Spreadsheets for Images

Marc Levoy
Computer Science Department
Stanford University

Abstract

We describe a data visualization system based on spreadsheets. Cells in our spreadsheet contain graphical objects such as images, volumes, or movies. Cells may also contain widgets such as buttons, sliders, or curve editors. Objects are displayed in miniature inside each cell. Formulas for cells are written in a general-purpose programming language (Tcl) augmented with operators for array manipulation, image processing, and rendering.

Compared to flow chart visualization systems, spreadsheets are more expressive, more scalable, and easier to program. Compared to conventional numerical spreadsheets, spreadsheets for images pose several unique design problems: larger formulas, longer computation times, and more complicated intercell dependencies. In response to these problems, we have extended the spreadsheet paradigm in three ways: formulas can display their results anywhere in the spreadsheet, cells can be selectively disabled, and multiple cells can be edited at once. We discuss these extensions and their implications, and we also point out some unexpected uses for our spreadsheets: as a visual database browser, as a graphical user interface builder, as a smart clipboard for the desktop, and as a presentation tool.

CR Categories: I.4.0 [Image Processing]: General — *Image processing software*; I.3.6 [Computer Graphics]: Methodology and Techniques — *Interaction techniques*, *Languages*; D.3.2 [Programming Languages]: Language Classifications — *Data-flow languages*

Additional keywords: data visualization, user interfaces, flow charts, visual programming languages, spreadsheets

1. Introduction

The majority of commercially available image processing and data visualization systems employ a flow chart paradigm. Users select processing modules from a menu and wire them together using a mouse. Although elegant in principle, flow charts are limited in expressiveness and scalability. Useful programming constructs like procedure calls and variable substitution cannot be conveniently expressed in these systems. Flow charts spend their screen real estate on operators and their interconnections, which becomes uninteresting once the flow chart has been specified, and they run out of screen space if the chart exceeds a few dozen operators. Flow charts also provide no convenient mechanism for managing multiple datasets.

We propose an alternative paradigm based on spreadsheets. Broadly speaking, a spreadsheet is a system for specifying constraints among cells arranged in a grid. Cells may contain a constant data value or a formula that evaluates to a data value. Formulas may reference the value of other cells, but they may not alter the value of other cells. Formulas are typically written in a simple, interpreted language. Examples of spreadsheet systems are Microsoft's Excel, Lotus's 1-2-3, and Borland's Quattro.

We have implemented a spreadsheet for images (henceforth denoted SI) in which we extend the notion of a data value to include graphical objects such as images. These objects are displayed in miniature inside each cell. Double clicking on a cell brings up the full-size object. Cells may also contain interactive widgets. Manipulating a widget modifies the data associated with the cell. If formulas in other cells reference the modified cell, they are recomputed as well.

Formulas in our spreadsheet are written in Tcl, a general-purpose programming language that provides variables, assignment statements, procedures, and a full complement of control structures. The formula for a cell can range from a one-line expression to an entire program. To support editing of such formulas, SI is intimately tied to Emacs, a popular, customizable text editor. Double clicking on a cell brings up an Emacs window devoted to that cell.

Compared to flow chart systems, the presence of an embedded formula language makes SI more expressive. The infinite grid of the spreadsheet, together with the ability to resize cells, gives SI better scalability. SI also spends its screen space on operands rather than operators, which are usually more interesting to the user. Finally, because spreadsheets are two-dimensional, they provide a natural mechanism for applying multiple operators to multiple datasets.

Compared to conventional numerical spreadsheets, SI offers three important extensions. Firstly, a formula in SI can display its result anywhere in the spreadsheet. This allows users to intermix functional and imperative programming styles, simplifying many common tasks. Secondly, SI allows cells to be selectively disabled. This allows users to work on one part of the spreadsheet at a time, a useful feature in the face of long cell computation times. Thirdly, SI allows multiple cells to be edited at once and fired as a group. This simplifies development of complicated spreadsheets. These three extensions complicate the dependency analysis and the cell firing algorithm, as we shall see.

The remainder of the paper is organized as follows. Section 2 presents our reasons for choosing Tcl as our formula language, and it describes how Tcl and SI fit together. Section 3 describes the logical structure and command set of SI. The remaining sections describe SI's implementation, our experiences with SI, comparisons with other systems, and the future of SI.

Address: Center for Integrated Systems
Stanford University
Stanford, CA 94305-4070
Email: levoy@cs.stanford.edu
Web: http://www-graphics.stanford.edu

Register manipulation	`load, store, display, openwindow*, closewindow*, popwindow, pushwindow`
Spreadsheet services	`loadsheet, storesheet, winsize*, titleheight, view-pixel*`
Cell manipulation	`cellsize*, view-cell, cut*, copy*, paste*, delete*, enable*, disable*`
Data structure queries	`regexists, queryreg, codereg, codecell`

Figure 1: Commands of the SI kernel. Starred commands are also available using a point and click interface.

2. Tcl as a formula language

From a conceptual point of view, the choice of a formula language is unimportant. We envision SI as a kit of parts in which the language is a replaceable module. For our prototype, we sought a language that was powerful, easy to type, and interpreted rather than compiled (for interactivity). Our choice was Tcl (Toolkit Command Language) [3]. Tcl consists of an application-independent embeddable command interpreter, a set of built-in commands for manipulating variables, strings, lists, and files, and a set of C-callable interface routines for adding additional commands. Examples of Tcl code are scattered throughout this paper.

From the user's point of view, Tcl's advantages are that it is easy to type (like UNIX shell commands) and that it provides a variety of control structures and substitution mechanisms (like the UNIX shell but better). From the implementors' point of view, Tcl's advantages are its small code size, its fast execution (fast enough to use for mouse event loops), and its simple interface to C — procedure calls with string arguments.

Tcl has one further advantage: it is the basis for Tk [4], an X11 toolkit similar to Xt. Tk provides a base set of graphics and text-oriented widgets, a mechanism for defining new widgets, and a simplified interface between user applications and the X window system. For SGI users, Tk replaces the window management and event handling services that are present in GL but are missing in OpenGL. As the Tcl/Tk user communities expand, we expect to see Tk widget sets for 3D graphics and image processing.

Tcl appears in two places in SI. Firstly, it is the language in which formulas are written. Secondly, the SI program provides a Tcl command prompt. Users may invoke all of the functionality of SI, including functions normally driven by the mouse, by entering commands at this prompt. This capability allows users to record and play back interactive sessions, to customize SI from an initialization script, and to perform many other useful tasks.

3. The structure and commands of SI

SI consists of a kernel and one or more standalone application packages. The kernel manages memory, displays the spreadsheet, and contains the firing algorithm. The application packages create and manipulate data registers and are responsible for defining Tk-compatible widgets to display the registers they create. This modular design reflects one of Ousterhout's goals for Tcl: systems composed of compact, reusable parts.

Register creation	`scalar, vector, scanline, image, volume`
Display widgets	`button, slider, label, plot, imageviewer, cineviewer`
Register manipulation	`copy, extract, insert, promote, slice, delete`
Pixel operations	`add, subtract, multiply, divide, mod, over, and, or, makeramp, ramp, shift`
Spatial operations	`rotate, convolve, scale, displace, warp, makedisplacement`
3D occupancy grids	`readabekas, deinterlace, profile, opinion, occupancy`

Figure 2: Commands of a prototype image processing package, including the commands for processing 3D occupancy grids that were used to generate figure 5.

In this section, we take a tour through the logical structure and command set of the SI kernel. Our examples include commands from a simple image processing application package. The command set of the SI kernel is listed in figure 1. The command set of our image processing package is listed in figure 2.

3.1. Registers

The basic unit of storage in SI is called a register. A register is a named allocation of memory. Registers may contain anything: images, volumes, geometry, etc. The SI kernel controls the allocation and deallocation of registers, but the kernel knows nothing about the contents of a register. The contents and interpretation of registers is determined by those application package commands that know how to manipulate them.

Commands in SI generally consist of a command name followed by options and one or more arguments. The argument list for most commands includes the name of one or more registers. For example,

```
rotate -Bspline myreg Y 45 newreg
```

rotates a volume register named `myreg` around its Y-axis by 45 degrees. The command uses a cubic B-spline as its resampling filter, and places its result into a register named `newreg`.

To minimize the number of type coercions a user must perform, most commands accept a variety of register types, performing conversions, applying defaults, or ignoring arguments as appropriate. One very important default is that if the name of the output register (`newreg` above) is omitted, SI will make up a name. To make this form useful, commands that produce a register as output return a string result giving the name of the output register. The register produced by such a command can be used as the input to another command using Tcl's command substitution mechanism:

```
rotate -Bspline [load head.mri] Y 45 newreg
```

In this formula, the `load` command executes first, generating an arbitrary name for its output register, e.g. `Reg123`. The `rotate` executes next, with arguments `Reg123 Y 45 newreg`. The register produced by the `load` command is never seen by the user and is unimportant. It is deleted automatically by SI when the formula is modified or when the cell is deleted.

3.2. Display widgets

The contents of registers are by themselves undisplayable. The second building block in SI is a display widget. It is a view of a register. Some types of registers may have more than one widget that knows how to display them; others may have none. Such a register would need to be converted to a displayable type in order to view it.

Display widgets are associated with registers using a widget command. For example,

```
cineviewer -rocking [load head.mri]
```

loads a volume into a register, then opens a window on the workstation screen that contains an instance of the cine viewer widget. This widget contains image subwindows and interactive controls for viewing slices of a volume as a flipbook animation. The `-rocking` option specifies that the animation should alternate directions rather than circling from the last frame back to the first frame.

3.3. Cells

The third building block in SI is a cell. In addition to their usual appearance, all display widgets in SI know how to draw themselves in miniature inside a spreadsheet cell. Miniature versions of widgets may be live, meaning that they respond to mouse events just like the full-size widget, or dead, in which case double clicking on the miniature version brings up the full-size widget.

Display widgets are associated with cells by adding a cell name argument to the widget command. To display a miniature version of the cine viewer widget in cell a1, we type

```
cineviewer -rocking [load head.mri] a1
```
(1)

So far, we have assumed that all formulas are entered at the SI program prompt. If a formula is instead typed into an Emacs window that is associated with a particular cell, the cell name argument may be omitted:

a1: `cineviewer -rocking [load head.mri]`

We use the notation "a1:" (typeset in Times Roman) to signify that the formula that follows (typeset in Courier) is contained in the cell a1. The "a1:" does not appear in the cell. Every type of register has a default display widget. For volumes, it is the cine viewer. Therefore, the formula in cell a1 could be further simplified to read

a1: `load head.mri`

Executing this formula would cause the specified file to be loaded into a volume register, and a miniature version of the cine viewer to be displayed in the cell. If the formula contains more than one command separated by newlines, only the register returned by the last command executed will be displayed in the cell.

In the common case, the user doesn't need to think about registers or display widgets when writing formulas. The vast majority of formulas will look like this last example. The key to providing both brevity and flexibility in the formula language lies in the liberal use of defaults.

3.4. Chaining formulas together

Cell names may be used in any context in which a register name is valid. This allows us to reference the data in a cell by either its register name or its cell name. Here is a simple three-cell spreadsheet:

a1: `load alps.rgb`
b1: `rotate a1 45`
c1: `ramp b1 [makeramp {{0 255} {255 0}}]`

The first command loads an image into cell a1. A miniature version of the image is displayed in the cell. The second command rotates the image by 45 degrees and displays the result in cell b1. The third command inverts the pixel values in the rotated image, displaying its result in cell c1. The `makeramp` command accepts a Tcl list of coordinate pairs and returns a Tcl list of coordinates piecewise linearly interpolated from the specified coordinates. In this example, the command would return the Tcl list `{{0 255} {1 254} {2 253}...}}`. This list becomes an input argument to the `ramp` command, which modifies the image from cell b1.

3.5. Ways to reference a cell

As in numerical spreadsheets, references to cells can be relative or absolute. `b1` is a relative reference. If a formula containing this reference were moved down one row using cut and paste, the reference would be changed to `b2`. If the formula is being edited in an Emacs window at the time it is relocated, SI sends the updated text to Emacs. In contrast, the notation `/b1`, `b/1`, or `/b/1` forces the column, row, or both coordinates to be absolute, respectively. Absolute references are not modified if the cells are relocated.

SI also supports arithmetic calculations on cell references. For example, `a1+2+3` references the cell two rows down and three columns right from the base address `a1`. If a formula containing this reference were moved down one row, the base address `a1` would be changed to `a2`, and the reference would stay correct. Note that this construction is similar to Excel's `OFFSET(A1,2,3)`, but is somewhat easier to type.

Finally, all of Tcl's substitution mechanisms can be applied to cell references. Thus, `a1+$i+$j` references the cell whose row and column offsets from `a1` are given by the Tcl variables `i` and `j` respectively. Similarly, `a1+[foo]+[bar]` references the cell whose offsets are the values returned by the Tcl commands `foo` and `bar`. Finally, every occupied cell in the spreadsheet has associated with it a Tcl command that returns the contents of the register displayed in that cell. Thus, `a1+[b1]+[b2]` references the cell whose offsets are the values contained in cells `b1` and `b2`.

3.6. Active widgets

In addition to being live or dead, widgets may be passive, meaning that they only display their underlying registers, or active, meaning that they both display and modify their underlying registers. For example:

a1: `load alps.rgb`
b1: `slider`
b2: `rotate a1 [b1]`

The command `slider` in cell b1 is a widget command, Since it is invoked without arguments in this example (compare to the `cineviewer` command in example (1)), a default scalar integer register is created, and the slider displays the contents of that register. The operand `[b1]` in cell b2 invokes the command `b1`, which returns the current value of the slider in cell b1, and the `rotate` command rotates the image in cell a1 by this amount. Since the formula in cell b2 depends on cell b1, moving the slider causes the rotation to be recomputed.

The spreadsheet for this example is shown in figure 3. The slider widget is really Tk's "scale" widget. The options visible on the `slider` command in the figure are options defined by Tk for its `scale` command. In addition to these Tk-defined options, most active widgets accept a `-[no]continuous` option. Specifying `-continuous` means that the widget will fire its descendents repeatedly (as fast as possible) until the mouse button is released. If the slider in the previous example were so defined, dragging the slider bar back and forth would cause the image to rotate back and forth. To reduce computational delays if cell b2 were the beginning of a long sequence of operations, active widgets also accept a `-[no]firedescendents` option. Specifying `-nofiredescendents` means that the widget will fire only its immediate children as long as the mouse button is down. When the button is released, the widget's other descendents will be fired.

3.7. Control structures

SI supports all of the control structures in Tcl, including `if`, `while`, `for`, `foreach`, and `case`. Of particular interest are the looping commands. Loops in SI take one of three general forms:

Single-cell loops. The easiest way to code a loop is entirely within one cell using the Tcl `for` command:

```
a1:  load alps.reg temp
     for {set i 0} {$i <= 90} {incr i 30}
              {rotate temp $i a1}
```

This formula will step the alpine pasture image through four rotational positions, each of which will appear briefly in cell a1.

Multi-cell for-loop. If the user has already built a sequence of processing steps and decides retrospectively to iterate one or more parameters of the sequence over a range of values, this can be done without reworking the entire spreadsheet by inserting one additional cell at the beginning of the loop to trigger it:

```
a2:  for {set i 0} {$i <= 90} {incr i 30}
              {byte $i a2; fire b2}         (2)

b1:  load alps.reg
b2:  rotate b1 [a2]
b3:  ramp b2 [makeramp {{0 0} {255 100}}]
```

The original spreadsheet consisted of cells b1 through b3. Cell a2 has been added to control the loop. The `byte` command creates a scalar byte register and displays it in a2 using a Tk label widget. The `fire` command executes cell b2 as a subroutine. When cell b2 and its descendents (b3 in this example) have finished executing, control is returned to a2, which increments `i` and loops.

Multi-cell while-loop. If the user wishes to predicate loop termination on a value computed by the loop body, two cells are required to control the loop:

```
a1:  load alps.reg                          (3)
c1:  byte 3
b2:  convolve -box [c1] [c1] a1
c3:  if {[max [gradient b2]] > 50}
         {byte [expr [c1] + 1] c1}
```

In this example, the `byte` command in cell c1 initializes the loop. The `if` command in cell c3 evaluates a data object computed by the loop body and conditionally modifies cell c1, on which the loop body depends. The loop body will thereby be reexecuted repeatedly until the condition becomes false. In this example, cell a1 is blurred by a box filter of increasing width (starting at 3x3), stopping when the maximum gradient magnitude in the image drops below 50.

Figure 3: Slider widget being used to control a rotation. Cell b2 rotates the image in cell a1 by the angle specified on the slider in cell b1. Each time the slider is moved, cell b2 (and its descendents, if it had any) are recomputed.

4. Selected implementation details

4.1. The user interface

SI is implemented in C, C++, Tcl/Tk, and Emacs Lisp. These depend on UNIX and the X window system but are otherwise platform independent. Some of the widgets currently depend on GL — the nonportable version of Silicon Graphics's graphics library — but these will shortly be converted to OpenGL, a platform-independent library.

The unique characteristics of SI pose several challenging user interface design problems. Firstly, our cells are larger than those in numerical spreadsheets, so fewer of them are displayed at once. To make navigation easier, we provide an accelerated scrolling tool and the ability to quickly change the size of all cells. (Individual cells cannot be resized, as this would destroy the regularity of the spreadsheet grid.)

Secondly, our longer formulas and powerful language semantics lead to more complicated intercell dependencies than in numerical spreadsheets. To keep users from getting lost, the formula for each cell is displayed inside the cell. Long formulas can optionally be decimated to fit (see figures 5 and 6). Although the decimated text is not legible, its overall structure is clearly visible. To clarify intercell dependencies, the dependency graph can be displayed as an overlay (see figures 4 and 5).

Thirdly, our cells take longer to compute than cells in numerical spreadsheets — several minutes in extreme cases. To keep the spreadsheet visually consistent during long computations, cells that depend on modified cells are grayed out (in Macintosh style) to indicate that they are out of date. As each cell fires, it is highlighted to provide feedback of its progress. The mouse is alive during cell computations and can be used to navigate through the spreadsheet or abort an errant computation.

As a further response to long cell computation times, SI allows cells to be selectively disabled, allowing the user to work on one part of the spreadsheet at a time. In addition, the user can select a group of cells to edit simultaneously in Emacs, and then fire the entire group at once with a single keystroke. While most of the features described above are cosmetic, these last two profoundly affect the firing algorithm, which is discussed in the next section.

4.2. Managing dependencies

The dependency relationships in SI are represented by a directed acyclic graph having two types of nodes, formulas and objects. Objects consist of cell names, register names, Tcl variables, and Tcl procedures. When a formula consumes an object (e.g. specifies a cell as input, invokes a Tcl procedure, etc.), this is represented in the graph by a directed edge from the object to the formula. When a formula produces an object (e.g. specifies a cell as output, defines a Tcl procedure, etc.), this is represented by a directed edge from the formula to the object.

Following user modification of one or more formulas or objects, the dependency graph is traversed as described in Appendix A, and the modified formulas and their descendents are recomputed. The time required to perform a dependency analysis is usually several orders of magnitude smaller than the time required to execute a formula or decimate an image for display, so we do not discuss it further here.

SI's firing algorithm differs from the firing algorithms found in conventional spreadsheets in two ways. Firstly, the ability to specify objects as outputs in formulas forces us to distinguish formulas from objects in the dependency graph and gives rise to producer edges (that is, edges from formulas to objects). Surprisingly, this additional flexibility does not complicate the dependency analysis in any substantive way.

The second difference arises from our ability to selectively disable and reenable cells and to edit several formulas at once. Conventional spreadsheets do not allow either action. As a result, our firing algorithm begins with a queue of edited or reenabled formulas which need to be executed. To further complicate matters, Tcl allows conditionally executed commands and substitutions in command operands (see section 3.5). This prevents us from lexically scanning a formula in advance of execution to determine the set of objects it will consume or produce. Without this information, it is impossible to determine which formula on the queue should be executed first.

This is a standard problem in operating system design, and there are standard solutions to it. The solution used in SI (and described in the appendix) is to execute modified formulas in arbitrary order, requeueing them if they consume undefined or invalid objects. If the dependency relationships are indeed acyclic, this algorithm is guaranteed to find a valid firing order. The presence of a cycle leads to a condition called livelock, and it will be detected by our firing algorithm. This solution can lead to wasted computation if a formula contains a long computation followed by a reference to an invalid object, requiring the formula to be requeued and recomputed from scratch later, but such cases are rare. In practice, formulas usually make their consumer references early, so the time lost to requeuing is negligible, and the user sees only the final firing order. We are currently investigating other solutions, including blocking a formula's execution until the invalid object is available (in this case, cycles lead to a condition called deadlock), lexically scanning the cell to resolve as many references as possible, or allowing formulas to declare a set of objects they might consume or produce when executed.

5. Experience and examples

Our experience with SI has been limited but positive. Although its image processing package offers only rudimentary functionality, we have used it in several research projects (see figure 5). We have also found some unexpected uses for SI, such as summarizing research results for colleagues and giving public presentations (see figure 6). Sometimes, we use it simply as a smart clipboard for storing images on our desktop, like the Macintosh clipboard but more powerful. Other plausible applications for SI are as a database browser, as an exposure sheet for computer animation, or as a video postproduction planner.

The ability to specify outputs in formulas makes SI very different from a conventional spreadsheet, so it is worthwhile to consider how this additional power might be used. Consider the spreadsheet shown in figure 4. The user in this example began by building a pipeline for classifying 3D medical datasets. Puzzled by the classified volume displayed in cell d2, the user added the following diagnostic code to the formula in that cell:

```
set means ""
for {set z 0} {$z < [d2 zlen]} {incr z} {
   lappend means [mean [slice d2 $z]]
}
set order [lorder $means]
for {set i 0} {$i < 5} {incr i} {
   slice d2 b4+0+$i [lindex $order $i]
}
```

This code creates a Tcl list containing the mean pixel intensity for each of the `[d2 zlen]` slices in the classified volume, calls `lorder`, a Tcl proc (defined elsewhere in the formula) to generate a second list containing slice numbers in order of decreasing mean slice pixel intensity, and displays the brightest 5 slices in cells b4 through f4. Still puzzled, the user created formulas in cells f5 and g5 to analyze one of these slices.

In this example, the ability to display results anywhere in the spreadsheet made it easy to insert the visual equivalent of a printf statement into an existing formula — without having to reorganize the spreadsheet. The ability to define and reference local variables made it easy to write the for-loops needed to sort the slices — conventional spreadsheets don't allow local variables in formulas. The flexibility to use an imperative programming style (that is, with assignment statements) in cell d2 to produce cell f4 and a functional programming style in cell g5 to consume cell f4 extends but does not break the spreadsheet paradigm — on the contrary, it seems very natural. This example would have been difficult to write without this flexibility.

6. Comparisons and discussion

Spreadsheets for images are not a new idea. Piersol's ASP package [6], a spreadsheet program based on the Smalltalk-80 object-oriented programming environment, anticipates many of the features of SI. Cells are allowed to contain any sort of object, including images and other spreadsheets, and formulas are written in Smalltalk — a general purpose programming language. In keeping with the conventional spreadsheet paradigm, formulas in cells in ASP are not permitted to alter the value of other cells.

Palaniappan's IISS environment [5] combines a custom Mathematica-like formula language, Mark Overmars's Forms UI toolkit, and the Khoros image processing library. IISS appears to allow assignment statements, but a complete definition of their language and firing algorithm has not yet been published.

A key feature of SI is its ability to intermix functional and imperative programming (see the example in section 5).

Figure 4: A simple classification pipeline for a 3D medical dataset. A slice chosen using the slider in cell b1 is classified according to the transfer function selected in cell c1. The unclassified and classified slices are displayed in cells b2 and c2, respectively. When the user presses the button in cell d1, the entire volume is classified and displayed in cell d2. In addition, the brightest 5 slices are displayed in row 4. The display of these classified slices is controlled by the formula in cell d2 (see section 5). An Emacs editor window is also visible; it is currently editing the contents of the formula in cell d2.

Interestingly, conventional numerical spreadsheets also offer imperative programming, usually in the form of a second, hidden command language that is more flexible than the cell formula language. To write an imperative program using Excel Version 4.0 [2], the user creates an auxiliary spreadsheet called a macro sheet that has special properties. Formulas in a macro sheet can assign values to any cell in the macro sheet using the `SET.VALUE()` function. This notation is not permitted in the main spreadsheet. The advantage of a two-language design is that it presents a simple programming model to the novice computer user. The disadvantage is that the jump in complexity from spreadsheet programming to macro sheet programming is large.

The currently dominant paradigm for visualizing image data is flow charts, so it is worthwhile comparing them to SI. The earliest system to combine a graph-based execution model with a visual programming interface was Paul Haeberli's ConMan [1]. Currently popular flow-chart visualization packages include AVS, Explorer, apE, Khoros, IBM's Data Explorer, PV-Wave, Wavefront's Data Visualizer, FIELDVIEW from Intelligent Light, VoxelView, and many others. SGI's Explorer [7] is perhaps the most highly developed of these packages, so we base our comparisons on it. Three major factors can be identified:

Expressiveness. The "repeat" and "while" modules of Explorer approximate the `for` and `while` loops of Tcl. Explorer contains no modules, however, that evaluate conditionals or perform substitutions (unless the user writes a custom module).

Scalability. The "micro" form of a module icon in Explorer measures 116 x 40 pixels; 30 modules and their associated wiring makes for a crowded window. Modules may be coalesced into a single icon, but the user must perform this reduction. Cells in SI can be resized down to 12 x 17 pixels simply by dragging the window frame, allowing up to 6000 cells to be displayed at once. Although cells are unreadable at that size, such a view makes it easy to navigate through a large spreadsheet.

Customization. Explorer provides extensive support for writing custom modules, but the jump in complexity from visual programming to module programming in C or Fortran is nontrivial. In SI, the formula language is also the customization language. The transition from novice user to expert user is therefore smooth. To facilitate rapid module prototyping, Explorer also offers an interpreted language called Shape. Its power is greater than Tcl because it directly supports array manipulations, but its interface to the flow chart via the encapsulating "LatFunction" module is somewhat cumbersome.

7. Status and future work

The kernel of SI is complete and relatively stable. Our efforts are now focused on building application packages. The image processing package used in these examples needs more commands and a richer library of widgets. We plan to soon add a volume visualization package, a polygon mesh package, and a surface fitting package.

The most critical issue for the future of SI is performance. Spreadsheets offer a natural mechanism not present in flow charts — and not yet exploited in SI — for controlling computational expense; images need only be computed at a resolution commensurate with the size of the cells in which they are displayed. In the early stages of a data exploration, miniature images suffice, and computations should be fast. If the user stretches the spreadsheet, images get bigger and computations slow down. If the user double clicks on a cell, that cell is recalculated at full resolution. Many image processing operators lend themselves in an obvious way to such computation shedding; spatial warps can be subsampled; frequency domain operators can be windowed; polygonal meshes can be retiled using fewer polygons. Our goal is to make these optimizations transparent to the user.

Another area for future development is the formula language. Tcl is not an ideal solution in many respects. It offers only one datatype — strings. Because there are no numerical datatypes, arithmetic expressions are cumbersome to write, as the examples in this paper demonstrate. Tcl also does not support multidimensional arrays. All manipulation of arrays (and hence images) in SI must be done through C-language commands. Finally, Tcl does not have the speed of a compiled language like C. We often find ourselves prototyping a computation in Tcl, then rewriting it in a combination of Tcl and C. Alternatives to Tcl include Lisp, a C or C++ interpreter (several now exist), or a new language that combines the simplicity of Tcl with the power of an array manipulation language like Mathematica or MATLAB.

To summarize, SI combines the power of a data analysis language, the interactivity of a flow chart visualizer, and the extemporaneous qualities of a spreadsheet. While the power of SI seems useful and easily manageable in examples such as the one shown in figure 4, SI is nevertheless a general programming environment, and it is possible to create confusing programs using it. In particular, the flow of control in the while-loop in section 3.7 is not obvious. In general, the presence of conditionally executed commands and substitutions in command operands means that the reference patterns of formulas in SI are dynamic; cycles can appear and disappear during spreadsheet recomputation. The ability of SI to display the dependency graph and to detect cycles helps, but it is not foolproof. We are continuing to refine the design of SI as we search for a data analysis paradigm that is simple enough to keep a novice out of trouble yet powerful enough to satisfy the needs of a scientist/programmer.

8. Acknowledgements

Discussions with David Heeger, Richard Frank, Bob Brown, and Robert Skinner were useful in the early stages of the project. I wish to particularly acknowledge many fruitful discussions with Philippe Lacroute. This research was supported by the NSF under contract CCR-9157767 and by Software Publishing.

9. References

[1] Haeberli, Paul, ''ConMan: A Visual Programming Language for Interactive Graphics,'' *Computer Graphics (Proc. SIGGRAPH)*, Vol. 22, No. 4, Atlanta, Georgia, August, 1988, pp. 103-111.

[2] Microsoft Corporation, *Excel User's Guide 2*, Microsoft Corporation, Document Number XL26297-1092, 1992.

[3] Ousterhout, John K., ''Tcl: An Embeddable Command Language,'' *Proc. 1990 Winter USENIX Conference*.

[4] Ousterhout, John K., ''An X11 Toolkit Based on the Tcl Language,'' *Proc. 1991 Winter USENIX Conference*.

[5] Palaniappan, K., Hasler, A.F., Manyin, M., ''Exploratory Analysis of Satellite Data Using the Interactive Image Spreadsheet (IISS) Environment,'' Preprint volume of the *9th Internation Conference on Interactive Information and Processing* Anaheim, California, January, 1993, pp. 145-152.

[6] Piersol, K.W., ''Object Oriented Spreadsheets: The Analytic Spreadsheet Package,'' *Proc. OOPSLA '86*, September, 1986, pp. 385-390.

[7] Silicon Graphics Inc., *IRIS Explorer User's Guide* and *IRIS Explorer Module Writer's Guide*, Silicon Graphics Inc., Document numbers 007-1371-020 and -1369-, 1992-1993.

Appendix A: The firing algorithm

Dependency relationships in SI are represented by by a directed acyclic graph having two types of nodes, formulas and objects, as described in section 4.2. Formula are marked as modified or unmodified, and objects are marked as modified or unmodified, and as valid or invalid. Formulas become modified in one of three ways:

(1) The user changes a formula using the Emacs text editor.

(2) The user adds, deletes, cuts, pastes, or loads a cell.

(3) The user enables for firing a previously disabled cell.

Following user modification of one or more formulas, an execution queue is created and is initialized to the set of modified formulas, arranged in arbitrary order. The firing algorithm then proceeds as follows:

> **Step 1: execute a formula.** Remove a formula *i* from the front of the queue, delete all edges originating or terminating at *i*, and submit it to the Tcl interpreter for execution. For each object *j* consumed (or produced) by formula *i* as it executes, add a directed edge from *j* to *i* (or from *i* to *j*) to the graph. If two producer references point to the same object, a collision has occurred; flag it as an error. If *i* is found to consume an undefined or invalid object (e.g. an empty or invalidated cell, an undefined Tcl procedure, etc.), abort execution and move *i* to the back of the queue. If no formula on the queue can be executed successfully, the spreadsheet contains a cycle; flag it as an error.
>
> **Step 2: invalidate its descendents.** If formula *i* executes successfully, mark as invalid all objects *k* such that there exists a path of length one or more originating from *i* and terminating at *k*. Allow cycles of length two involving a formula and an object. This allows a formula to read/modify/write a register. Cycles of length greater than two are flagged as errors.
>
> **Step 3: fire its direct consumers.** Add to the execution queue all direct consumers of all objects produced by *i*, i.e. all formulas *m* for which the graph now contains a directed edge of length two from *i* to *m* (passing through an object).

Following user modification of an object (e.g. by manipulating an active widget or by resetting a global variable at SI's Tcl command prompt), all formulas that consume (either directly or indirectly) the object are queued for execution using a similar algorithm.

To avoid introducing cycles into the dependency graph, multi-cell loops require the following special treatment. In the for-loop of example (2) (section 3.7), the `fire` command in cell a2 executes cell b2 and its descendents as a subroutine. The subroutine returns when the execution queue is empty, allowing cell a2 to continue execution. This mechanism bypasses the usual dependency analysis, allowing us to omit from the graph a producer edge from the formula in cell a2 to cell b2 and a consumer edge from cell b3 to the formula in cell a2, which would form a cycle. In the while-loop of example (3), the `byte` command in cell c3 conditionally overwrites cell c1, creating an edge in the graph from c3 to c1 and temporarily creating a cycle. This edge is deleted each time c3 begins execution. On the last iteration through the loop, the `byte` command is not executed and no edge is created. Therefore, in the quiescent states that precede and follow execution of the loop, the graph is acyclic.

Figure 5: This spreadsheet depicts the flow of data in a 3D fax machine - a machine we are building in our lab for digitizing the shape and external appearance of physical objects using a laser scanner and precision motion platform. When the button in cell a1 is pressed, four laser reflection image sequences are loaded into cells a2 through a5, respectively. A cine viewer widget associated with each cell displays a frame from that image sequence in miniature in the cell. The slice specified in the slider in cell b1 is then loaded into cells b2 through b5. Two different occupancy grid algorithms are applied to these slices, leading after some intermediate steps to the results shown in cells c5 and i5. Volume renderings of the complete volumetric occupancy grid have been imported from another spreadsheet and are shown in cells c6 and i6. The user has double clicked on cell i6, so its image is also shown full size at the bottom of the screen. The scene is a pile of wooden children's blocks. An Emacs editor window is also visible; it is currently editing the contents of the formula in cell g2.

Figure 6: This spreadsheet was used to give a live classroom demonstration of the effects of image quantization. The original 8-bit images are in cells a3 through a5. The sliders in cells c1 through i1 are used to set the number of bits to which the images appearing in that column are quantized. This example demonstrates how the two-dimensional grid of a spreadsheet lends itself naturally to visualizing multiple operators applied to multiple datasets. The user has double clicked on cells g3 and c5, so they are also shown at full size. This spreadsheet took about 10 minutes to construct.