COMPUTER SYSTEMS LABORATORY

STANFORD UNIVERSITY STANFORD. CA 94305-4055

SPREADSHEETS FOR IMAGES

Marc Levoy

Technical Report: CSL-TR-94-607

February 1994

This research was supported by the National Science Foundation under contract CCR-9157767 and by Software Publishing Corporation.

SPREADSHEETS FOR IMAGES

Marc Levoy

Technical Report: CSL-TR-94-607

Computer Systems Laboratory
Department of Computer Science
Stanford University
Stanford, CA 94305-4070

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 graphical widgets such as buttons, sliders, or movie viewers. Objects are displayed in miniature inside each cell. Formulas for cells are written in a programming language that includes operators for array manipulation, image processing, and rendering. Formulas may also contain control structures, procedure calls, and assignment operators with side effects.

Compared to flow chart visualization systems, spreadsheets are more expressive, more scalable, and easier to program. Compared to numerical spreadsheets, spreadsheets for images pose several unique design problems: larger formulas, longer computation times, and more complicated intercell dependencies. We describe an implementation based on the Tcl programming language and the Tk widget set, and we discuss our solutions to these design problems. 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.

Keywords:
Data visualization, User interfaces, Flow charts, Visual programming languages, Spreadsheets

Spreadsheets for Images

Marc Levoy
Computer Science Department
Stanford University

January, 1993

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 graphical widgets such as buttons, sliders, or movie viewers. Objects are displayed in miniature inside each cell. Formulas for cells are written in a programming language that includes operators for array manipulation, image processing, and rendering. Formulas may also contain control structures, procedure calls, and assignment operators with side effects.

Compared to flow chart visualization systems, spreadsheets are more expressive, more scalable, and easier to program. Compared to numerical spreadsheets, spreadsheets for images pose several unique design problems: larger formulas, longer computation times, and more complicated intercell dependencies. We describe an implementation based on the Tcl programming language and the Tk widget set, and we discuss our solutions to these design problems. 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: 1.4.0 [Image Processing]: General — Image *pro-&sing software*; 1.3.6 [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 run-time variable substitution cannot be conveniently expressed in a flow chart. 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 program size exceeds a few dozen operators. Flow charts also provide no convenient mechanism for managing multiple datasets. As a result, these systems often prove too cumbersome to support the extemporaneous style of data analysis for which they were originally intended.

We propose an alternative paradigm based on spreadsheets. Broadly speaking, **a** spreadsheet is a two-dimensional cellular automaton. Cells may contain a data value or a formula. Formulas compute a value for one cell as a function of the values in other cells. These formulas are typically written in an simple, interpreted language. Examples of spreadsheet systems are Microsoft's Excel, Lotus's l-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. Widgets are a powerful addition to the semantics of a spreadsheet. In a sense, they turn the spreadsheet into a graphical user interface builder.

Formulas in our spreadsheet are written in Tcl, a general-purpose programming language that provides variables, procedures, assignment statements with side effects, and a full complement of control structures. Using such a language, 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. Double clicking on other cells brings up additional Emacs windows, allowing the user to write formulas for several cells at once.

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

The remainder of the paper is organized as follows. Section 2 presents our reasons for using Tcl as the 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.

Kegister manipulation	*load, store, display, undisplay, openwindow*, closewindow*,* popwindow, pushwindow
Spreadsheet services	loadsheet, *storesheet, winsize*, titleheight, view-pixel**
Cell manipulation	*cellsize*, view-cell, 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) [Ousterhout90]. 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. Here is the Tcl code to compute the factorial of 5:

```
set result 1
set i 5
while ($i > 0) {
    set result [expr $result * $i]
    incr i -1
}
```

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 run-time 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 [Ousterhout91], 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. First, it is the language in which formulas are written. Second, 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.

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.

3. The structure and commands of SI

SI consists of a kernel program and one or more standalone application packages. This modular design reflects one of Ousterhout's goals for Tcl; it leads to systems composed of compact, reusable parts. 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, geometry, formulas, etc. SI controls the allocation and deallocation of registers and keeps track of which formula produces and consumes each register, but SI knows nothing about the contents of a register. The contents and interpretation of registers is determined by those commands that know how to manipulate them and those widgets that know how to display them.

Commands 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 Bspline as its resampling filter, and it places its result into a register named **new reg.**

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 registerproducedbythe `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 between forwards and backwards animation rather than circling from the last frame back to the first frame. Other display widgets include buttons for Boolean registers, labels and sliders for scalar registers, plots for vector registers, and image viewers for image registers.

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 they may be dead, meaning that they are for viewing only. If the miniature version of a widget is dead, double clicking on it brings up the full-size widget. For example, the miniature version of a slider widget is live, although its positional resolution is necessarily reduced. The miniature version of a cine viewer is not live because that would require decimating the entire image sequence before it could be animated. However, any time the cine viewer is not animating, its current frame is decimated and displayed in the associated cell.

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 al, we type

```
cineviewer -rocking [load head.mri] al (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`:

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

We use the notation "al:" (typeset in Times Roman) in this paper to signify that the formula that follows (typeset in Courier) is contained in the cell al. The "al:" does not appear in the cell. Every type ofregister has adefaultdisplay widget. For volumes, it is the cine viewer. Therefore, the formula in cell al could be further simplified to read

```
al: 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 several more than one command (separated by newlines or semicolons in accordance with Tcl syntax), only the register returned by the last command executed will be displayed in the cell.

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:

```
al: load alps.rgb
bl: rotate al 45
cl: ramp bl [makeramp {{0 255) (255 0}}]
```

The first command loads an image into cell al. 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 bl. The third command inverts the pixel values in the rotated image, displaying its result in cell cl. 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 bl .

3.5. 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:

```
al: load alps.rgb
bl: slider
b2: rotate al [bl]
```

The command slider in cell bl is a widget command, Since it is invoked without arguments (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 `[ bl ]` in cell b2 invokes a command named bl. 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, the command `bl` returns the value displayed on the slider, and the rotate command rotates the image in cell al by this amount. Since the formula in cell b2 depends on cell bl, 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 `-continuous option`, meaning that they will fire their 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 chain of operations, active widgets also accept a `-nofiredescendents option`. If specified, the widget will fire only its

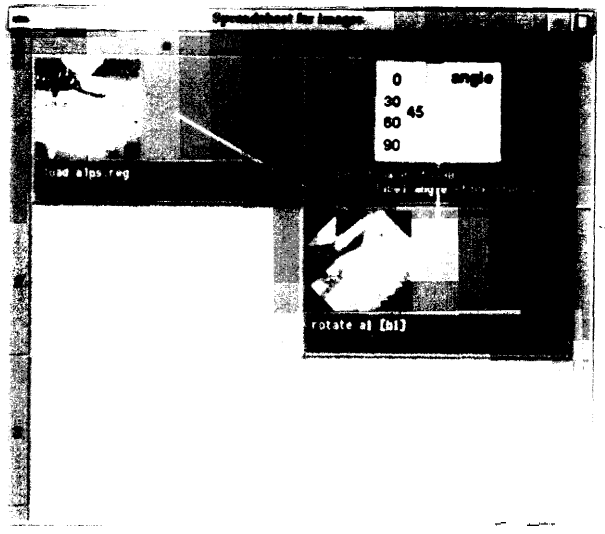

Figure 3: Slider widget **being used** to **control** a rotation. Cell b2 rotates the image in cell al by the **angle** specified on the slider in cell bl. Each time the **slider** is moved, cell b2 (and its **descendents**, if it had any) arc **recomputed.**

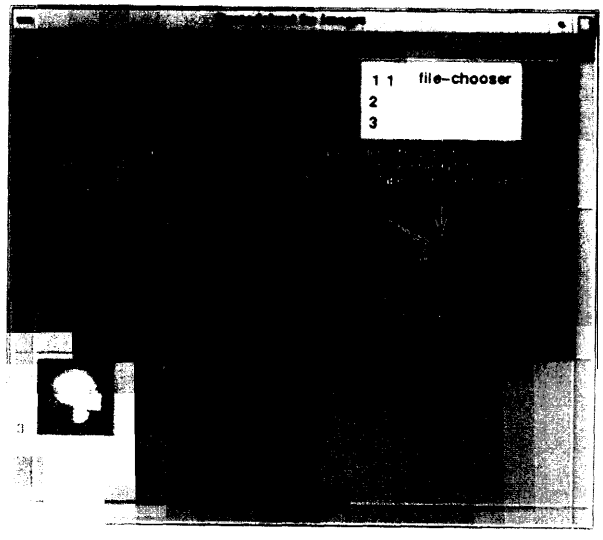

Figure 4: A slider that chooses among three input files. To insure a correct **firing order, cell al declares that it produces cells al, a2, and** a3. Cell **c2 declares that it might consume cells al, a2, or a3. With the slider set as shown, c2 consumes al, hence its arrow is** darker.

immediate children as long as the mouse button is down. When the button is released, the widget's other **descendents** will be fired.

3.6. Functional versus imperative styles

In a purely functional programming language such as ML, a function communicates only through its calling arguments and **return** value. In an imperative programming language such as **C,** a function may assign a value to a variable that outlives the function invocation. In other words, functions in an imperative language may have side effects.

Conventional numerical spreadsheets enforce a functional programming style. The formula associated with a cell computes a value for that cell as a function of the other cells in the spreadsheet, and a formula cannot assign values to cells other than itself. While a functional style is preferred for many tasks, certain programs can be expressed more naturally using an imperative style.

The option in SI of specifying a register or cell name as the output of a command means that SI supports both functional and imperative styles. As an example of when an imperative style is more natural than a functional style, consider the following fragment that splits a color image into its red, green, and blue components:

```
a1: load alps. rgb
b1: extract -comp 1 a1
b2: extract -comp 2 a1
b3: extract -comp 3 a1
```

Here is an imperative formulation of the same program:

```
c1: load alps.rgb temp                    (2)
    foreach i {1 2 3}
        {extract -comp $i temp c$i}
```

In the second formulation, **the load** command places its **output** into a temporary register that is referenced later in the code but never displayed. The extract command places its outputs into cells cl, **c2,** and **c3**. Although both formulations are reasonable in this example, the imperative formulation would prove superior if the loop contained 100 iterations rather than 3, or if several nested loops were present.

In the context of spreadsheet systems, an important advantage of the functional approach is simplicity; one data value per cell, one update rule per cell, and no hidden memory. An important advantage of the imperative approach is ease of editing since the commands making up a program are not scattered in different cells. In the extreme case, the entire program is contained in one cell, and the rest of the spreadsheet serves as an addressable clipboard. Rearranging the visual appearance of an imperative computation involves merely changing a variable in a program, rather than cutting and pasting blocks of cells. Another advantage of the imperative approach is that it allows commands that read, modify, and write their operands. A commonly encountered example of such a command is **insert,** which **might be used for inserting** slices into a volume. A functional program must make a new copy of the volume, which is inefficient.

3.7. Substitutions on operands

There are numerous ways to specify operands in SI. As in numerical spreadsheet systems, references to cells can be relative or absolute. **al** is a relative cell name. If cells al and bl are moved to cells **f3** and g3 using cut and paste, a reference by bl to al will be changed to reference **f3.** 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 **/al, a/l, or /a/l** forces the column, row, or both coordinates to be absolute, respectively. Absolute references are not modified if the cells are relocated.

If an operand includes a **Tcl** variable substitution, it is not known until run time whether the reference is to a cell, and whether it is absolute or relative. If the operand is moved using cut and paste, it cannot be changed to reflect its new location. To

2 + 2 O Norm

support relocatable cell names in the presence of run-time substitutions, we define a command cellexpr that computes arithmetic expressions on cell locations. As an example, the reference `c$i` in example (2) is not relocatable as written. To make it relocatable, we rewrite it as

```
{extract -comp $i temp
    [cellexpr cl + a$i]}
```

The addition of `two cell` names using `cellexpr` cause their row and column addresses to be added. Rows and columns count downwards and rightwards, with `al` being the additive identity. If the formula above were moved from cell cl to cell fl, a rightward shift of 3 columns, the cl will be recognized as a relative cell name and will be changed to `f1`. The resulting formula will place the three color component images into cells fl, f2, and f3. These substitution mechanisms can be combined with Tcl procedures to define powerful spreadsheet macros.

3.8. Static declarations

The flow of data in a conventional numerical spreadsheet (not including its command macro language, if it has one) can be represented by a directed acyclic graph. Each node in this graph is a cell. Each reference by a formula to a cell is a directed edge. When a cell is modified, its formula is scanned, edges are added to or deleted from the graph as appropriate, and the modified cell and its descendents are recomputed. If more than one cell has been modified, as occurs in a paste operation, any cell that has no modified ancestors may be fired first.

The presence in SI of conditionally executed commands and run-time substitutions in formulas means that the set of data objects that will be consumed or produced by a formula cannot be determined in advance of execution. (Doing so would be equivalent to solving the halting problem.) Without this information, it is impossible to determine which cell from a group that has been modified should be fired first.

To resolve this dilemma, we have added two commands to the language, `consumes` and `produces`, which statically declare the set of all objects that a formula *might* consume or produce when it is executed. Formally, these commands allow us to partition a spreadsheet into disjoint sets of cells and to draw a directed acyclic graph whose nodes are these sets. For example:

```
b2: consumes al a2 a3
    copy [cellexpr al + a[b1]]
```

states that cell b2 will be assigned a copy of cell al, a2, or a3, the choice to be determined at run-time by the value of cell bl. In the figure; cell bl contains a slider, which serves to choose among three input files.

The requirement of static declarations in SI restricts its power, but we have not found this requirement objectionable in practice. The ability to declare the set of references a formula might make, rather than the set it will make, makes the requirement palatable. In fact, we find that enforcing a partition of the spreadsheet into producer and consumer regions leads to clearer designs and fewer errors. To further reduce the onus on the user, SI applies the following defaults. If a formula contains no `consumes` statements, it is assumed to consume all cells that can be recognized in a lexical scan of the formula. If a formula contains no `produces` statements, it is assumed to produce only the cell in which the formula appears. These assumptions cover the most common case. In practice, therefore, static declarations are only used in rare special cases.

3.9. 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. **A** loop can be coded entirely within one cell using an imperative programming style:

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

This formula will step the alpine pasture image through four rotational positions.

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

bl: load alps. reg
b2: rotate bl [al]
b3: ramp b2 [makeramp {{0 0} (255 100}}]
```

The original spreadsheet consisted of cells bl 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 immediately. When cell b2 (and its descendents, although there are none 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:

```
al: load head.pix                                   (3)
cl: byte 3
b2: convolve -box [cl] [cl] al
c3: if {[max [gradient b2]] > 50)
       {byte [expr [cl] + 1] cl}
```

In this example, the byte command in cell cl initializes the loop. The `if` command in cell c3 evaluates a data object computed by the loop body and conditionally modifies cell cl, on which the loop body depends. The loop body will be executed repeatedly until the condition is false. In this example, cell al is blurred by a box filter of increasing width, stopping when the maximum gradient magnitude in the image drops below 50.

The ability to construct a loop in SI would appear to severely impact the firing algorithm, since it introduces a cycle into the dependency graph. Loops in SI, however, are always driven by imperative programming constructs, so it is possible to omit the edge that generates a cycle in the graph without affecting the operation of the loop. The details are given in Appendix A.

4. Implementation

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. Our widgets do not rely for their performance on the SGI rendering pipeline and could have been written using X instead of GL. On the other hand, the fast decimation hardware of SGI's RealityEngine offers a natural path for improving the performance of SI.

The SI kernel communicates with its application packages, which run as separate programs, using shared memory and the Tcl inter-process send command. The kernel manages shared memory, displays the spreadsheet, and contains the firing algorithm. Application packages, which have their own embedded Tcl interpreter, are responsible for parsing commands not recognized by the kernel. To assist in this task, SI provides a library of functions for creating and manipulating registers. The contents of registers, however, are understood only by the application packages. Each application package is also responsible for defining a set of Tk-compatible widgets that can display the registers it creates.

Dependencies among formulas and data objects in SI are represented using doubly-linked lists. The firing algorithm is described in detail in Appendix A. The complexity of this algorithm is linear in the number of modified nodes and their descendents. Except for actions that affect all cells such as loading, storing, or resizing, there is no reason to traverse the entire graph. The time required to perform a dependency analysis is usually several orders of magnitude smaller than the time required to compute a formula or decimate an image.

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. Secondly, our cells also 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 manipulate the spreadsheet (as long as it doesn't change it) or to abort an errant computation. Blocks of cells can also be temporarily disabled, allowing the user to work on one part of the spreadsheet at a time. Thirdly, 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 figure 5).

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. If the presentation is given using a live workstation image, impromptu changes can be made in the spreadsheet in response to audience questions. We have successfully used SI in this way to teach graphics concepts to undergraduates (see figure 6).

Sometimes, we use SI 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. Indeed, SI bears some resemblance to several commercially available multimedia authoring tools.

6. Comparisons

The internal structure of SI was inspired by OBVIUS [Heeger91], an image processing environment based on Common Lisp and Emacs. OBVIUS defines "viewables" and "pictures" that are similar to SI's registers and display widgets, but OBVIUS has no notion comparable to SI's cells or spreadsheet. The advantage of using Lisp as the user programming language is that OBVIUS is itself implemented in Lisp, leading to a seamlessly extensible environment. The disadvantages of Lisp are its clumsy syntax, the difficulty of interfacing to C code, and its lack of acceptance in the scientific community.

It is interesting to compare SI to conventional spreadsheet systems that offer command macro languages. Excel's command language, for example, offers many of the imperative programming constructs available in Tcl [Microsoft92]. Programs written in this language, however, stand apart from the spreadsheet and are triggered by events such as button presses. If a macro changes a spreadsheet cell, its descendents are recomputed. In SI, the imperative programming constructs are contained in the cells, and they participate in determining the order in which cells are fired. This approach reduces unnecessary recomputation in many cases and leads to a more self-documenting program

Closest in spirit to SI are the flow chart visualization environments. The earliest system to combine a graph-based execution model with a visual programming interface was Paul Haeberli's ConMan [Haeberli88]. Currently popular packages include AVS, Explorer, apE, Khoros, IBM's Data Explorer, PV-Wave, Wavefront's Data Visualizer, FIELDVIEW from Intelligent Light, VoxelView, and many others. A good survey of these systems can be found in [Earnshaw92]. SGI's Explorer [SGI93] is perhaps the most highly developed of these packages, so it is instructive to compare it to SI. 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 run-time substitutions (unless the user writes a custom module). As a result, the wiring in an Explorer flow chart is static, whereas the reference patterns of formulas in SI are dynamic (modulo the statically declared partition of the spreadsheet into producing and consuming regions as described in section 3.8).

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 grouped together and represented by a single icon, but the user is responsible for performing this reduction. Cells in SI can be resized down to 12 x 17 pixels simply by dragging the window frame. Formulas, images, and widgets are automatically decimated to fit. This allows up to 6000 cells to be displayed at once in a typical workstation window. Although cells are unreadable at that size, such a view makes it easy to navigate through a big 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 large. In a spreadsheet, 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 in some respects because it directly supports array manipulations, but the encapsulating "LatFunction" module still requires compilation, and the language currently lacks robust program development tools.

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 in these investigations is to make optimization nearly transparent to the user, like enabling a compiler option.

Another area for future development is the formula language. Tcl is not an ideal solution in many respects. It offers only one datatype — strings. Interfacing a C routine to a Tcl routine requires converting all numerical arguments to and from string arguments. 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. It offers a new paradigm for interacting with images, and it suggests a new direction for data visualization environments.

8. Acknowledgements

This research was supported by the National Science Foundation under contract CCR-9157767 and by Software Publishing Corporation. Discussions with Richard Frank of Software Publishing and Bob Brown, Robert Skinner, and other members of the Explorer team at Silicon Graphics were useful in the early stages of the project. I wish to particularly acknowledge the many fruitful discussions I had with Philippe Lacroute, who also provided a helpful reading of the manuscript. Michael Halle wrote the Tk GLX widget used in the current implementation.

9. References

[Earnshaw92] Earnshaw, R.A., *An Introductory Guide to Scientific Visualization*, Springer-Verlag, 1993.

[Haeberli88], Haeberli, Paul, "ConMan: A Visual Programming Language for Interactive Graphics," *Computer Graphics (Proc. SZGGRAPH)*, Vol. 22, No. 4, Atlanta, Georgia, August, 1988, pp. 103-l 11.

[Heeger91] Heeger, David, Eero Simoncelli, and Eduardo-Jose Chichilnisky, *OBVIUS: Object-Based Vision and Image Understanding System*, Version 2.2, MIT Media Lab, April, 1993.

[Microsoft92] Microsoft Corporation, *Excel User's Guide 2*, Microsoft Corporation, Document Number XL26297-1092, 1992.

[Ousterhout90] Ousterhout, John K., "Tcl: An Embeddable Command Language," *Proc. 1990 Winter USENIX Conference.*

[Ousterhout91] Ousterhout, John K., "An Xl 1 Toolkit Based on the Tcl Language," *Proc. 1991 Winter USENIX Conference.*

[SGI93] 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

The flow of data in SI can be represented by a directed acyclic graph having two types of nodes, formulas and data objects. A cell consists of a formula and a data object. A formula will be marked as modified or unmodified, and its data object will be marked as modified or unmodified and as valid or invalid. Data objects consist of cells, named registers, Tcl variables, and Tcl procedures. (When a formula specifies a register by cell name, the cell name is entered in the graph, not the made-up name of the underlying register.) Formulas in SI 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, the firing algorithm proceeds as follows: (Error handling has been omitted from this description.)

Step 1: scan modified cells. For each modified formula *i*, delete all edges originating or terminating at *i*. Parse all `consumes` and `produces` commands. If the formula contains no `consumes` command, lexically scan it for recognizable references. If the formula contains no `produces` command, assume that it produces the cell in which the formula appears. For each consumes (or produces) reference by formula *i* to data object *j*, add a directed edge from *j* to *i* (or from *i* to j) to the graph. If two produces references point to the same data object, a collision has occurred; flag it as an error.

Step 2: invalidate descendents. For each modified formula *i* that survived step 1, mark as invalid all data 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 a data object. This allows a formula to read/modify/write a data object. Cycles of length greater than two are flagged as errors.

Step 3: fire cells having valid inputs. For each modified formula *i* that survived step 2, if all edges terminating at *i* originate from data objects marked as valid, then *i* may be executed. Delete all statically declared references by *i* to produced objects, i.e. all edges originating from *i.* For each data object j actually produced by *i* during execution, add a directed edge from *i* to j. When execution completes, fire all immediate consumers of the produced data objects, i.e. resubmit to step 3 all formulas *m* that survived step 2 and for which the graph now contains a directed edge of length two from *i* to *m* (i.e. passing through a data object).

Following user modification of a data object, for example by an active widget, all formulas that consume the object are invalidated and fired using **a** similar algorithm.

To avoid introducing cycles into the dependency graph, loops whose termination is predicated on a value computed by the loop body require the following special treatment. In example 3 (section 3.9), we have deliberately omitted the static declaration produces cl **from the** loop controller in cell c3. Without it, there is no cycle in the dependency graph before iteration begins. When c3 is fired, it conditionally overwrites cl using a byte command, creating an edge connecting the formula in c3 to the scalar register in cl and temporarily generating a cycle in the graph. 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.

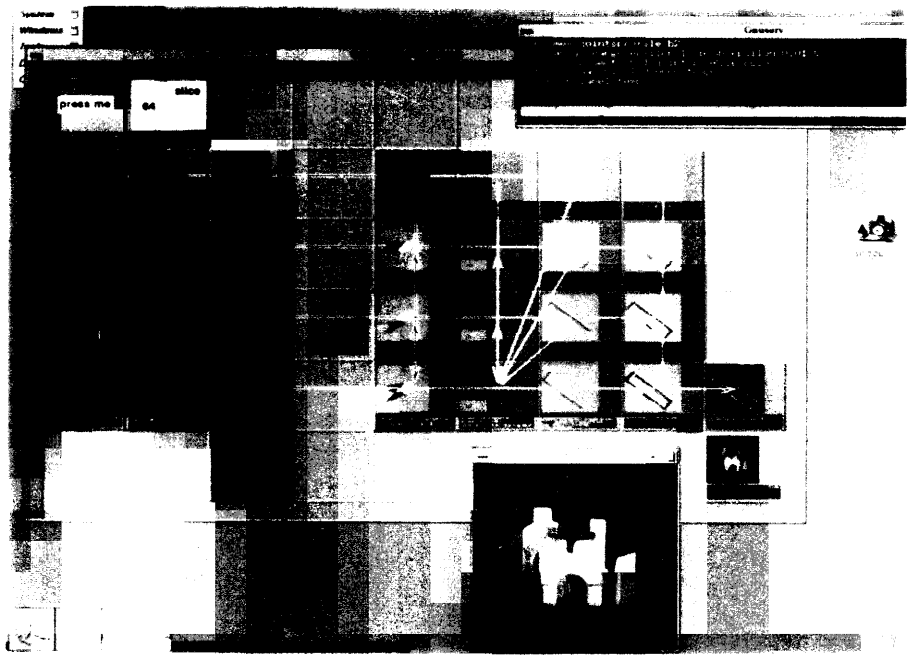

Figure 5: This **spreadsheet depicts** the flow of data **in** a 3D fax **machine—a** machine **we are building** in our **lab** for digitizing the shape and external appearance **of physical objects** using a **laser** scanner and **precision motion** platform. When the button in cell **a1** is pressed, four laser **reflection image sequences** are **loaded into cells a2 through a5, respectively. A cine viewer** widget associated wrth each **cell** displays a **frame from** that image **sequence** in miniature in the cell. **The slice specified in the** slider **in cell** bl 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** Mocks. An **Emacs editor** window is also visibk; it is currently editing the contents of the formula in all **g2.**

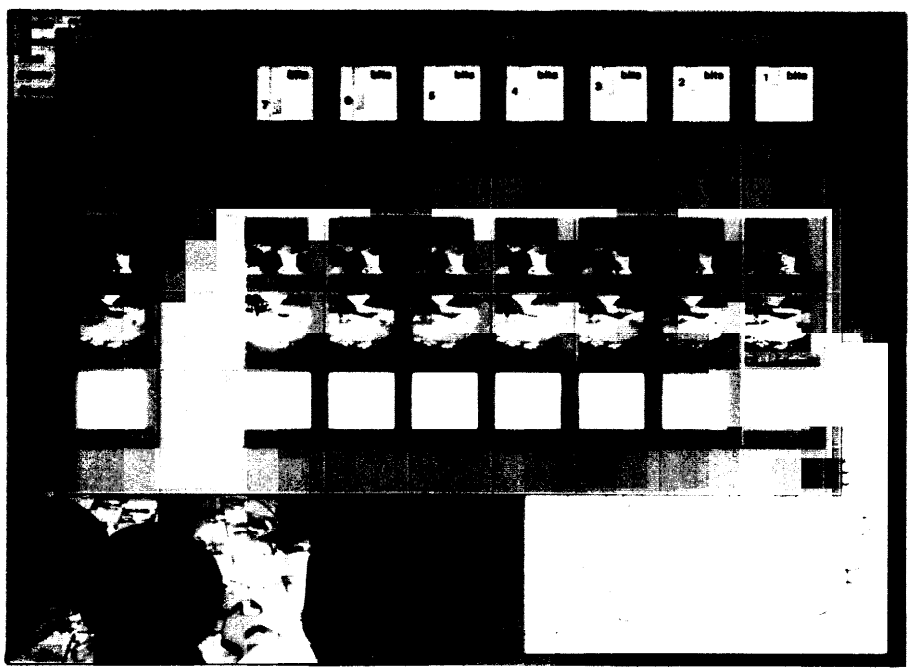

Figure 6: This spreadsheet was used to give a live **classroom demonstration** of the effects of image **quantization.** The **original 8-bit** images **are** in **cells** a.3 through 3. **The** sliders in **cells** cl **through** rl **are used to set the number** of bits to **which** the images appearing 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.**

+2 +1 -2 0 none