textual form. Note that some cell nodes do not have any textual representation, and in that case `#<undefined>` will be displayed. If there is a circular reference somewhere in the spreadsheet, then `#<circref>` will be displayed.

3D Renderer

This node displays 3D objects passed to it in an interactive window.

In the small edit box you must enter a cell reference, e.g. A3 and the contents of that cell will be displayed in an interactive 3D window. Note that some cell nodes do not have any polygonal representation, and in that case nothing will be displayed. This is also the case If there is a circular reference somewhere in the spreadsheet.

A.4.7 3D Interaction

1D Dragger

This node constructs 1D dragger interactors for display by a Render Node.

It should be passed a list of lists, each of the sublists containing the initial translation of each dragger.

3D Pick

This node is used to pick 3D objects from a 3D renderer.

Enter the name of the render node you wish to interact with in the Viewer field and the geometry node you're interacting with in the Select field. When the user clicks on an item, the index number of the selected item will be returned.

A.4.8 Animation

Timer

This node is used to animate parts of a spreadsheet. Enter the name of a spreadsheet cell into the Src field, and a delay in milliseconds in the Period field. The timer cell will read in the list generated by the "Src" spreadsheet cell, and output each item after the given number of milliseconds has elapsed. The One-shot checkbox determines whether this action repeats itself indefinitely or stops when the end of the list is reached. The Active checkbox can be used to turn the timer cell on or off.

Appendix B

Extending ViSSh

B.1 Introduction

Although one of the main advantages of ViSSh is it's flexibility, this flexibility comes at a cost, namely speed of execution. Internally, ViSSh uses a Scheme [26] interpreter for all its calculations. Although this is fairly fast (it precompiles expressions upon editing, generating compact p-code which is interpreted at evaluation time), it is not as fast as native code. A compromise between speed and flexibility is possible, however: extending the actual spreadsheet language by adding more primitives. This has the advantage of speed of evaluation (for the function being implemented by the new primitive), while not sacrificing flexibility. The downside is that the new nodes must be compiled into the ViSSh binary.

B.2 The Anatomy of ViSSh

ViSSh has been written in C++ [58], following the ANSI standard as much as possible, given current (1998-1999) compiler technology. In particular, namespace support is not yet trustworthy so no use is made of it. TrollTech's *Qt* user interface toolkit [60] (version 1.44) was used to build the GUI. Extensive use was made of the C++ Standard Template Library (STL), although for the sake of compatibility the standard `string` class (from the standard header file `<string>`) was not used. Instead, *Qt*'s `QString` class (from the *Qt* header file `<qstring.h>`) was used for most places where dynamic string manipulation was desired (where maximum performance was needed, plain arrays of `char` were used). The Scheme interpreter used as the main computation engine is the MZScheme library [51], while the 3D geometry manipulation and rendering engine is Open Inventor [64].

B.2.1 The Lazy List Mechanism

In order to implement lazy evaluation (which is not a part of Scheme), the lists that are returned by Nodes are not MZScheme lists, (i.e., these lists are not `Scheme_Objects`). Instead, retrieval of a list generated by a Node is done in two steps. Firstly, the `listlen()` member function of the node is called. This returns the size of the list that is "stored" in that node. Then, for each item that must be retrieved, the `result()` member function is called (the details of how to write the `result()` function for a new node are explained in Section B.4). This member function takes as argument the index number of the list element, and returns a `Scheme_Object *` which points to the actual scheme object that is the desired list item. List indices are zero-based. If no valid item can be returned (either because the node has no data at the time, or the list index is out of range), then the special scheme object `scheme_undefined` will be returned.

B.3 Adding a new Node

Adding a new type of node is a fairly simple operation. One does it in 4 steps:

1. adding an entry to `nodeid.h`
2. writing the code for the node
3. telling class Node about the new node
4. updating the makefile and compiling

These steps are more fully explained below:

B.3.1 Adding an entry to `nodeid.h`

The file `"nodeid.h"` is the central database for numeric node identifiers. Each node type must have a unique id. In order to simplify this process, and in order to promote backwards compatibility, nodes are classified into node groups, each of which can hold up to 65535 node types. Node groups define the rough characteristics of a node. For example, there are data source nodes, such as numeric entry, functional nodes, such as function nodes, etc.

Node id's are 32 bit signed integers. At the time of writing (1999), in most C++ compilers this is generally a `long int`. All valid node id's are positive and greater than zero. The most significant 16 bits define the node's group, while the least significant define the node's position within its

group. For example, take the list length node. Its node ID is `0x00030006`, which means that it is a functional node (node group 3) and furthermore it is the sixth type of functional node.

Adding a new id to the database is simple: just come up with a relevant node id, as described above (remember that it **must** be a unique id), then add it to the file with the others in its group.

B.3.2 Writing the code for the node

All nodes must be subclasses of `class Node`, and the macro `Q_OBJECT` must be defined as the first thing in the class (look at `class NumberNode` in the header file `"numbernode.h"`). Section B.4 describes in more detail the member functions of `class Node`, and explains the virtual functions which need to be reimplemented.

If your node somehow generates or modifies a 3D object (e.g., the Ball node or the Rotate node), then the node's origin must be encoded into the Inventor `SoNode` returned by the `visual()` member function. This is done by simply calling the `tagSoNode()` method of `class Node`, passing it the `SoNode` to be returned and the list position as arguments (e.g., `tagSoNode(newnode,pos);`).

When you are building the user interface to the spreadsheet cell, remember that there is a pain constant associated with user interface design: if the programmer does not feel any pain while working on it, the users are going to when they are using it. The corollary is that pain and anguish on the part of the user interface designer usually translates into an ulcer-free user.

B.3.3 Registering the new node with `class Node`

This is quite simple, and is done entirely inside the file `"node.cc"`:

- First add a new entry for the node in the array called `allnodes`. This array is defined near the top of the file. The first item is the text that will appear in the *Node Palette* to describe that node, and the second the node's id as defined in the file `"nodeid.h"`. The third item is the name of the *xpm* file that will be used as the icon for the new node in the *Broad Overview* and *Cell Dependencies* windows. These images should be drawn as black on a white background and **must** be 32 by 32 pixels in size. All icon files are stored in the `icons/` subdirectory. An example entry would be `{"My test node",TESTNODE_ID,"testnode.xpm"}`. Remember that the last entry in this array **must always** read `{ NULL,-1,NULL }`.

- Then add an entry in the big switch statement in `Node::create()`. This should be a no-brainer.

Don't forget to `#include` your node's main header file at the top of `"node.cc"`!

B.3.4 Updating the makefile and compiling

This is quite simple. There are 3 steps:

1. First add the name of the source file(s) to the list called `SOURCES`.
2. Then add to the list `SRCMOC` the name of the header file that includes the main node definition, modified to end in `".moc.cc"`, e.g., if your header file is called `"mynode.h"` then you'd add `"mynode.moc.cc"`. This file is needed by the *Qt Meta Object Compiler*. If you forget to do this you will get linker errors, usually about a symbol called `_vtbl`.
3. Type `make` at the shell prompt.

B.4 class Node

This is the base class for all nodes. It has several virtual functions, most of which which must be reimplemented. The functions do the following:

- `rebuildDeps()` is called when the spreadsheet is likely to have changed enough that the cell's dependencies need to be rebuilt. This usually means when cells have been added or deleted, or a cell is made to examine other cells, e.g., a FuncNode. This member function is optional, but if you reimplement it you **must** call `Node::rebuildDeps()`.
- `recalculate()` is called when a node that this node depends on has changed. Unless this node is a data sink (usually this means a render node of some type) or you're caching a value that comes in from a node this one depends on, this function need not be reimplemented, since `class Node`'s implementation will properly propagate the call. However, for a render node this should be reimplemented to refresh the display.

  NB: If you're reimplementing this, one of the things your new function must do is to call `Node::recalculate()` so the propagation can take place. Typically this will be the last line of the reimplemented function.
- `isSink()` should return `TRUE` if the node is a data sink (usually this means that the node displays data) and `FALSE` otherwise. Since the default implementation returns `FALSE`, this need only be reimplemented if the new node is a data sink.

- `hasVisual()` should return `TRUE` if the node has a visual aspect (i.e., if it can be displayed in a 3D view) and `FALSE` otherwise. Since the default implementation returns `FALSE`, this need only be reimplemented if the new node has a visual aspect. If `Node::hasVisual()` returns `TRUE`, then `Node::visual()` should be reimplemented.

- `result()` is called when a node that this one depends on is recalculated, and takes as an argument which list element must be recalculated. If this argument is larger than the number of elements in the entire result set, then `scheme_undefined` should be returned. This, together with `Node::recalculate()`, forms the heart of the lazy evaluator. Look at Section B.2.1 to see what the behaviour of this member function should be.

- `visual()` is similar to `Node::result()`, only it returns the visual component of a list element. In this current implementation, all visual components are little Inventor trees, the root of which is an `SoSeparator`. This member function returns that `SoSeparator`. Note that, as an optimization, `Node::visual()` can return `NULL` if the object would not be visible (e.g., a BallNode with a radius of 0). Look at BallNode for a simple example.

- `listlen()` is called when a node that this one depends on wants to find out the size of the result list.

- `serialize()` is called when a node is to be saved or copied to the clipboard, and returns a textual description of the node, in a QString. Section B.5 describes the format this function should encode the node as.

- `deserialize()` is called when a node is loaded or pasted from the clipboard. It takes a QString describing the node (created by `Node::serialize()`) and sets the internal state of the node to be that described by the string. Section B.5 describes the format this function should expect the data to be in.

- `bigView()` is a virtual public slot function that is called when a user clicks on the small button on the top right of a cell node. It should show the big view of the node if it's hidden, and hide it if it's visible. The big view should contain at least online help for the node, and possibly a roomier user interface. For example look at the function node - there's not much room in the actual node, so the big view provides a much larger QMultiLineEdit for users to enter their functions. Note that the large and small views always remain synchronised. This is important.

B.5 The Node Serialization Data Format

When a node is serialized, it is converted from an object suitable for efficient manipulation inside ViSSh to an object suitable for storing, either in the clipboard or in a disk file. Deserializing an object's representation reverses this process, yielding a node usable by ViSSh.

In order to ensure portability between different ports of ViSSh to different architectures, serialized objects will consist entirely of printable characters. In the case of numbers, these should be stored as their human-readable form (i.e., ASCII numerals), with as much precision as possible. If binary data needs to be stored, some mechanism must be devised to convert this into printable characters and back. Possibilities include *uuencode* and base-64. It does not matter which format is used, as long as it is used consistently.

Each node is stored as a record, which is delimited by braces (`"{"` and `"}"`). Inside each record there will be a set of strings of characters, separated by blanks. Should any string need to contain blanks, this string must be delimited by pipes (`"|"`), e.g., `"|this has blanks|"`. If any pipes appear in the text, these must appear in pairs, e.g., `"a|b"` must encode as `"a||b"`. Since neither scheme nor any human language use pipes, these should appear very seldom indeed. Functions to do conversion to and from this format are in the file `"misc_util.cc"`.

The very first string in the record will be a hexadecimal, 32 bit integer describing the type of node. All 8 hex digits must be present, including any leading 0's. After this will come a set of strings, as described above, which will depend on the type of node.

As an example, consider a text entry node containing the text "hello world." This would be serialized as `"{ 00010001 |hello world| }"`.

Bibliography

[1] AgentSheets, Inc. AgentSheets home page. Online HTML document, 1998. "http://www.agentsheets.com/".

[2] J. Backus. Can programming be liberated from the von neumann's style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8), August 1978. (ACM Turing Award Lecture).

[3] A. F. Blackwell and T. R. G. Green. Investment of attention as an analytic approach to cognitive dimensions. In T. Green, R. Abdullah, and P. Brna, editors, *Collected Papers of the* 11^{th} *Annual Workshop of the Psychology of Programming Interest Group (PPIG-11)*, 1999.

[4] Borland International. *Quattro Pro Users' Guide, Version 3.0.* Borland International, 1991.

[5] T. Brus, M. van Eekelen, M. van Leer, M. Plasmeijer, and H. Barendregt. Clean — a language for functional graph rewriting. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '87)*. Springer-Verlag, 1987.

[6] M. Burnett, J. Atwood, and Z. Welch. Implementing level 4 liveness in declarative visual programming languages. In *1998 IEEE Symposium on Visual Languages*, pages 126–133, Halifax, Nova Scotia, Canada, September 1–4 1998.

[7] M. Burnett and H. Gottfired. Graphical definitions: Expanding spreadsheet languages through direct manipulation and gestures. *ACM Transactions on Computer-Human Interaction*, 5(1):1–33, March 1998.

[8] Casady & Greene, Inc. Spreadsheet 2000. Online HTML document, 1997. "http://www.emer.com/s2k/".

[9] E. H. Chi, P. Barry, J. Riedl, and J. Konstan. A spreadsheet approach to information visualization. In *Information Visualization Symposium*, 1997.

[10] E. H. Chi, P. Barry, J. Riedl, and J. Konstan. Principles for information visualization spreadsheets. *IEEE Computer Graphics and Applications*, 18(4):30–38, July/August 1998.

[11] W. de Hoon. Designing a spreadsheet in a pure functional graph rewriting language. Master's thesis, University of Nijmegen, 1993.

[12] W. de Hoon, L. Rutten, and M. van Eekelen. Implementing a functional spreadsheet in clean. *Journal of Functional Programming*, 5(3):383–414, July 1995.

[13] Dept. of Geology, University of Cape Town. The kaapvaal craton data centre. Online HTML document, 1999. "http://kaapbase.geo.uct.ac.za/".

[14] D. Dyer. A dataflow toolkit for visualisation. *IEEE Computer Graphics and Applications*, 10(60), 1990.

[15] G. Edwards. The design of a second generation visualisation environment. In *Proceedings of Computer Graphics 1991*. Blenheim Online, 1991.

[16] U. Eriksson. Scheme in a grid. Online HTML document, 1999. "http://siag.nu/siag/".

[17] T. Green and A. Blackwell. Cognitive Dimensions of Information Artefacts: a tutorial. Available online, 1998. "http://www.ndirect.co.uk/~thomas.green/workStuff/Papers/".

[18] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7:131–174, 1996.

[19] J. G. Hays and M. M. Burnett. A guided tour of forms/3. Technical Report 95-60-6, Department of Computer Science, Oregon State University, June 1995. Revised January 1997.

[20] S. E. Hudson. User interface specification using an enhanced spreadsheet model. *ACM Transactions on Graphics*, 13(3):209–239, July 1994.

[21] J. Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

[22] T. Igarashi, J. D. Mackinlay, B.-W. Chang, and P. T. Zellweger. Fluid visualization of spreadsheet structures. In *Proceedings of the IEEE Symposium on Visual Languages*, 1998.

[23] T. Isakowitz, S. Schocken, and H. C. Lucas. Toward a logical/physical theory of spreadsheet modeling. *ACM Transactions on Information Systems*, 13(1):1–37, 1995.

[24] G. S. Iwerks and H. Samet. The spatial spreadsheet. In *Visual Information and Information Systems, 3rd International Conference, VISUAL '99*, 1999.

[25] J. A. Johnson, B. A. Nardi, C. L. Zarmer, and J. R. Miller. Ace: building interactive graphical applications. *Communications of the ACM*, 1993.

[26] R. Kelsey, W. Clinger, and J. Rees (Editors). Revised (5) report on the algorithmic language scheme. Online HTML document, 1992. `"http://swissnet.ai.mit.edu/~jaffer/r5rs_toc.html"`.

[27] B. Kernighan and D. Ritchie. *The C Programming Language (2nd edition)*. Prentice Hall, 1988.

[28] Khoral Reserach, Inc. Khoros pro 2000 student edition. Online HTML document, 1997. `"http://www.khoral.com/"`.

[29] F. Kriwaczec. Logicalc: a PROLOG spreadsheet. In J. E. Hayes, D. Michie, and J. Richards, editors, *Machine Intelligence 11 — Towards an Automated Logic of Human Thought*. Clarendon Press, 1985.

[30] M. Levoy. Spreadsheets for images. In *Computer Graphics Proceedings*, Annual Conference Series, pages 139–146. ACM SIGGRAPH, July 1994.

[31] C. Lewis. Nopumpg: Creating interactive graphics with spreadsheet machinery. In E. P. Glinert, editor, *Visual Programming Environments: Paradigms and Systems*, pages 526–546. IEEE Computer Society Press, Los Alamitos, 1990.

[32] Lotus Development Corporation. *Lotus 1-2-3 Reference Manual, Release 2*. Lotus Development Corporation, 1985.

[33] B. Lucas, G. D. Abram, N. S. Collins, D. A. Epstein, D. L. Gresh, and K. P. McAuliffe. An architecture for a scientific visualisation system. In *Proceedings of Visualisation '92*. IEEE Computer Society Press, 1992.

[34] F. T. Marchese. Teaching computer graphics with spreadsheets. In *Educators Program*, pages 84–87. ACM SIGGRAPH, 1998.

[35] Microsoft Corporation. *Microsoft Excel Users' Guide, Version 4.0.* Microsoft Corporation, 1992.

[36] Microsoft Corporation. Visual basic for applications start page. Online HTML document, 1998. "http://msdn.microsoft.com/vba/".

[37] F. Modugno, T. R. G. Green, and B. A. Myers. Visual programming in a visual domain: A case study of cognitive dimensions. In G. Cockton, S. W. Draper, and G. R. S. Weir, editors, *People and Computers IX*, Proc. of HCI'94, Glasgow, pages 91–108. Cambridge University Press, 1994.

[38] B. S. Murray and E. A. Edmonds. Flexibility in interface development. In *Computers and Digital Techniques*, volume 141, pages 93–98. IEE, March 1994.

[39] B. Myers. Graphical techniques in a spreadsheet for specifying user interfaces. In *Proceedings CHI'91: Conference on Human Factors in Computing Systems*, pages 243–249, New Orleans, LA, April 28–May 2 1991. ACM.

[40] B. A. Nardi and J. R. Miller. The spreadsheet interface: A basis for end user programming. Technical Report HPL-90-08, Software Technology Laboratory, HP Laboratories, March 1990.

[41] B. A. Nardi and J. R. Miller. Spreadsheets' support for shared work. Technical Report HPL-90-13, Software Technology Laboratory, HP Laboratories, March 1990.

[42] B. A. Nardi and J. R. Miller. Twinkling lights and nested loops: distributed problem solving and spreadsheet development. *International Journal of Man-Machine Studies*, 34:161–184, 1991.

[43] E. Neuwirth. Spreadsheet structures as a model for proving combinatorial identities. Technical Report 93-07, University of Vienna, Institute of Statistics, Operations Research, and Computer Methods, 1993.

[44] E. Neuwirth. Visualizing formal and structural relationships with spreadsheets. Technical Report 94-22, University of Vienna, Institute of Statistics, Operations Research, and Computer Methods, 1994.

[45] J. K. Ousterhout. Tcl: An embeddable command language. In *Proceedings of the 1990 Winter USENIX Conference*, pages 133–146, 1990.

[46] J. Paine. Model master: an object-oriented spreadsheet front end. Technical report, Department of Experimental Psychology, Oxford University, June 1998.

[47] K. Palaniappan, M. Manyin, and A. Hasler. Interactive image spreadsheet user's manual. Online HTML document, 1998. "http://meru.cecs.missouri.edu/mvl/iiss/users_manual/".

[48] R. Panko. Hitting the wall: Errors in developing and debugging a "simple" spreadsheet model. In *Proceedings of the Twenty-Ninth Hawaii International Conference on System Sciences*, January 1996.

[49] K. W. Piersol. Object oriented spreadsheets: The analytic spreadsheet package. In *OOPSLA '86 Proceedings*, 1986.

[50] R. Rao and S. K. Card. The table lens: Merging graphical and symbolic representations in an interactive focus+context visualization for tabular information. In *Human Factors in Computing Systems, CHI'94*, pages 318–322, 1994.

[51] Rice University. The mzscheme home page. Online HTML document, 1998. "http://www.cs.rice.edu/CS/PLT/packages/mzscheme/".

[52] B. Ronen, M. A. Palley, and H. C. Lucas, Jr. Spreadsheet analysis and design. *Communications of the ACM*, 32(1):84–93, January 1989.

[53] D. Schreiner, editor. *OpenGL Reference Manual, Third Edition: The Official Reference Document to OpenGL, Version 1.2*. Addison-Wesley, 1999.

[54] W. J. Schroeder, K. M. Martin, and W. E. Lorensen. *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*. Prentice-Hall, 1996.

[55] H. Shiozawa, K.-i. Okada, and Y. Matsushita. 3d interactive visualization for inter-cell dependencies of spreadsheets. In *Proceedings of the 1999 IEEE Symposium on Information Visualization*, 1999.

[56] N. C. Shu. *Visual Programming*. Van Nostrand Reinhold, 1988.

[57] M. Spenke, C. Beilken, and T. Berlage. Focus: The interactive table for product comparison and selection. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, 1996.

[58] B. Stroustrup. *The C++ Programming Language (3rd edition).* Addison-Wesley, 1997.

[59] H. Thimbleby. *User Interface Design.* ACM Press, 1990.

[60] Troll Tech AS. The Qt user interface toolkit. Online HTML document, 1998. "http://www.troll.no/products/qt.html".

[61] C. Upson, T. Faulhaber, Jr., D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam. The application visualisation system: A computational environment for scientific visualisation. *IEEE Computer Graphics and Applications*, 9(20), 1989.

[62] J. J. van Wijk and R. van Liere. Hyperslice. In G. M. Nielson and D. Bergeron, editors, *Proceedings of IEEE Visualisation '93 Conference.* IEEE Computer Society Press, 1993.

[63] P. Wadler. The essence of functional programming. In 19th *Annual Symposium on Principles of Programming Languages*, 1992.

[64] J. Wernecke. *The Inventor mentor : programming object-oriented 3D graphics with Open Inventor, release 2.* Addison-Wesley, 1994.

[65] N. Wilde and C. Lewis. Spreadsheet-based interactive graphics: from prototype to tool. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI90).* ACM SIGCHI, April 1990.

[66] N. P. Wilde. Using cognitive dimensions in the classroom as a discussion tool for visual language design. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI96).* ACM SIGCHI, April 1996. (Short Paper).

[67] M. Yazdani and L. Ford. Reducing the cognitive requirements of visual programming. In M. Burnett and W. Citrin, editors, *IEEE Symposium on Visual Languages.* IEEE Computer Society, September 1996.

[68] P. G. Zimbardo. *Psychology and Life.* Harper Collins, thirteenth edition, 1992.