Graphical Definitions: Making Spreadsheets Visual through Direct Manipulation and Gestures

Herkimer J. Gottfried
Hewlett-Packard Company
1000 NE Circle Blvd., Corvallis, OR 97330
herkyg@cv.hp.com

Margaret M. Burnett*
Department of Computer Science
Oregon State University, Corvallis, OR 97331
burnett@cs.orst.edu

Abstract

Until now, attempts to extend the one-way constraint evaluation model of the spreadsheet paradigm to support complex objects, such as colored circles or user-defined types, have led to approaches featuring either a direct way of creating objects graphically or strong compatibility with the spreadsheet paradigm, but not both. This inability to conveniently go beyond numbers and strings without straying outside the spreadsheet paradigm has been a limiting factor in the applicability of spreadsheets. In this paper we present a technique that removes this limitation, allowing complex objects to be programmed directly—and in a manner that fits seamlessly within the spreadsheet paradigm—using direct manipulation and gestures. An empirical study has shown that programmers can use this technique to program complex objects faster and with fewer errors. We show that the graphical definitions technique not only expands the applicability of spreadsheet languages, it also adds to their support for exploratory programming and to their scalability.

1: Introduction

In recent years, many new graphical techniques have been developed to support the use of visual objects. Of particular note are the contributions of demonstrational programming research, which have brought straightforward, graphical techniques for creating and working with visual objects to both end-users and programmers. Unfortunately however, users of spreadsheets have been left out of these advances, and still find themselves stranded in a highly textual world with limited abilities to incorporate visual objects into their computations.

We set out to correct this problem. Our goal was to incorporate visual objects into spreadsheets in a way that would fit seamlessly within the one-way constraint model of the spreadsheet paradigm. Further, we wanted our approach, like most other features found in spreadsheets, to be applicable to *all* users of spreadsheet languages. That is, we wanted to support the simple, built-in graphical objects likely to be used by ordinary end-users, in a way general enough to also support the complex, user-defined objects needed by programmers.

In this paper we present such an approach. It allows both simple and complex objects to be defined graphically in a spreadsheet language using direct manipulation and gestures. We call these direct manipulations and gestures *graphical definitions* to emphasize that they are a declarative way to *define formulas* for cells in a *graphical* manner. The contributions of the graphical definitions approach are that (1) it is the first approach that fully supports working directly and visually with objects in a way that fits seamlessly within the spreadsheet paradigm; (2) it adds to both the support for exploratory programming and to the scalability of spreadsheet languages; and (3) it introduces gesture spaces, a technique that takes a step forward in the practicality of programming with gestures.

1.1: Design goals

We use the term *spreadsheet languages* to refer to all systems that follow the spreadsheet paradigm, from commercial spreadsheets to more sophisticated systems whose computations are defined by one-way constraints in the cells' formulas. By "fitting seamlessly within the spreadsheet paradigm," we mean that the approach follows the declarative, one-way constraint paradigm of spreadsheets, emphasizing that it should follow the *value rule* for spreadsheets, which states that a cell's value is defined solely by the formula explicitly given it by the user [9]. The characteristic of seamlessness within the spreadsheet paradigm was one of our two primary design goals.

Our other primary design goal was *directness,* a term we will use to mean following the principles advocated by Shneiderman; by Hutchins, Hollan, and Norman; and by Nardi. The term *direct manipulation* was coined by Shneiderman [16], who describes three principles of direct manipulation systems: continuous representation of the objects of interest, physical actions or presses of labeled buttons instead of complex syntax, and rapid incremental reversible operations whose effect on the object of interest is immediately visible.

Hutchins, Hollan, and Norman [8] expand upon these notions, suggesting that the degree to which a user interface feels *direct* is inversely proportional to the cognitive

* This work was supported by Hewlett-Packard and by the National Science Foundation under grants CCR-9308649, ASC-9523629, and an NSF Young Investigator Award.

effort needed to use the interface. They describe directness as having two aspects. The first aspect is the *distance* between one's goals and the actions required by the system to achieve those goals. In traditional spreadsheet programming, this distance is fairly small because there is a well-understood, one-one mapping from each operator and term in the goal to the formula that must be specified (e.g., from the goal "add A and B" to the formula "A + B"). The second aspect is a feeling of *direct engagement,* "the feeling that one is directly manipulating the objects of interest." Nardi [15] sees direct engagement as a critical element in spreadsheets, emphasizing freedom from low-level programming minutiae in favor of task-specific operations. Direct engagement has been largely absent from prior approaches to supporting graphics in spreadsheet languages.

2: Related work

Many commercial spreadsheets provide the capability to display simple graphics and charts. However, these visual objects are strictly output mechanisms: they cannot be values of cells, other cells' values cannot depend on them, and only the charts (not the other kinds of graphics) can be dependent on other cells. Furthermore, these spreadsheets do not allow users to extend the set of visual objects that are supported. In some spreadsheets, it is possible to gain some graphical support for objects through the use of macro languages and incorporation of state-modifying programming languages, but these approaches violate the spreadsheet value rule. Macros violate it because a macro stored in one group of cells actually changes other cells' formulas *during execution*—the spreadsheet equivalent of self-modifying programs.

Although some research spreadsheet languages have used graphical techniques, they have not achieved the combination of generality and directness that we sought for the spreadsheet paradigm. For example, the NoPumpII spreadsheet language [19] includes some built-in graphical types that may be instantiated using cells and formulas, and supports limited (built-in) manipulations for these objects, but does not support complex or user-defined objects. Penguims [6] is an environment based on the spreadsheet model for specifying user interfaces. It supports complex objects by providing the capability to collect cells together into *objects*, but it also introduces several new concepts that violate the spreadsheet model, such as interactor objects that can modify the formulas of other cells, and imperative code similar to macros. Action Graphics [7] is a spreadsheet language for graphics animations. It also provides some support for complex objects, but does not provide the directness we sought. A version of Prograph incorporates user interface objects into a spreadsheet in order to provide spreadsheet users with a graphical interface for input and feedback [17]. However, although the Prograph approach to spreadsheets adds the ability to incorporate visual objects into spreadsheets, it does not make programming them more direct.

Wilde's WYSIWYC spreadsheet [20] aims to improve traditional spreadsheet programming by making cell formulas visible and by making the visible structure of the spreadsheet match its computational structure. Although this work is similar to ours in its attempt to emphasize the task-specific operations of spreadsheet languages, Wilde focuses on the resulting program rather than on the means of specifying it, and does not address visual types.

C32 [14] uses graphical techniques along with inference to specify constraints in user interfaces. Unlike the other spreadsheet languages described, C32 is not a full-fledged spreadsheet language; rather, it is a front-end to the underlying textual language Lisp used in the Garnet user interface development environment [13]. C32 is a way of viewing constraints, but does not itself feature the graphical creation and manipulation of visual objects. Instead, this function is performed by another part of the Garnet package. Together C32 and the other portions of the Garnet package feature strong support for direct manipulation of built-in graphical user interface objects, but not for any other kinds of objects, which must be written and manipulated in Lisp.

Our work is also related to pen-based computing [4] and to programming by demonstration [3]. Of the latter, the most closely related to our work are systems featuring declarative approaches to programming. KidSim [18] is a demonstrational system that uses direct manipulation to specify declarative graphical rewrite rules. Although KidSim's approach is similar to ours in its emphasis on directness, it does not provide the kind of flexible, declarative specification of objects and attributes that we sought for a full-featured, spreadsheet-based approach. The multi-way constraint-based systems TRIP3 [11] and IMAGE [12] use direct manipulation as a means of specifying relations declaratively; in these systems a visual example defines a relationship between the application data and its visual representation. However, our interest was in finding a declarative approach that would work with the *one*-way constraints of spreadsheet languages.

3: Programming visual objects directly

We have prototyped our approach in the spreadsheet language Forms/3 [1, 2], and the examples in this paper are presented in that language. Forms/3 has long supported both built-in graphical types and user-defined graphical types. (Built-in types are provided in the language implementation but are otherwise identical to user-defined types.) Attributes of a type are defined by formulas in groups of cells, and an instance of a type is the value of an ordinary cell that can be referenced just like any other cell.

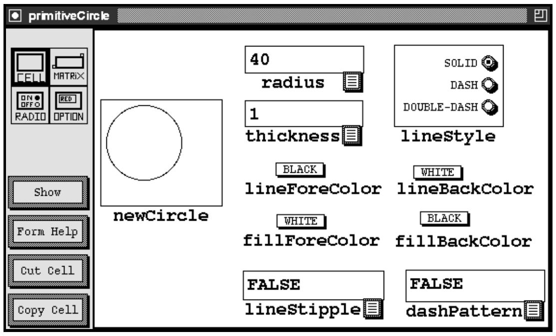


Figure 1: A portion of a Forms/3 form that defines a circle. The circle in cell *newCircle* is defined by the other cells, which define its attributes. A user can view and specify spreadsheet formulas by clicking on the formula tabs (); radio buttons and popup menus can be used to specify constant formulas.

For example, the built-in circle object shown in Figure 1 is defined by cells defining its radius, line thickness, color, and other attributes. In this paper we show how this approach can be extended to support the direct style that characterizes spreadsheets.

3.1: How are graphical definitions used?

To introduce graphical definitions, we consider tasks that a traditional spreadsheet user might be interested in performing, but that are difficult to do or are beyond the capabilities of current spreadsheets. One such task is displaying a graphical representation of data, using domain-specific visualization rules. Figure 2 shows such a visualization that a population analyst might wish to specify in a spreadsheet language. The program categorizes population data into cities, towns, and villages, and represents each with a differently sized black circle. The population analyst can use our approach to define these graphical objects using direct manipulation and gestures.

Simple graphical objects such as circles can be defined by drawing a gesture in the shape of the object, and can be sized by directly manipulating the object. To define the large *city* circle for the program, the population analyst first draws a circle gesture (Figure 3a). This defines the cell's formula to be a reference to cell *newCircle* on a copy

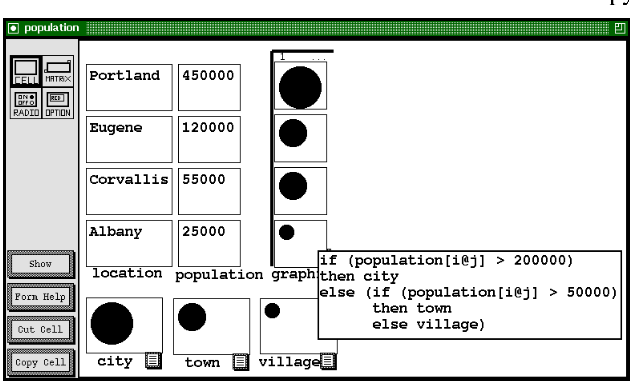


Figure 2: A visualization of population data.

of the built-in circle definition form whose radius formula is defined to be the radius of the drawn circle gesture. However, the circles in the program are to be solid black. Because there are no gestures to specify fill color, the population analyst clicks on the circle to display its definition form, and then defines the formula for cell *fillForeColor* (Figure 3b).

One way to define the circles for cells *town* and *village* is by using direct manipulation to specify how they are different from the *city* circle. For example, to define the *town* circle, the population analyst clicks on cell *city* instead of drawing a new circle. This displays the circle in the formula edit window so that it can be manipulated (Figure 4). The population analyst then resizes the circle in the formula window to define the *town* circle, which has all of the attributes of the *city* circle except its radius.

3.2: Graphical definitions and the value rule

That graphical definitions are consistent with the value rule follows from the fact that they have the same semantics as Forms/3's previous, textual-spreadsheet-formula mechanism, which is consistent with the value rule. For example, cell *town* would have been specified in

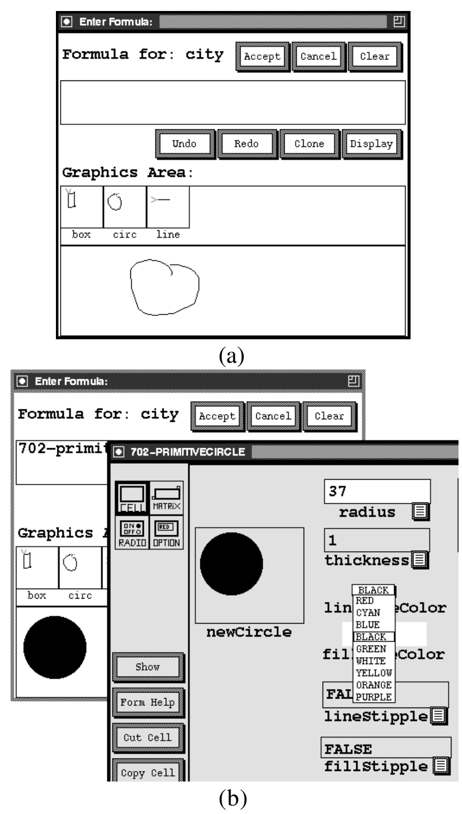
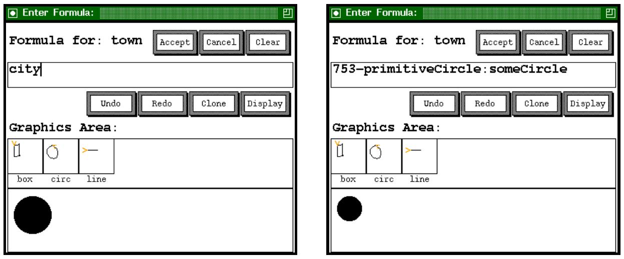


Figure 3: Defining the circle for cell *city*. (a) The population analyst first draws a *circle* gesture to define the circle. (b) After clicking on the circle to display its definition form, the population analyst defines the *fillForeColor* formula via a popup menu.



(a) (b)

Figure 4: Defining the circle for cell *town.* (a) The population analyst clicks on cell *city* to display the large *city* circle. (b) The population analyst directly manipulates the circle to define the smaller *town* circle, which has the black color and other attributes of the original circle.

the previous approach by copying the built-in *primitive-Circle* form (Figure 1), entering new *radius* and *fillFore-Color* formulas, and referring to the copy's cell *newCircle*.

Note that with graphical definitions, what the user directly manipulates is a cell's formula, not its value. If graphical definitions operated on cell values instead, the spreadsheet value rule would be violated. For example, if a cell *x*'s formula is "*newCircle*", then manipulating *x*'s value instead of its formula would have to contradict *x*'s formula or change *newCircle*'s formula, neither of which would be consistent with the value rule. Our system makes explicit the fact that the user is manipulating formulas by supporting graphical definitions in the formula edit window instead of in the main part of the spreadsheet.

3.3: Using gestures with user-defined types

Even traditionally abstract types are visual if a programmer chooses to think of them as such. To demonstrate the generality of the approach, we show in this section how graphical definitions can be used even in a traditional data processing example, such as a search.

Suppose the programmer wants to develop a binary search algorithm using a binary tree that was previously

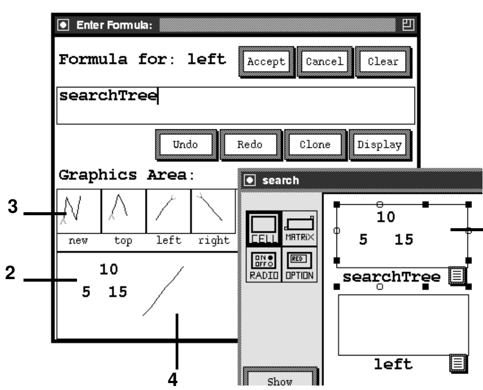


Figure 5: The programmer clicks on the (**1**) search tree to set the (**2**) context for the gesture. (**3**) Iconic representations of the tree gestures are automatically displayed. The programmer (**4**) then draws a gesture to reference the left subtree.

implemented by some other programmer. The user-defined tree type contains operations to insert a *new* element into a tree, report the *top* element of the tree, and report the *left* and *right* subtrees. The tree implementor has also defined gestures, which are automatically displayed, to perform these operations (Figure 5(**3**)). The gestures allow the low-level details of the tree implementation to be abstracted away, letting the programmer of the search algorithm perform tree operations without explicitly copying the tree definition form, defining new formulas for cells on the definition form, or explicitly referencing those cells.

To program a standard recursive search algorithm given such a tree, the programmer can use graphical definitions to access different elements of the search tree. For instance, if the search element is smaller than the top element of the tree, the search algorithm is called recursively on the *left* subtree. (Recursion is supported in Forms/3 by referencing cells on copies of the form being defined, which are then automatically generalized using a deductive technique [21].) The programmer can define a formula to access the left subtree by clicking on the search tree cell and drawing the *left* gesture—a line pointing down to the left (Figure 5). This direct action defines a formula that is equivalent to that defined by copying the tree definition form, defining the formula for cell *inputTree* on that form to be a reference to the search tree, and referencing cell *left.* (The tree definition form will be discussed in detail in the next section.) However, unlike the actions of copying the form and writing textual formulas, this gesture corresponds directly to the programmer's intent: "I want *that* tree's *left* subtree."

4: Defining new visual objects

A programmer defines a new type with a *type definition form* [2] containing an *abstraction box,* which defines an instance of the type as the composition of its attributes; an *image cell,* whose formula defines the type's appearance(s); and other cells and abstraction boxes to specify the type's operations.

For example, a tree definition form (Figure 6) might contain input abstraction box *inputTree* intended to contain an incoming tree, input cell *newElement* for an element to be inserted into the tree, and output abstraction box *newTree* to define a tree into which the new element has been inserted. Other cells providing operations for the tree (such as the predicate reporting whether the incoming tree is empty, and a cell reporting the root element) are also usually present. Multiple instances of a tree can be instantiated using multiple copies of the form, which are the underlying equivalent of the graphical definitions shown in the example of the previous section.

For graphical definitions to be possible with such a type, a programmer needs a way to specify the set of

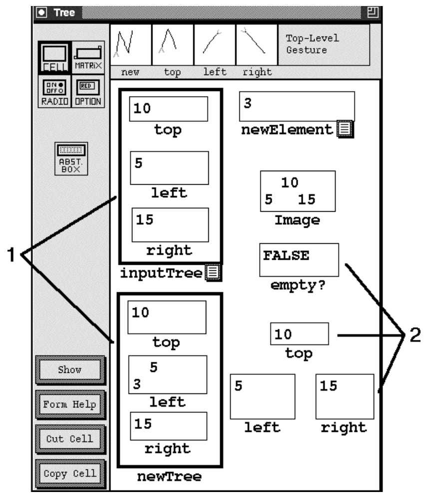


Figure 6: A tree definition form. The cells inside the (1) abstraction boxes are by definition "hidden," and cannot be accessed by cells outside this form. The implementor of the tree has provided (2) access cells such as *empty?* to report attributes of *inputTree*. The formula tabs on cells *newElement* and *inputTree* signify that these cells are intended for input. (The formulas are not shown.)

graphical definitions for the type, enabling their use for purposes such as the binary search algorithm of the previous section.

4.1: How do programmers define new gestures?

The first step in specifying the set of graphical definitions for a user-defined type is to specify the set of gestures that are applicable to the type. In our implementation, gestures are defined and trained using the Agate gesture recognizer [10], which is part of the Garnet environment [13]. The programmer presses a button on the type definition form to start Agate, and then types the name of a gesture and draws a few examples of the gesture. Our implementation automatically displays miniature gesture icons at the top of the type definition form when Agate is exited.

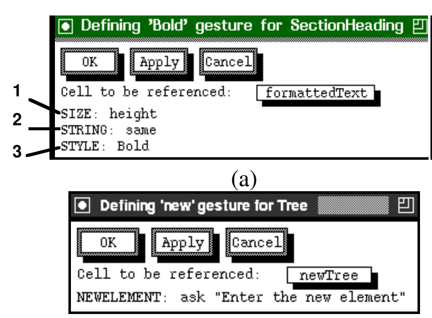
After demonstrating the gesture, the programmer specifies the gesture's semantics—the formulas that will be defined when the gesture is drawn. For instance, the *new* tree gesture at the top of Figure 6 specifies a reference to cell *newTree* on a copy of the tree definition form, in which the formula for cell *newElement* is the element to be inserted into the tree, and the formula for the abstraction box *inputTree* is a reference to the tree being manipulated.

As this example shows, the gesture's semantics are specified by two things: a cell to be referenced, and formula specifications for each of the definition form's input cells. (Because the formula for the input abstraction box is always a reference to the object being manipulated, its formula is defined automatically.) There are four ways a formula can be specified by a gesture:

- A gesture attribute formula specification for a cell says that the formula is defined by some attribute of the gesture itself, such as its height, width or radius. For example, a programmer defining a gesture for formatting text might define cell *size*'s formula to be defined as the gesture's height (Figure 7a(1)).
- A *same* formula specification says that this cell's formula on the new object's definition form is the same as on the original object's definition form (Figure 7a(2)).
- A *constant* formula specification says that this cell's formula is simply the name of the gesture (Figure 7a(3)).
- An *askUser* formula specification says that the user will be asked to specify the formula for the cell after the gesture is drawn. The *new* tree gesture (Figure 7b) defines an *askUser* formula specification for cell *newElement.* When the gesture is drawn, a dialog box will be opened asking the user to enter a formula for cell *newElement* (Figure 8).

In addition to specifying gestures that derive one object from another, the programmer can specify a gesture to create an object not derived from any other object. To specify such a gesture, the programmer presses the "top-level gesture" button on the type's definition form, and specifies a new gesture (whose name is the name of the type). This gesture is automatically added to the set of gestures understood by the top-level gesture recognizer.

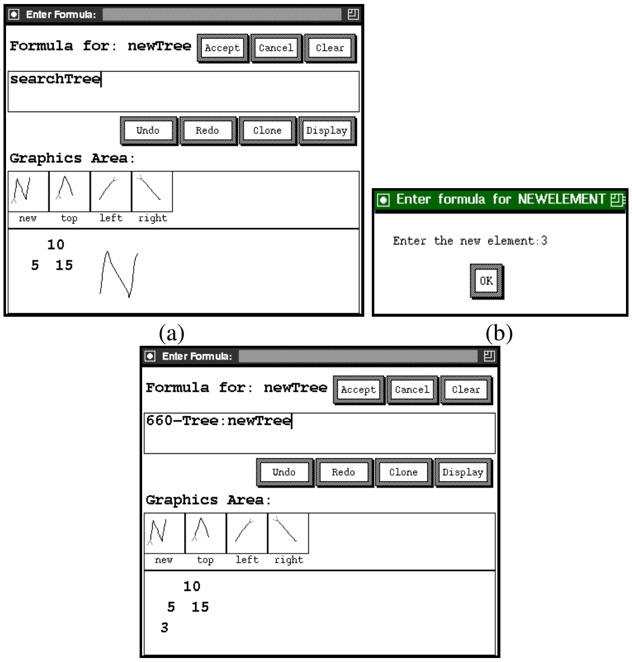
Top-level gestures are important to the consistency of the approach for two reasons. First, they allow user-defined types to be instantiated with the same directness that is provided for built-in types. Second, they provide the same interface for instantiating a new visual object "from scratch" as for deriving one object from another object.



(a)

(b)

Figure 7: Defining gesture semantics. (a) The *bold* gesture defines a reference to cell *formattedText* on a copy of the sectionHeading definition form in which (1) the formula for cell *size* is defined to be the height of the drawn gesture, (2) *string* is defined to be the same as the *string* formula for the sectionHeading object being manipulated, and (3) *style* is the constant "Bold". (b) The *new* tree gesture defines a reference to cell *newTree* on a copy of the tree definition form whose *newElement* formula is to be entered by the user.



(c)

Figure 8: Using graphical definitions to insert a new element into a tree. (a) The programmer draws the *new* gesture. (b) After drawing the gesture, the programmer is prompted for the element to be inserted. (c) The resulting formula is a reference to a new copy of the tree definition form in which cell *newElement* has the formula 3 and cell *inputTree* is a reference to the original tree.

5: Other contributions of the approach

5.1: Gesture spaces

Landay and Myers [10] mention the desirability of supporting context-dependent gestures: "The system needs a way to map the same gesture into multiple meanings based on the context." Our approach solves this problem by making the set of gestures recognized by the gesture classifier depend on the context of the formula being edited. Recall from Figure 5 that the programmer clicks on a value of the desired type to set the context.

By partitioning the gestures into different *gesture spaces* (a concept similar to name spaces in programming languages), gestures need only be distinct within a specific context. For example, the top-level gestures and type-specific gestures may overlap. This allows a gesture to be reused in different contexts, while eliminating possible ambiguities over the gesture's meaning. Thus, the set of allowable gestures for any context remains relatively small and recognizable even for large programs.

The scope rules used in our approach to determine which gestures are applicable in any context are simple. If the formula being edited is a reference to an instance of a user-defined type or to a cell on a user-defined type definition form, then the set of gestures for that type—and only those gestures—will be recognized. Otherwise, the recognized gestures are the set of top-level gestures.

One problem that programmers in any programming language face is that of remembering the permissible operations on an object, and this problem is exacerbated if the operations are invisible gestures that must be memorized. Our approach addresses this problem by displaying miniature icons of the allowable gestures (and their names) for the current context. These icons document the set of allowable operations, and can even be used as an alternative means of specifying gestures: rather than drawing a gesture, a programmer can click on a gesture icon. (This technique has fewer degrees of freedom than an actual gesture, but communicates equivalent information as a gesture whose size and orientation are unimportant.) The partitioning of the gestures into different gesture spaces along with the automatic display of the allowable gestures contributes to the practicality of our approach, keeping the set of operations permissible at any one time small, recognizable, and visible.

5.2: Exploratory programming

One popular use of spreadsheets is in investigating "what-if" scenarios, in which users experiment with different formulas for cells to see what values they produce for other cells. Our approach extends this support for exploratory programming to visual objects. By exploratory programming, we mean allowing the programmer to interactively gesture and directly manipulate objects, immediately see the effects of these manipulations, and use this feedback to perform further manipulations. This is supported by our approach in a number of ways that work together to satisfy Shneiderman's third principle of direct manipulation: *rapid incremental reversible operations whose effect on the object of interest is immediately visible* [16].

Because the result of applying a graphical definition to an object is a new object to which further manipulations may be applied, our approach provides incremental operations. The new object defined by a graphical definition is immediately displayed and manipulable, so the effects of such manipulations are immediately visible. And because graphical definitions define declarative formulas for cells rather than performing any state modification, it is trivial to provide reversible operations—just revert to the previous formula for the cell. We have added *undo* and *redo* buttons in the formula edit window that allow the programmer to easily and quickly undo (or redo) the effects of any graphical definition.

Exploratory programming can aid in understanding and debugging complex data structures. For instance, a programmer might test the correctness of the implemen-

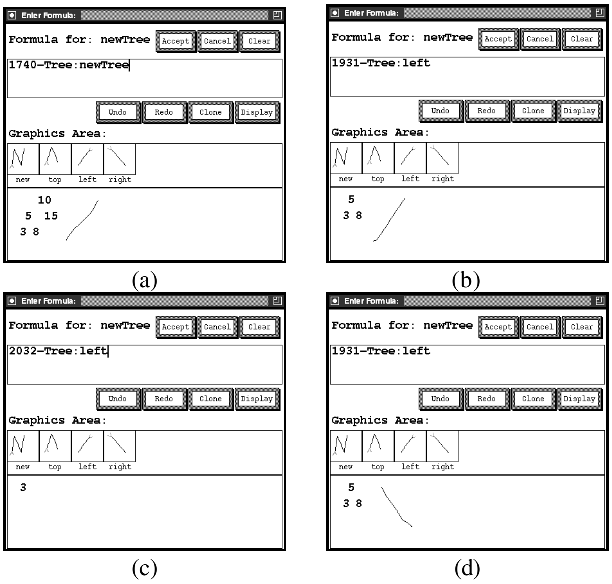


Figure 9: Using gestures to explore a binary tree. (a) The programmer draws a *left* gesture to show the left subtree. (b) The subtree is immediately displayed, and the programmer can draw another gesture to show *its* left subtree. (c) The resulting subtree (the single element 3) is now shown. (d) The programmer has pressed the *undo* button to revert to the previous formula, and can now explore the right subtree or perform other manipulations.

tation of the binary tree by creating a tree, inserting a few new elements, and then accessing the top element and left and right subtrees to ensure that they are correct. Without graphical definitions, this process is straightforward but somewhat tedious: define formulas for cells *inputTree* and *newElement,* create another tree, define its *inputTree* formula to reference cell *newTree* on the previous form, and so on. With graphical definitions, the programmer simply draws a *tree* gesture, a few *new* gestures, and then explores the tree by drawing *top, left,* and *right* gestures (Figure 9). Explorations like this for even a small tree with only a few elements would require the creation of several forms and the definition of several formulas, whereas gestures provide the same functionality more quickly, more directly, and with more flexibility.

5.3: Scalability

Another practical contribution of our approach is that it allows the screen real estate and memory usage of a spreadsheet program to be reduced significantly, thus helping make spreadsheet languages more suitable for building large applications. Perhaps even more important to the programmer is that graphical definitions reduce the amount of work required to create programs containing visual objects (Table 1). To consider a small example, building the population visualization program shown in Figure 2 without graphical definitions would have required the programmer to copy the circle definition form 3 times, to define the *radius* formula on each copy, and to reference each circle from the population form; whereas graphical definitions required only a single copy of the definition form to define the first circle's fill color. Although each visual object specified with a graphical definition is defined by a definition form behind the scenes, only the visual object itself is explicitly displayed onscreen; its definition form is shown only if the programmer elects to display it by clicking on the object. Because so many fewer visual components need to be constructed, displayed, and redrawn, supporting the programmer's manipulations requires less screen real estate, less memory, and less computation time.

Actions needed to create visual objects *without* graphical definitions					
To create these visual objects	*# formulas defined*	*# gestures*	*# cells referenced*	*# off-form cells referenced*	*# type definition forms copied*
3 circles (population program)	9	N/A	3	3	3
n circles (population program)	3*n*	N/A	*n*	*n*	*n*
3-element search tree	6	N/A	3	3	3
n-element search tree	2*n*	N/A	*n*	*n*	*n*
Actions needed to create visual objects *with* graphical definitions					
To create these visual objects	*# formulas defined*	*# gestures*	*# cells referenced*	*# off-form cells referenced*	*# type definition forms copied*
3 circles (population program)	4	3	2	0	1
n circles (population program)	$n + 1$	*n*	$n - 1$	0	1
3-element search tree	1	4	0	0	0
n-element search tree	1	$n + 1$	0	0	0

Table 1: Programmers perform fewer actions using graphical definitions; in some cases the reduction is as much as a factor of *n*. Of particular importance is the reduction in the more complex programming actions; that is, those that require multiple forms (linked spreadsheets), shown in the two rightmost columns.

6: Empirical study

We conducted an empirical study in which 20 graduate students wrote two small Forms/3 programs, using graphical definitions on one and on the other explicitly copying forms and entering textual formulas. The details of the study are given in [5]; its primary results were that significantly more programs were completed correctly with graphical definitions than with the copying technique, and that the programs were completed significantly faster using graphical definitions.

7: Conclusion

Spreadsheets have traditionally been limited to supporting only the simplest of textual types, namely numbers and strings. Prior attempts to remove this limitation have resulted in a number of interesting approaches, but none of them have featured a seamless fit within the one-way constraint model of the spreadsheet paradigm while still satisfying the principles of directness advocated by Shneiderman; by Hutchins, Hollan, and Norman; and by Nardi.

Graphical definitions solve this problem, demonstrating that direct manipulation and gestures can be used to specify formulas in a spreadsheet language in a way that is entirely compatible with the spreadsheet value rule. By integrating modern techniques from visual programming and direct manipulation programming seamlessly into spreadsheet languages, graphical definitions increase not only the applicability of spreadsheet languages, but also their support for exploratory programming and their scalability.

8: References

[1] Atwood, J., M. Burnett, R. Walpole, E. Wilcox, and S. Yang, "Steering Programs via Time Travel," *1996 IEEE Symp. on Visual Languages*, Boulder, CO, Sept. 3-6, 1996, 4-11.

[2] Burnett, M. and A. Ambler, "Interactive Visual Data Abstraction in a Declarative Visual Programming Language," *Journal of Visual Languages and Computing,* March 1994, 29-60.

[3] Cypher, A. (ed.), *Watch What I Do: Programming by Demonstration*, MIT Press, Cambridge, MA (1993).

[4] Egenhofer, M., "Spatial-Query-by-Sketch," *1996 IEEE Symp. on Visual Languages*, Boulder, CO, Sept. 3-6, 1996, 60-67.

[5] Gottfried, H. and M. Burnett, "Programming Complex Objects in Spreadsheets: An Empirical Study Comparing Textual Formula Entry with Direct Manipulation and Gestures," *Proc. Empirical Studies of Programmers*, Washington DC, Oct. 1997.

[6] Hudson, S., "User Interface Specification Using an Enhanced Spreadsheet Model," *ACM Trans. on Graphics*, July 1994, 209-239.

[7] Hughes, C. and J. Moshell, "Action Graphics: A Spreadsheet-Based Language for Animated Simulation," in *Visual Languages and Applications* (T. Ichikawa, E. Jungert, R. Korfhage, eds.), Plenum Publishing, New York, NY (1990), 203-235.

[8] Hutchins, E., J. Hollan, and D. Norman, "Direct Manipulation Interfaces," in *User Centered System Design: New Perspectives on Human-Computer Interaction* (D. Norman, S. Draper, eds.), Lawrence Erlbaum Assoc., Hillsdale, NJ (1986), 87-124.

[9] Kay, A., "Computer Software," *Scientific American,* Sept. 1984, 53-59.

[10] Landay, J. and B. Myers, "Extending an Existing User Interface Toolkit to Support Gesture Recognition," *Adjunct Proc. INTERCHI '93,* Amsterdam, The Netherlands, Apr. 24-29, 1993, 91-92.

[11] Miyashita, K., S. Matsuoka, S. Takahashi, A. Yonezawa, and T. Kamada, "Declarative Programming of Graphical Interfaces by Visual Examples," *ACM Symp. on User Interface Software and Technology,* Monterey, CA, Nov. 15-18, 1992, 107-116.

[12] Miyashita, K., S. Matsuoka, S. Takahashi, and A. Yonezawa, "Iterative Generation of Graphical User Interfaces by Multiple Visual Examples," *ACM Symp. on User Interface Software and Technology,* Marina del Rey, CA, Nov. 2-4, 1994, 85-94.

[13] Myers, B., et al., "Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces," *Computer,* Nov. 1990, 71-85.

[14] Myers, B., "Graphical Techniques in a Spreadsheet for Specifying User Interfaces," *CHI '91,* New Orleans, LA, Apr. 28 - May 2, 1991, 243-249.

[15] Nardi, B., *A Small Matter of Programming: Perspectives on End User Computing,* MIT Press, Cambridge, MA (1993).

[16] Shneiderman, B., *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, Addison-Wesley, Reading, MA (1992).

[17] Smedley, T., P. Cox, and S. Byrne, "Expanding the Utility of Spreadsheets Through the Integration of Visual Programming and User Interface Objects," *Advanced Visual Interfaces '96*, Gubbio, Italy, May 27-29, 1996, 148-155.

[18] Smith, D., A. Cypher, and J. Spohrer, "KidSim: Programming Agents Without A Programming Language," *Comm. ACM,* July 1994, 55-67.

[19] Wilde, N. and C. Lewis, "Spreadsheet-Based Interactive Graphics: From Prototype to Tool," *CHI '90,* Seattle, WA, Apr. 1-5, 1990, 153-159.

[20] Wilde, N., "A WYSIWYC (What You See Is What You Compute) Spreadsheet," *1993 IEEE Symp. on Visual Languages*, Bergen, Norway, Aug. 24-27, 1993, 72-76.

[21] Yang, S. and M. Burnett, "From Concrete Forms to Generalized Abstractions through Perspective-Oriented Analysis of Logical Relationships," *1994 IEEE Symp. on Visual Languages,* St. Louis, MO, Oct. 4-7, 1994, 6-14.