AN ABSTRACT OF THE THESIS OF

Herkimer John Gottfried for the degree of Master of Science in Computer Science presented on December 9, 1996. Title: Graphical Definitions: Expanding Spreadsheet Languages through Direct Manipulation and Gestures.

Abstract approved:

[illegible]

Margaret M. Burnett

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 thesis 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. We also present the results of an empirical study which suggests that programmers can use this technique to program complex objects faster and with fewer errors. 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.

Graphical Definitions:
Expanding Spreadsheet Languages through Direct Manipulation and Gestures

by

Herkimer John Gottfried

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed December 9, 1996
Commencement June 1997

Master of Science thesis of Herkimer John Gottfried presented on December 9, 1996

APPROVED:

[illegible]

Major Professor, representing Computer Science

Chair of Department of Computer Science

Redacted for privacy

Dean of Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for privacy

Herkimer John Gottfried, Author

ACKNOWLEDGMENT

We would like to thank the members of our research group, in particular John Atwood, Rebecca Walpole, and Sherry Yang, for their work on the Forms/3 implementation and for their feedback on graphical definitions. Special thanks go to Shikha Ghosh Gottfried and Judy Hays for their help in testing the empirical study, and to the subjects of the study for their participation.

This work was supported in part by Hewlett-Packard and by the National Science Foundation under grant CCR-9308649 and an NSF Young Investigator Award.

Table of Contents

Page

Table of Contents, Continued

List of Figures

List of Tables

List of Appendix Figures

Graphical Definitions: Expanding Spreadsheet Languages through Direct Manipulation and Gestures

1. Introduction

In recent years, many new graphical techniques have been developed to support the use of graphical objects. Of particular note are the contributions of demonstrational programming research, which have brought straightforward, graphical techniques for creating and working with graphical 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 graphical objects into their computations.

We set out to correct this problem. Our goal was to incorporate graphical 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 thesis 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 provides fully declarative, graphical support for working directly 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 contributes gesture spaces, a technique that takes a step forward in the practicality of programming with gestures.

1.1 Organization of this Thesis

We begin with a discussion of the design goals of our approach. In Chapter 2, we review related work, evaluating other systems with regard to these design goals. In Chapter 3 we provide a brief introduction to the Forms/3 spreadsheet language in which our approach is prototyped, along with examples of how our technique might be used by end-users and by programmers. In Chapter 4 we present the formal semantics of graphical definitions. We describe other contributions of the approach in Chapter 5. We present the results of an empirical study in Chapter 6, and we conclude in Chapter 7.

1.2 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 [8]. 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 [19, 20], 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 [7] expand upon these notions, suggesting that the degree to which a user interface feels *direct* is inversely proportional to the cognitive 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 [18] 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

Our approach is most closely related to research in spreadsheet languages and demonstrational systems.

2.1 Spreadsheet Languages

Microsoft Excel [13] and other commercial spreadsheets provide the capability to display simple graphics and charts in spreadsheets. However, these graphical objects are strictly output mechanisms rather than first-class objects. 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 in the spreadsheet. Furthermore, these spreadsheets do not allow users to extend the set of graphical 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, NoPumpG [11] and NoPumpII [25] are simple spreadsheet languages designed to support interactive graphics. The design goal of these systems was to provide the capability to create low-level graphical primitives while adding as little as possible to the basic spreadsheet paradigm. Thus, NoPumpG and NoPumpII include some built-in graphical types that may be instantiated using cells and formulas, and support limited (built-in) manipulations for these objects, but do not support complex or user-defined objects.

Penguims [5] is an environment based on the spreadsheet model for specifying user interfaces. Its goal is to allow interactive user interfaces to be created with little or no explicit programming. This work is similar to ours in its support for abstraction—it provides 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 formula of other cells, and imperative code similar to macros. Penguims provides the capability to build interactive user interfaces, but this programming is still done indirectly by defining formulas for cells.

Action Graphics [6] is a spreadsheet language for graphics animations. It provides some support for complex objects, such as the ability to group cells into "composite cells," but does not provide the directness we sought. Also, animation in Action Graphics is performed through functions that cause side-effects; thus, this approach violates the spreadsheet value rule.

Smedley, Cox, and Byrne [21] have incorporated the visual programming language Prograph and user interface objects into a conventional spreadsheet in order to provide spreadsheet users with a graphical interface for input and feedback. However, like Penguims, this approach does not follow the value rule because user interface objects can modify the formulas of other cells. Also, although the Prograph approach to spreadsheets adds the ability to incorporate graphical objects into spreadsheets, it does not make programming them more direct.

Wilde's WYSIWYC spreadsheet [26] 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 visual representation of the resulting program rather than on the means of specifying it, and does not address graphical types.

C32 [17] is a spreadsheet language that 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 [16]. C32 is a way of viewing constraints, but does not itself feature the graphical creation and manipulation of graphical objects. Instead, this function is performed by the demonstrational system Lapidary [23], which is another part of the Garnet package. The combination of C32 and Lapidary (and the other portions of the Garnet package) features 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.

2.2 Demonstrational Systems

Our work is also related to research on demonstrational programming by direct manipulation of objects, such as Chimera [9], KidSim [22], Mondrian [12], TRIP3 [14], and IMAGE [15]. Of these, the most closely related to our work are those featuring a declarative approach. KidSim [22] is a demonstrational system that uses direct manipulation to specify declarative graphical rewrite rules. Although the approach used in KidSim 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 two-way constraint-based systems TRIP3 [14] and IMAGE [15] 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, like many demonstrational systems, their approach uses inference to determine this relation rather than having the relation be specified explicitly by the programmer. Although our system shares with inferential languages the property that concrete examples are used in programming, our approach avoids using inference to derive the logic [27]. Also, the purpose of TRIP3 and IMAGE is to provide a *visual* interface to *textual* programming languages, while our approach attempts to extend the power of the spreadsheet without involving any other programming language.

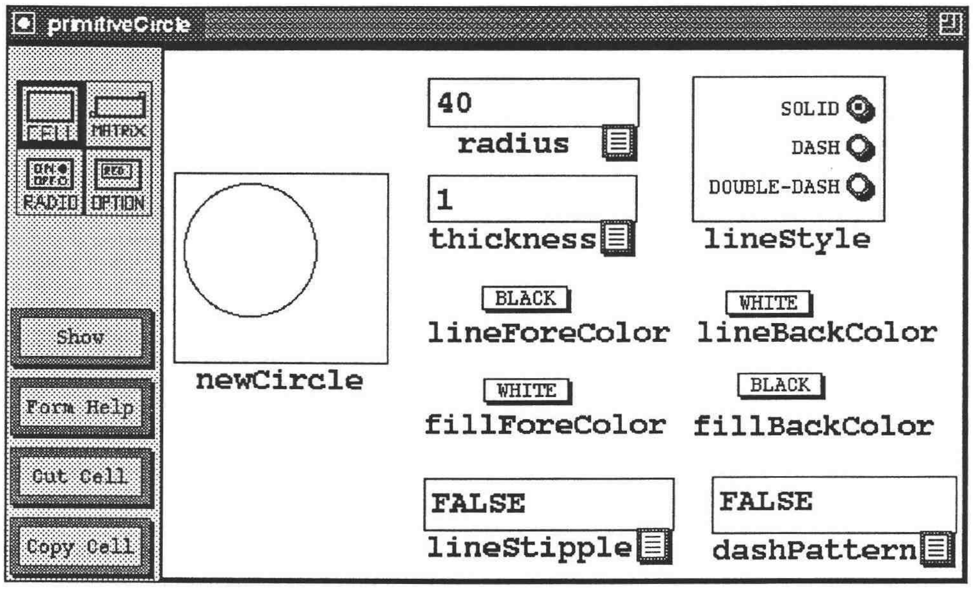
Furthermore, two-way constraints are not completely compatible with the spreadsheet paradigm because they violate the spreadsheet value rule. To see why, imagine specifying the formula for cell X to be a box whose width is a reference to cell W (whose formula is cell A plus cell B). If the user then selects and stretches the box in X, what does that mean for cells W, A, and B? If any of these are automatically changed, the value rule is violated for the changed cell(s); if they are not changed, the two-way nature of the constraints is not being maintained.

3. Programming Graphical Objects Directly

3.1 Introduction to Forms/3

We have prototyped our approach in the spreadsheet language Forms/3 [1, 2], and the examples in this thesis are presented in that language. Programs in Forms/3 consist of forms (similar to sheets in commercial spreadsheets) that contain cells whose values are defined by their formulas. In addition to traditional spreadsheet cells, Forms/3 supports 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. For example, the built-in circle object shown in Figure 3-1 is defined by cells defining its radius, line thickness, color, and other attributes.

The straightforward approach used in [2] to program graphical types is to make a copy of the definition form for each new instance of the type, specify formulas for each object's attributes, and then reference the new objects (see Figure 3-2). However, although this approach satisfies the value rule, it is a very indirect way of specifying graphical objects, because the process of specifying the object bears little resemblance to the object itself, violating the principles of directness advocated by Nardi and by Hutchins, Hollan, and Norman. The graphical definitions technique presented in this thesis solves this problem of indirectness, extending the approach of [2] to support the direct style that characterizes spreadsheets.



newCircle

radius: 40

thickness: 1

lineStyle: [✓ SOLID] DASH DOUBLE-DASH

lineForeColor: BLACK

lineBackColor: WHITE

fillForeColor: WHITE

fillBackColor: BLACK

lineStipple: FALSE

dashPattern: FALSE

Figure 3-1. A portion of a form used to define a circle in Forms/3. 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.

3.2 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 3-3 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.

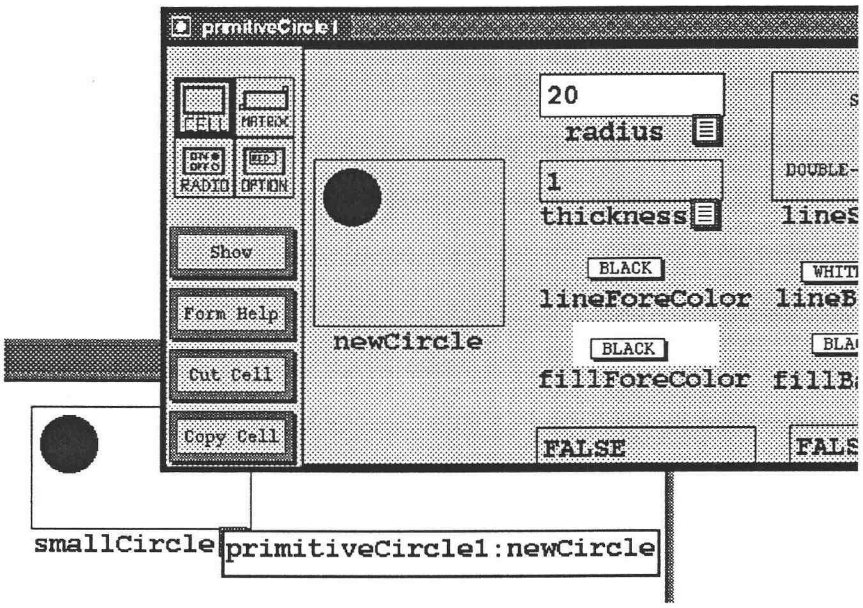
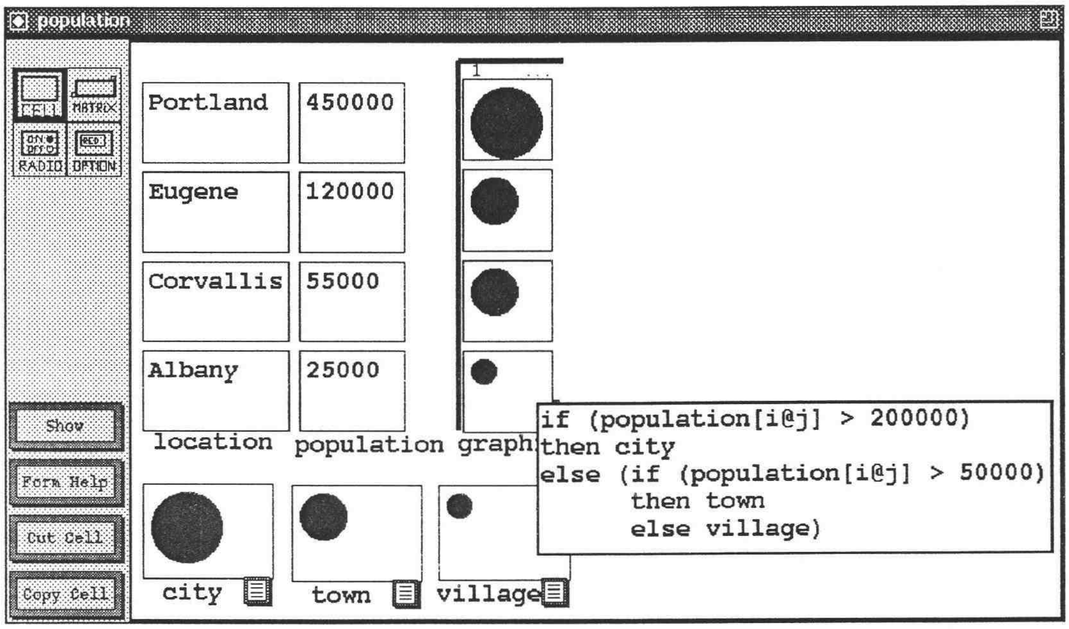


Figure 3-2. The indirect approach. A new circle is defined by copying the circle definition form and specifying formulas for some of its attributes (*radius* and *fillForeColor* in this example). The circle can be used in a program by referencing cell *newCircle*. This thesis presents a more direct approach that allows complex objects to be programmed using direct manipulation and gestures, rather than by copying forms and defining formulas.

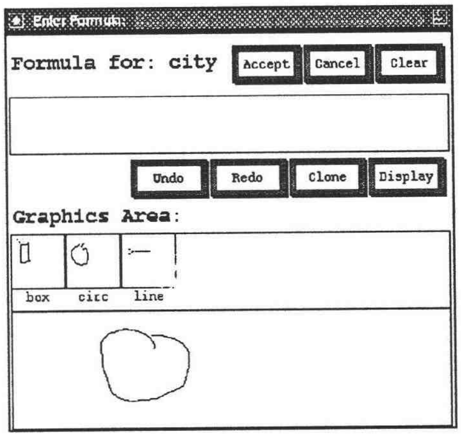


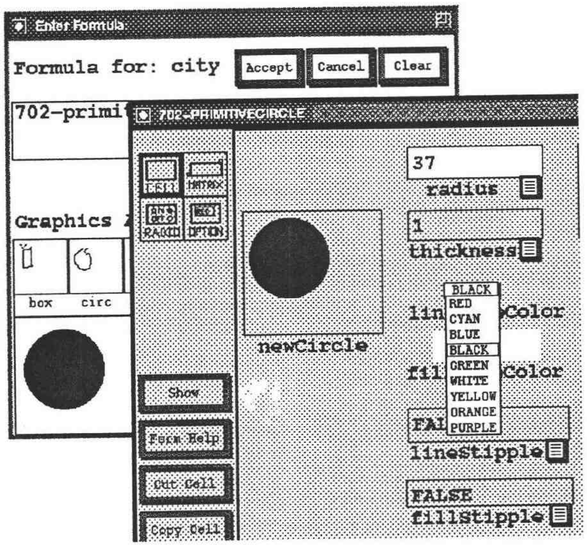
Portland	450000
Eugene	120000
Corvallis	55000
Albany	25000

```
if (population[i@j] > 200000)
then city
else (if (population[i@j] > 50000)
      then town
      else village)
```

Figure 3-3. A visualization of population data.

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 visualization program, the population analyst first draws a circle gesture (Figure 3-4(a)). This defines the cell's formula to be a reference to cell *newCircle* on a copy 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 graphical definitions 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 3-4(b)).

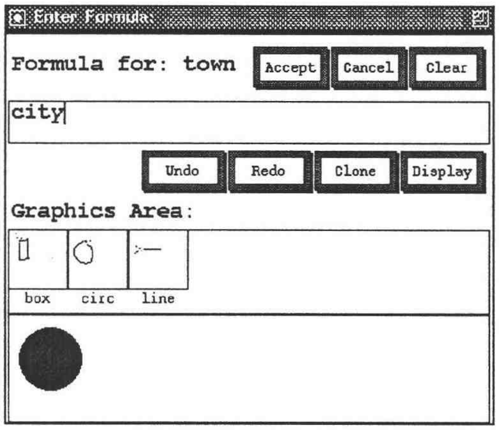




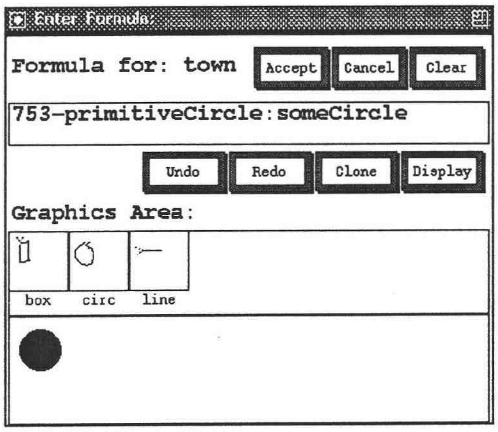
(a) (b)

Figure 3-4. 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.

The circles for cells *town* and *village* can be defined in the same way, or they can be defined by graphically specifying how they are different from the *city* circle, which already has the *fillForeColor* attribute defined. To define the *town* circle using the latter technique, 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 3-5). The population analyst then resizes the circle to define the *town* circle, which has all of the attributes of the *city* circle except its radius.



(a)



(b)

Figure 3-5. 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.

3.3 Graphical Definitions are Consistent with the Value Rule

Note that a graphical definition defines a reference to a *new* object, not a modification of an existing object. If direct manipulation were to modify the existing object, this would override the formulas of the cells that define its attributes, which would violate the spreadsheet value rule.

Instead, a graphical definition defines the creation of a *copy* of the definition form for the object (recall Figure 3-1), in which the cell formulas are defined to be the same as those that define the original object, except for any formulas defined by the manipulation itself. Thus, direct manipulation of a circle specifies a new circle whose radius is defined to be the radius of the manipulated circle, and whose other attributes are defined by the same formulas as on the original circle's definition form. This borrows from declarative textual languages the idea that the application of an operation to an object results in a new object, and is key to enabling the graphical definitions approach to fit seamlessly within the spreadsheet paradigm.

3.4 Using Gestures with User-Defined Types

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

Suppose the programmer wants to develop the binary search algorithm using a binary tree that was previously 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 3-6(a)). 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.

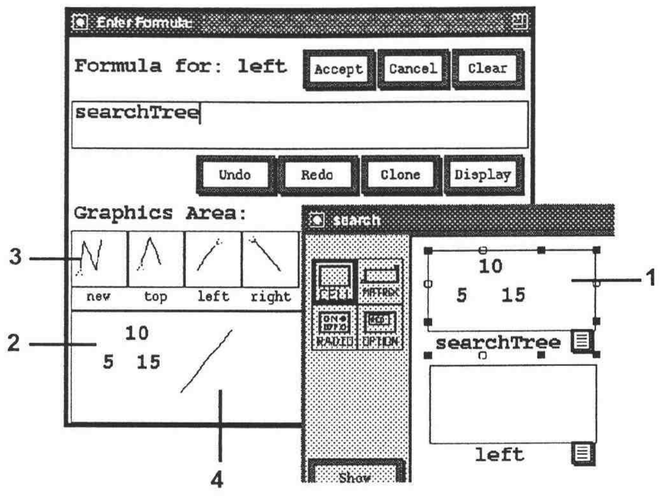


Figure 3-6. The programmer clicks on the (1) search tree to set the (2) context for the gesture. (3) Iconic representations of the tree gestures are automatically displayed. (4) The programmer then draws a gesture to reference the left subtree.

To program the search algorithm, the programmer can use graphical definitions to access different elements of the search tree. For instance, if the top element of the tree is greater than the search element, 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 [27].) 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 3-6). 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*[1]. 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."

[1] The tree definition form and formal semantics of graphical definitions will be discussed in detail in the next chapter.

4. The Semantics of Graphical Definitions

4.1 Graphical Types in Forms/3

Graphical definitions build upon previous work on graphical types in the spreadsheet paradigm [2]. The central philosophy of this work was that in a spreadsheet language, all on-screen cells' values are displayed, and therefore all types are in some sense graphical. Graphical types can be used to create such diverse applications as event-based programs [1], inventory tracking [2], a desktop analog clock application [24], exception handling [24], and algorithm animation [3].

In keeping with the philosophy that all types are graphical, in this work a type is the 4-tuple: (components, operations, graphical representations, interactive behaviors). In this model, there is no theoretical distinction between built-in and user-defined types; both are defined by the above 4-tuples. The only distinction is implementation; that is, whether the type's implementation has already been provided by the language implementor. We have extended the operations of a graphical type to include the graphical definitions defined for the type, which will be discussed further in sections 4.2 and 4.4.

To define a new type, a programmer uses a *type definition form* which, following the spreadsheet paradigm, consists of cells with formulas. The type definition form is the aspect of this work that graphical definitions directly affect. The form contains two distinguished cells: an *abstraction box,* which defines the structure of the type as the composition of its attributes (the first element of the 4-tuple); and an *image cell,* whose formula defines the type's appearance(s) (the third element of the 4-tuple). The operations and interactive behaviors are specified by additional abstraction boxes and ordinary (non-distinguished) cells on the form, in addition to the graphical definitions. An object's appearance is entirely flexible and can be based on its attributes, as demonstrated in the circle example. Each type has its own type definition form, and each object (instance of the type) has its own copy of the type definition form, upon which different formulas can be defined to allow individual differences among the objects. A discussion of information hiding and other type-related issues is omitted here, since it does not impact the graphical definitions technique presented in this thesis.

The circle form (Figures 3-1 and 3-2) is one example of a type definition form; because circles are a built-in type, the circle form is provided in the language implementation. By specifying formulas for the attribute cells, a new instance of a circle is defined in the abstraction box *newCircle.* Information about the instance of the type can be obtained by referencing cells on the definition form such as *radius* and *lineForeColor.*

4.2 Graphical Definitions for Built-In Types

We have implemented graphical definitions for the built-in types *box, circle,* and *line.* Graphical definitions allow these graphical objects to be instantiated and manipulated using gestures and direct manipulation, as shown in the example of Chapter 3. We have defined a gesture for each of these types, and a programmer can instantiate a new instance of the type by drawing the gesture or clicking on the gesture icon displayed in the formula edit window (recall Figure 3-4a). This action defines a formula that is a reference to an abstraction box on a copy of the definition form for the graphical type; the formal semantics of the defined formula are shown in Tables 4-1 and 4-2. The formulas for some of the cells on this definition form are defined by the attributes of the gesture itself: for instance, the circle gesture defines a reference to the abstraction box on a copy of the circle definition form in which the formula for cell *radius* is defined to be the radius of the drawn circle gesture. For objects that are instantiated by clicking on the gesture icon rather than by drawing a gesture, the defined formula is simply a reference to the abstraction box on the type definition form (with default formulas for the cells' formulas).

Gesture Attribute	**Value**
width	ω
height	η
radius	ρ
dx	ξ
dy	ψ

Table 4-1. Gesture attributes. The above notation is used in the tables in this chapter. An instance of a gesture is defined by the attributes and values shown in the table.

Graphical Type	**Action**	**Formula**
circle	draw gesture	primitiveCircle(*radius*☞ρ):*someCircle*
	click on gesture icon	primitiveCircle (*radius*☞25) :*someCircle*
box	draw gesture	primitiveBox(*width*☞ω, *height*☞η):*someBox*
	click on gesture icon	primitiveBox (*width*☞50,*height*☞50) :*someBox*
line	draw gesture	primitiveLine(*deltax*☞ξ, *deltay*☞ψ):*newLine*
	click on gesture icon	primitiveLine *(deltax*☞50,*deltay*☞50) :*someLine*

Table 4-2. Formulas defined by drawing gestures and clicking on gesture icons for built-in types. The formula notation is *FC*(*DefList*):*RC*, where *FC* is a copy of definition form *F*, *DefList* is a list of formula definitions for each cell that is defined differently on form *FC* than on *F*, and *RC* is the cell to be referenced on *FC*. The notation for each element of *DefList* is (*X*☞α), denoting that cell *X* has the formula α.

A new graphical object can also be created by directly manipulating an existing instance of the type, such as stretching the endpoint of a line or the edge of a circle. These manipulations, like the gestures described above, specify a reference to an abstraction box on a copy of the definition form for the graphical object. However, the formulas for all of the cells on the new definition form will be the same as those on the definition form for the object being manipulated, except for those formulas that depend on the attributes of the gesture itself (Table 4-3).

Graphical Type	Manipulation	Formula
circle	*stretch edge of circle*	primitiveCircle(*radius*☞ρ, cell$_\forall$☞cell$_\alpha$):*someCircle*
box	*stretch corner of box*	primitiveBox(*width*☞ω, *height*☞η, cell$_\forall$☞cell$_\alpha$):*someBox*
line	*stretch line endpoint*	primitiveLine(*deltax*☞ξ, *deltay*☞ψ, cell$_\forall$☞cell$_\alpha$):*someLine*

Table 4-3. Formulas defined by direct manipulation of an existing graphical object α. The notation *cell*$_\forall$☞*cell*$_\alpha$ denotes that for all cells X not specified explicitly in the table, the formula for cell X on FC is the same as the formula for cell X on F_α, where F_α is the definition form for object α.

4.3 Example: Defining a Binary Tree

For user-defined types, the programmer creates the type definition form, placing cells and abstraction boxes on it as needed and defining their formulas. Programmers will often use more than one abstraction box, placing an input abstraction box, other cells for input specifications and output information, and one or more output abstraction boxes on the definition form. Each abstraction box for a particular type definition form must contain the same set of cells, although they may have different formulas.

For example, a tree definition form (Figure 4-1) might contain an input abstraction box intended to contain an incoming tree, an input cell for an element to be inserted into the tree, and an output abstraction box that defines 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. For graphical definitions to be possible with such a type, a programmer needs a way to specify the set of graphical definitions for the type, enabling their use for purposes such as the binary search algorithm of the previous section.

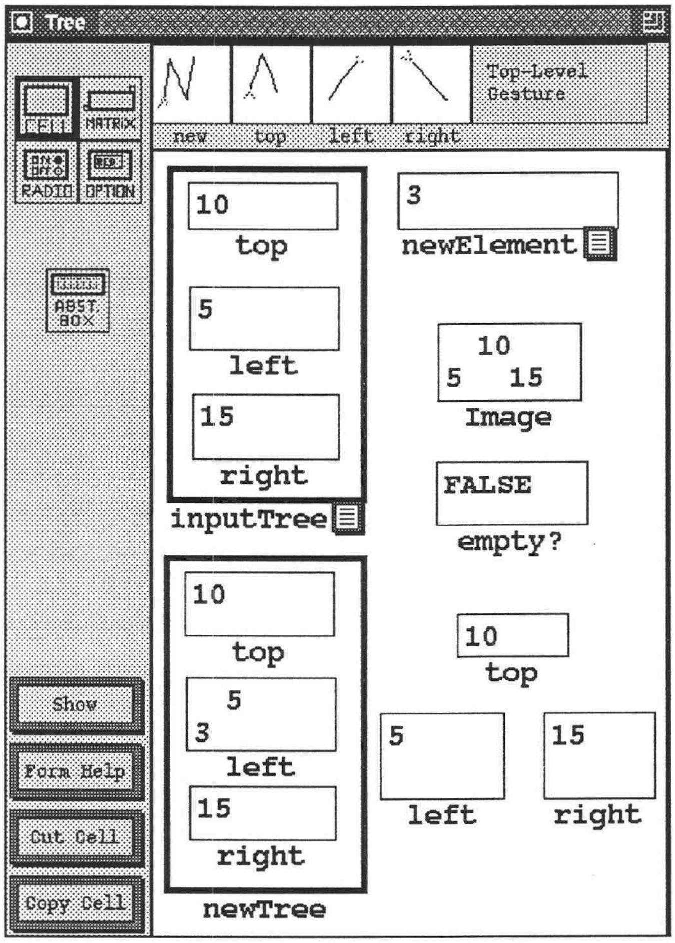


Figure 4-1. A tree definition form. The cells inside the abstraction boxes are by definition "hidden," and cannot be accessed by cells outside this form. The implementor of the tree has provided access cells such as *empty?* and *top* to report the values of the attributes of the incoming tree. The formula tabs on cells *newElement* and *inputTree* signify that these cells are intended for input. The formulas that define the cell values are not shown.

4.4 Defining 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. 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. Miniature gesture icons are automatically displayed at the top of the type definition form when Agate is exited.

After defining a gesture for the type, the programmer specifies the gesture's semantics. These specify the formula that will be defined when the gesture is drawn. For instance, the *new* tree gesture at the top of Figure 4-1 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.

To define the semantics of a gesture, the programmer specifies two things: the cell to be referenced, and formula specifications for each of the input cells on the definition form. (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 types of formula specification (defined formally in Tables 4-4 and 4-5):

- A *gesture attribute* formula specification for a cell means that the formula depends on some attribute of the gesture itself, such as its height, width or radius. For example, a programmer defining a gesture for a sectionHeading user-defined type to be used for formatting text might define the *gesture attribute* formula specification "height" for cell *size* (Figure 4-2a(**1**)).
- A *same* formula specification for a cell means that the formula for the cell on the new definition form is the same as that on the definition form of the object being manipulated (Figure 4-2a(**2**)).
- A constant formula specification depends only on the name of the gesture, and defines the new formula for the cell completely (Figure 4-2a(**3**)).

- An *askUser* formula specification means that the user will be asked to specify the formula for the cell after the gesture is drawn. The *new* tree gesture (Figure 4-2b) 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 4-3).

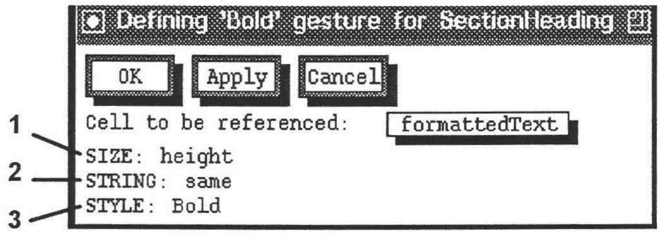
Type of formula specification	**Permissible values**	**Formula defined for cell *X***
gesture attribute	height[1]	η
	width	ω
	radius	ρ
	dx	ξ
	dy	ψ
same	same	X_α
constant	*anything*	same as formula specification value
askUser	ask "*string*"	anything (defined by user)

Table 4-4. The semantics of formula specifications. This table defines the formula that is defined for cell *X* on form *FC* when gesture *G* is applied to some graphical object α, where α is defined by definition form F_α, and the formula specification for cell *X* is given in the table. For the *askUser* formula specification, the keyword *ask* is followed by the prompt "string" that will be displayed when the user is asked to enter the formula.

[1] Agate provides many (17) primitive gesture attributes such as *minX*, *maxX*, and *initial-sin*, from which ours are defined. It would be simple to add more to our list (such as *angle*, perhaps), but the current set has been sufficient for our purposes.

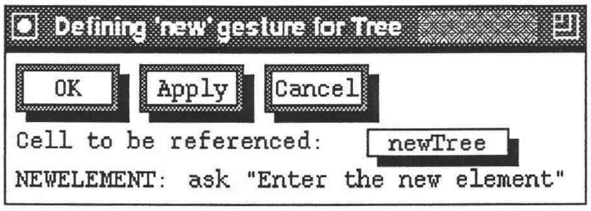
Action	Formula
draw gesture, click on gesture icon	FC(β☞α, cell$_{\forall}$☞formulaSpec$_{\forall}$):χ

Table 4-5. Formulas defined by drawing gestures and clicking on gesture icons for user-defined types. This table defines the formula that is defined by applying a graphical definition to some graphical object α. In the above notation, β and χ represent the input abstraction box and cell to be referenced, respectively, on definition form F_{α}. χ is defined by the gesture's semantics. The notation *cell*$_{\forall}$☞*formulaSpec*$_{\forall}$ denotes that for all cells X other than β, the formula for cell X on FC is defined by the formula specification for cell X, as defined in Table 4-4.



```
Cell to be referenced:  formattedText
SIZE: height
STRING: same
STYLE: Bold
```

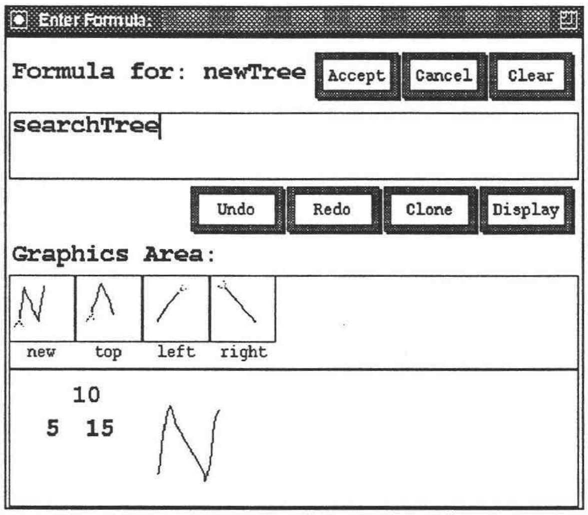
(a)

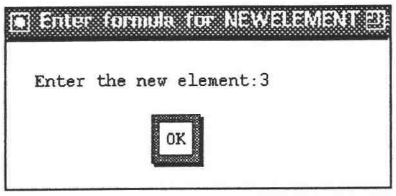


```
Cell to be referenced:  newTree
NEWELEMENT: ask "Enter the new element"
```

(b)

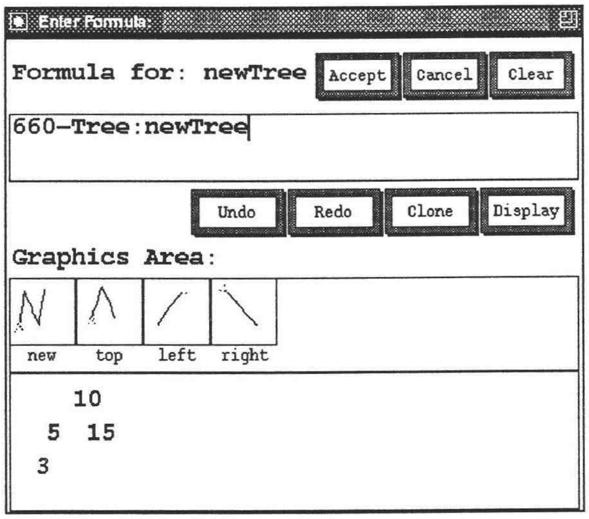
Figure 4-2. Defining gesture semantics. (a) The *bold* sectionHeading gesture defines a reference to cell *formattedText* on a copy of the sectionHeading definition form in which (1) the formula for cell *size* is defined to be the height of the drawn gesture, (2) *string* is defined to be the same as 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.





(a)

(b)



(c)

Figure 4-3. 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.

In addition to specifying gestures that manipulate an existing object, the programmer can specify a gesture to instantiate a new instance of the type that is not derived from any other instance of the type. The programmer presses the "top-level gesture" button on the type's definition form to edit the gesture, 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 new graphical objects as for manipulating them.

5. Other Contributions of the Approach

Here we elaborate on significant aspects of the approach that have not been covered fully in the rest of this thesis.

5.1 Gesture Spaces

Several researchers [4, 10] have discussed the need for context-dependent gestures. Landay and Myers [10] identify this as a problem to be solved: "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.

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 the same gestures to be reused in different contexts, while eliminating possible ambiguities over the meaning of a gesture. Thus, the set of allowable gestures for any context remains relatively small and recognizable even for large programs.

In the work of Gross and Do [4], as in ours, gestures are only applicable in certain contexts, but in their system the context is inferred and may not yet be defined at the time a particular gesture is drawn, and thus the meaning of the gesture may be ambiguous. Such ambiguities may be left unresolved until further information is added by the user . Since their system is intended to support conceptual and creative design, ambiguity may be an advantage because it supports the designer's creativity by allowing specific design choices to be deferred until some later time. In contrast, our approach is intended for programming, which is not compatible with ambiguity, and uses scope rules to determine the unique context which is current.

These scope rules, which determine which gestures are applicable, 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.

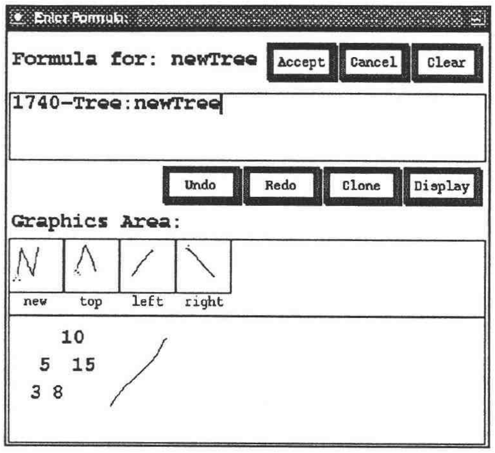
One problem that programmers in any programming language face is that of remembering the permissible operations on an object, and this problem is exacerbated if the operations are invisible gestures that must be memorized. Our approach addresses this problem by displaying miniature icons of the allowable 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. 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 graphical 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 [20].

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, consider the binary tree. The implementor of the tree type might test the correctness of the implementation 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; or a programmer wishing to use the type in a program might perform similar actions in order to better understand how to use the tree. Without graphical definitions, this process is straightforward but somewhat tedious: the programmer defines formulas for cells *inputTree* and *newElement,* creates another tree, defines its *inputTree* formula to be a reference to cell *newTree* from 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 5-1). Explorations like this for even a small tree with just 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.



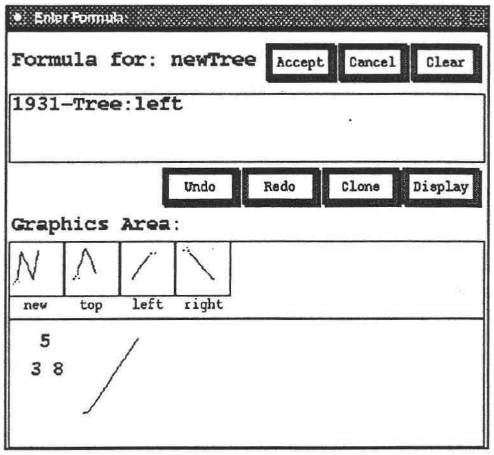
Formula for: newTree

1740-Tree:newTree

Graphics Area:

```
   10
 5  15
3 8
```

(a)



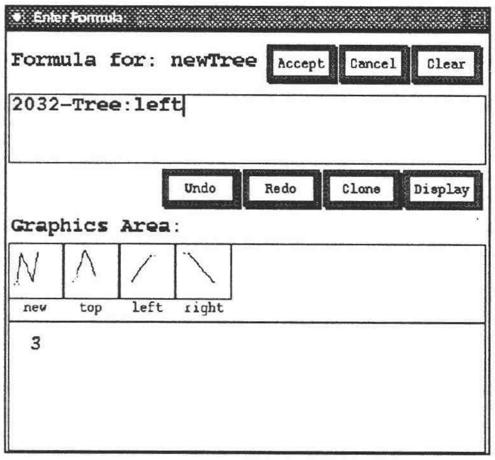
Formula for: newTree

1931-Tree:left

Graphics Area:

```
 5
3 8
```

(b)



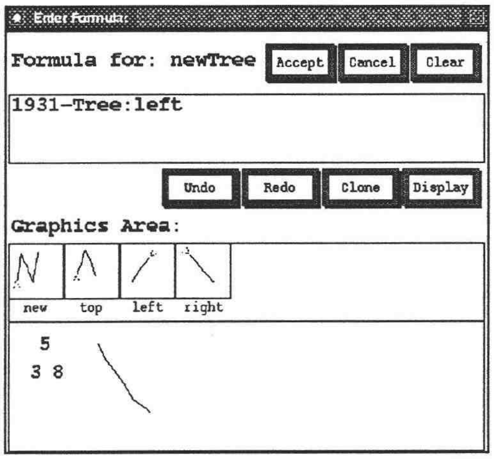
Formula for: newTree

2032-Tree:left

Graphics Area:

```
3
```

(c)



Formula for: newTree

1931-Tree:left

Graphics Area:

```
 5
3 8
```

(d)

Figure 5-1. 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.

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. To consider a small example, building the population visualization program shown in Figure 3-3 without graphical definitions would have required the programmer to copy the circle definition form and define the *radius* formula for each circle, as well as to define a reference to the circle from the population form, whereas graphical definitions required only a single copy of the definition form to define the fill color for the first circle. Although each graphical object specified with a graphical definition is defined by a definition form behind the scenes, only the graphical object itself is explicitly displayed onscreen; its definition form is only shown 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, memory, and computation time.

Perhaps even more important to the programmer is that graphical definitions reduce the amount of work required to create programs containing graphical objects (Table 5-1). Without graphical definitions, the programmer would have to copy type definition forms and create formulas defining the network of relationships among the cells on those forms. (Forms/3's multiple forms are similar to commercial spreadsheets' linked spreadsheets whose cells reference one another.) But with graphical definitions, programming with graphical objects is elevated from such low-level programming minutiae to the task-specific operations represented by each gesture.

Actions needed to create graphical objects *without* graphical definitions					
To create these graphical 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 graphical objects *with* graphical definitions					
To create these graphical 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 5-1. 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

In order to obtain empirical data on the usefulness of our approach, we conducted a user study. Among the questions we hoped to answer were the following:

- Do graphical definitions help programmers construct correct programs?
- Do graphical definitions help programmers construct programs more quickly?
- Do programmers using graphical definitions prefer to draw gestures or click on gesture icons?
- Do programmers enjoy using graphical definitions?

The study was conducted one subject at a time at the author's workstation. Each subject was given an introduction to Forms/3 programming, followed by instruction on how to create boxes using either graphical definitions or the "copying" technique shown in Figure 3-2. The subject was then asked to use the newly learned technique to create several colored circles in a larger program. This was followed by instruction in the second technique and a second programming task using user-defined types.

The study was counter-balanced with regard to the programming method involved; that is, each subject completed one of the programs using graphical definitions and the other using the copying technique. The same program was always performed first, which may have given the second program a learning advantage. However, because we did not assume that the problems were of equal difficulty, this did not affect the validity of the results. The data produced by the study included post-question and post-test questionnaires as well as notes and observations taken by the author during the study.

6.1 Details of the Empirical Study Procedure

The subjects were first given a brief hands-on introduction to Forms/3 in which they learned how to define simple formulas and reference cells in formulas. They were then taught how to create a box of a particular size and color using either graphical definitions or by copying the built-in box form and defining formulas for cells on that form. Half of the subjects were taught the copying technique, and the other half were taught graphical definitions. Each subject was given an information sheet describing the steps they took in constructing the example programs used in the instruction. They were allowed to refer to this information sheet when working on their assigned programs. They were then asked to complete a small program which required them to create three colored circles using the technique they had been taught (see Section 6.3). Upon completion of the program, they answered a questionnaire about their academic experience, previous exposure to Forms/3, and their confidence in the correctness of their solution.

They were then shown how to program using the user-defined binary tree type described in Section 4.3, either by making copies of the tree definition form and defining formulas for cells on those forms (if they had previously been taught graphical definitions), or using graphical definitions (if they had previously been taught the copying technique). They were then asked to instantiate a tree with several elements and reference its left subtree using the technique they had just been taught (see Section 6.3). After finishing this program, they were given another questionnaire in which they were asked questions about the tree problem as well as questions comparing the two problems and techniques. Appendix A contains all of the instructional materials and questionnaires used in the study.

6.2 Subjects

The subjects in the study were 20 computer science graduate students at Oregon State University. A summary of the subject backgrounds is shown in Table 6-1. Most of the subjects had little or no previous exposure to Forms/3. Two of them had done a small amount of Forms/3 programming in a visual programming course one and a half years earlier, but had not used graphical definitions or user-defined types. Four of the subjects were new or former members of our research group with little programming experience in Forms/3. Only one of these four had used graphical definitions previously, although several had seen it in demos. The subjects were assigned to one of the two groups randomly, except that the four from our research group were intentionally divided evenly between the groups to reduce possible bias that might occur if they were placed in the same group. There turned out to be no significant correlation between performance in this study and previous experience with Forms/3 (Fisher's exact test, p=0.406).

60% of the subjects in the study were Master's students, and the remaining 40% were Ph.D. students. Performance in the study was independent of degree status (Fisher's exact test, p=0.392). The students included both first year students (35%) and advanced students; performance was also independent with regard to number of years of graduate study (Fisher's exact test, p=0.474).

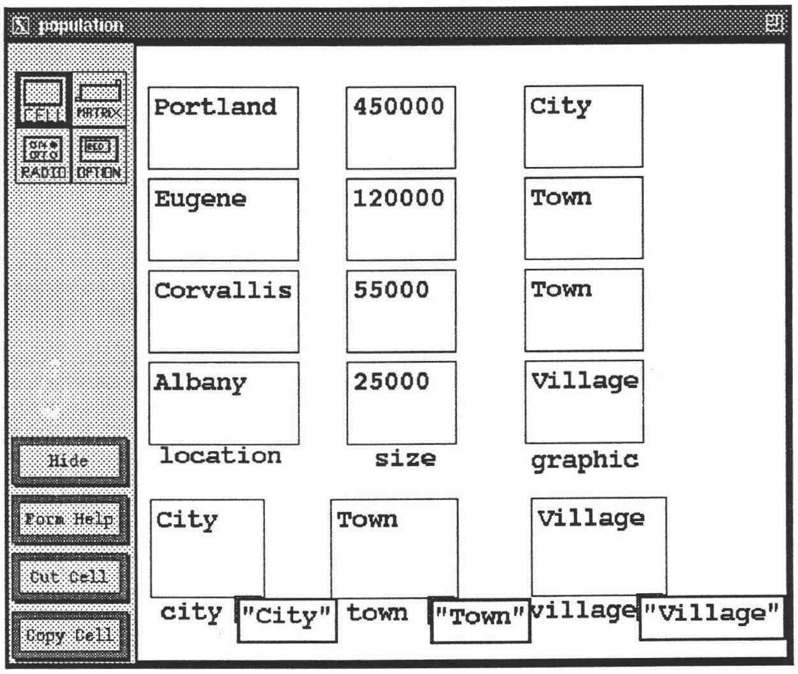
	Degree Status	**Years at OSU**	**Forms/3 experience**
	MS, Ph.D.	*mean, median*	*Never used it, Used it*
Graphical definitions first	6, 4	1.90, 2	9, 1
Copying technique first	6, 4	2.90, 2	6, 4
Cumulative	12, 8	2.40, 2	15, 5

Table 6-1. Summary of subject backgrounds. Performance on the study was independent with regard to degree status, years of graduate study and prior Forms/3 experience.

6.3 Programs

6.3.1 Population program

The first program completed by the subjects in the study is shown in Figure 6-1. In this program the subjects were to redefine the formulas for cells *city, town,* and *village* from textual representations to the graphical representations of black circles. Programming this with graphical definitions required the subject to draw a circle gesture or click on the circle gesture icon to define the first circle, resize the circle if necessary, display the circle's definition form by clicking the middle mouse button, and define the formula for the *fillForeColor* cell. The remaining circles could be programmed in the same way, or they could be programmed by clicking on the first circle and then resizing it to create a new circle, as described in Section 3.2.

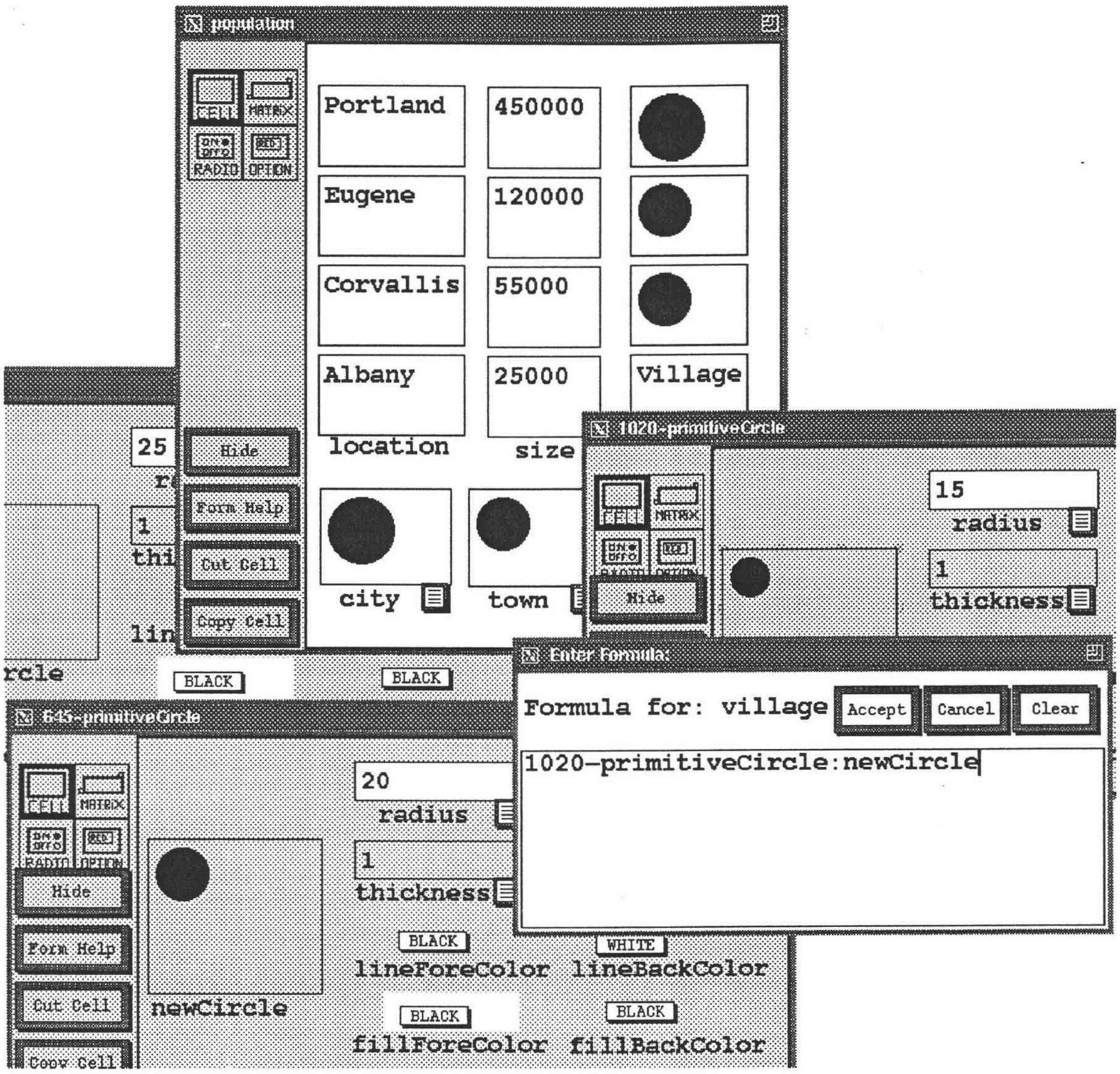


location	size	graphic
Portland	450000	City
Eugene	120000	Town
Corvallis	55000	Town
Albany	25000	Village

city	town	village
City	Town	Village
"City"	"Town"	"Village"

Figure 6-1. Population program (before programming). The subjects were asked to redefine the formulas of cells *city, town,* and *village*.

Programming the population program using the copying technique required the subject to make a copy of the primitiveCircle form for each circle and then define formulas for cells *radius* and *fillForeColor* (Figure 6-2).



Portland 450000
Eugene 120000
Corvallis 55000
Albany 25000 Village
location size
city town

15 radius
1 thickness

Formula for: village
1020-primitiveCircle:newCircle

20 radius
1 thickness
newCircle
BLACK lineForeColor
WHITE lineBackColor
BLACK fillForeColor
BLACK fillBackColor

Figure 6-2. Programming the population program using the copying technique. Each circle is defined by a copy of the primitiveCircle form on which formulas for cells *radius* and *fillForeColor* have been defined.

6.3.2 Tree program

In the second program, the subjects were asked to define the formulas for two cells: one that instantiates a tree containing three elements, and the other that reports the left subtree of that tree, as shown in Figure 6-3. To define the formula for cell *binTree* using the copying technique, the subject made several copies of the tree form, defined formulas for the *newElement* and *inputTree* cells on these forms, and referenced the *newTree* cell on the final copy of the form (see Figure 6-4). To define the formula for cell *left*, the subject made another copy of the Tree form, defined the *inputTree* formula to be a reference to cell *binTree,* and referenced cell *left* on that form. To define the formula for cell *binTree* using graphical definitions, the subject drew the top-level *tree* gesture or clicked on the *tree* gesture icon, then drew three *new* gestures and entered the values for each of the elements in the tree. To define the formula for cell *left,* the subject clicked on cell *binTree* and drew the *left* gesture (refer to Figure 3-6 for a similar example).

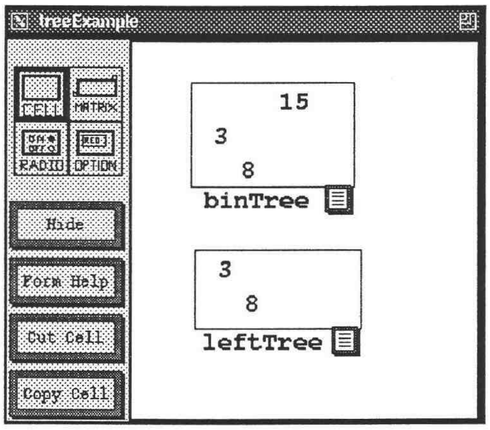


Figure 6-3. The output of the completed tree program.

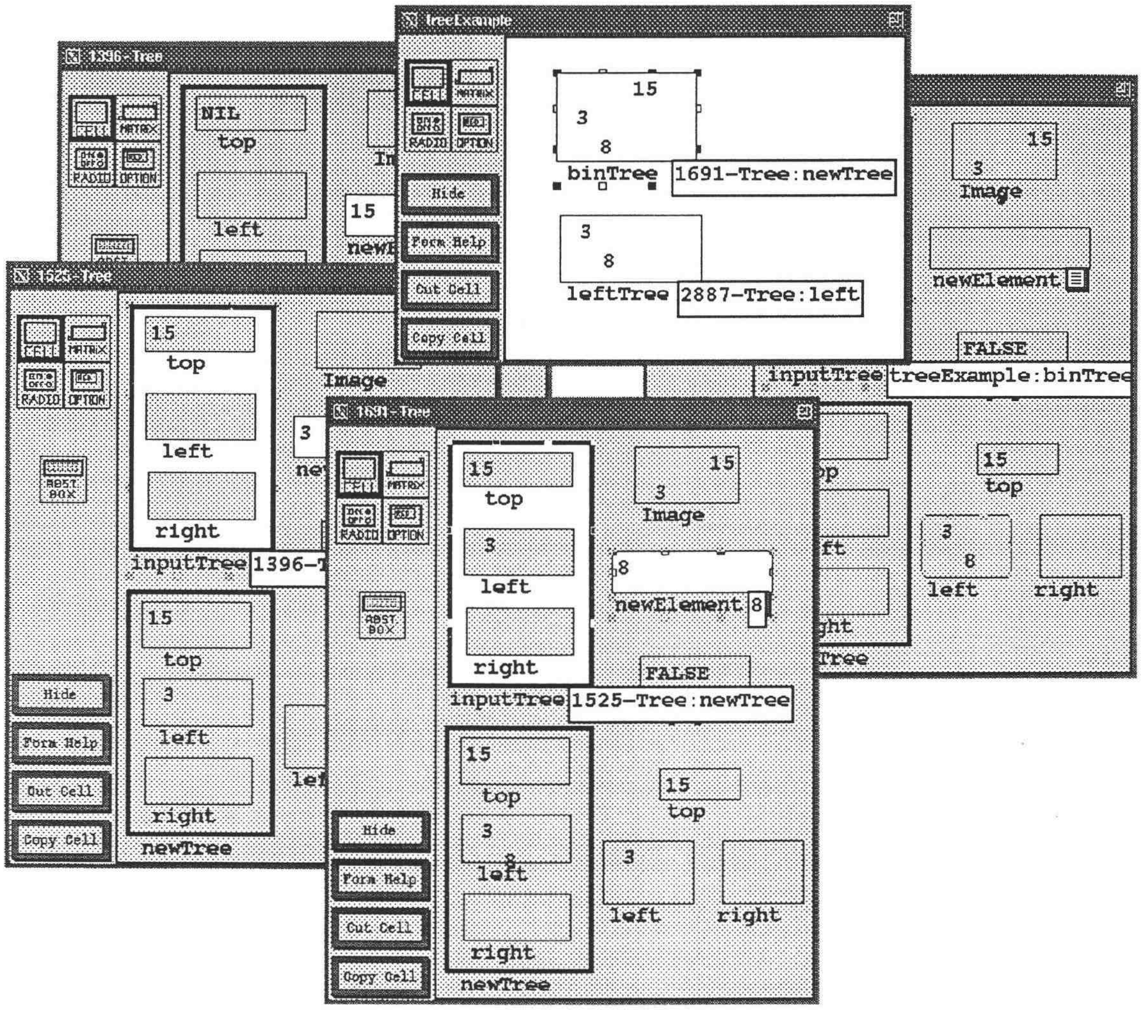


Figure 6-4. Defining formulas for the tree program using the copying technique. The forms 1396-Tree, 1525-Tree, and 1691-Tree are used to construct the tree shown in cell *treeExample:binTree*. The other form (2887-Tree) is used to report information about the tree in cell *binTree*; in this case, its left subtree.

A common error in this problem using the copying technique occurred in defining the formula for cell *leftTree*. Many subjects defined this formula to be a reference to cell *newTree*[*left*] on the third copy of the form (1691-Tree in Figure 6-4) rather than referencing cell *left* on a new form (2887-Tree) whose *inputTree* formula is a reference to cell *binTree*. Although this formula produces the correct answer, it violates the information hiding rules of Forms/3 by accessing private data in the *tree* abstraction box.

Our implementation of Forms/3 does not enforce these information hiding rules; however, the better solution (and the one taught to the subjects) is the one shown in Figure 6-4.

6.4 Results

We will discuss the results from the study as they relate to the questions listed at the beginning of this chapter.

6.4.1 Do graphical definitions help programmers construct correct programs?

A summary of the correctness results is shown in Table 6-2. All subjects—both those using graphical definitions and those using the copying technique—were able to complete the population program correctly. This is not particularly surprising since the formulas defined in this program were quite simple and the colored circles used in the program provided the subjects with a visual indication of the correctness of their formulas.

Significantly more subjects were able to complete the tree program correctly using graphical definitions than the copying technique (Fisher's exact test, p=0.03). Whereas 90% of the subjects using graphical definitions completed the program correctly, only 40% of the subjects using the copying technique did so. These results produced a significant difference in the cumulative results. The cumulative results show that significantly more programs were completed correctly with graphical definitions than with the copying technique (Fisher's exact test, p=0.05).

		Population		Tree		Total	
		n	%	*n*	%	*n*	%
Graphical Definitions	*Correct*	10	100%	9	90%	19	95%
	Incorrect	0	0%	1	10%	1	5%
Copying Technique	*Correct*	10	100%	4	40%	14	70%
	Incorrect	0	0%	6	60%	6	30%
Total	*Correct*	20	100%	13	65%	33	82.5%
	Incorrect	0	0%	7	35%	7	17.5%

Table 6-2. Program correctness. All subjects were able to complete the population problem correctly. 65% of the subjects completed the tree program correctly: 90% of the subjects who used graphical definitions, and 40% of the subjects who used the copying technique.

6.4.2 Do graphical definitions help programmers construct programs more quickly?

We measured the amount of time it took each subject to complete each program. Both programs were completed significantly faster using graphical definitions (population: Mann-Whitney test, $p<.02$; tree: Mann-Whitney, $p<.002$). These results were the most dramatic for the tree program. In fact, each of the subjects who used graphical definitions completed the tree program faster than *any* of the subjects who used the copying technique on that program. Table 6-3 and Table 6-4 contain detailed and summary results.

Subject #	Population time	Technique	Subject #	Tree time	Technique
11	75	Graphical Definitions	2	60	Graphical Definitions
7	95	Graphical Definitions	6	60	Graphical Definitions
13	135	Graphical Definitions	12	75	Graphical Definitions
17	160	Graphical Definitions	10	90	Graphical Definitions
4	240	Copying	4	105	Graphical Definitions
19	240	Graphical Definitions	18	105	Graphical Definitions
15	285	Graphical Definitions	14	120	Graphical Definitions
3	300	Graphical Definitions	8	180	Graphical Definitions
6	300	Copying	20	255	Graphical Definitions
9	300	Graphical Definitions	16	300	Graphical Definitions
1	330	Graphical Definitions	13	330	Copying
8	420	Copying	7	390	Copying
5	435	Graphical Definitions	17	390	Copying
10	450	Copying	19	450	Copying
2	555	Copying	3	480	Copying
12	600	Copying	11	540	Copying
16	615	Copying	9	600	Copying
20	705	Copying	15	720	Copying
18	825	Copying	5	870	Copying
14	840	Copying	1	900	Copying

Table 6-3. Program completion time (detail, in order of program completion time). All times are in seconds. Both programs were completed significantly faster using graphical definitions than with the copying technique.

	Population			Tree			Both
	mean	*median*	*sd*	*mean*	*median*	*sd*	*mean*
Graphical Definitions	235.5	262.5	115.9	135.0	105.0	83.4	370.5
Copying	555.0	577.5	204.3	567.0	510.0	202.2	1122.0

Table 6-4. Program completion time (summary). All times are in seconds. Note that median and standard deviation are not meaningful in the rightmost column (*Both*) because no single subject was assigned both programs using either graphical definitions or the copying technique.

6.4.3 Do programmers prefer to draw gestures or click on gesture icons when using graphical definitions?

We were also interested in determining which graphical definitions input technique the subjects preferred, and whether this depended on the problem they had solved using graphical definitions. We asked the question, "When you used gestures, did you prefer to draw the gesture or click on the gesture icon?" The possible answers to this question on the questionnaire were drawing the gesture, clicking on the gesture icon, or using both techniques. A little over half of the subjects (55%) said they preferred to click on the gesture icon, while most of the others said they preferred to use both techniques (χ^2=7.90, df=2, p<.02). Only one subject preferred solely to draw the gesture (Table 6-5).

		Problem done using graphical definitions		
		Population	*Tree*	*TOTAL*
Preferred method	*draw*	0	1	1
	click	7	4	11
	both	3	5	8

Table 6-5. The subjects' preferred method of using graphical definitions.

A greater, although not statistically significant, number of subjects who used graphical definitions in the population program preferred solely to click compared to those who used graphical definitions in the tree program (Fisher's exact test, $p=0.18$). We can speculate that this apparent trend may be related to difficulties some subjects encountered in drawing circle gestures in the population program and box gestures in the preceding training. Several subjects in this group had significant difficulty in using the mouse to draw gestures at all, and others initially drew gestures that were incorrectly recognized by the gesture recognizer as boxes or lines. This may have contributed to some subjects' reluctance to use the circle gestures in the population program. These gesture recognition errors did not occur with the tree gestures, which, unlike the circle gesture, all consisted of straight line segments. More experimentation would be useful to explore these results further.

6.4.4 Do programmers enjoy using graphical definitions?

In the post-test questionnaire we asked two questions pertaining to this question: "If you were able to choose either approach to use on a third problem, which would you use?" and "Which problem did you feel more comfortable working on?" A majority of subjects who specified a preference, although not statistically significant (69%, $\chi^2=2.25$, $df=1$, $p<0.14$), said they would choose graphical definitions to use on a third problem. A significant majority of subjects (79%, $\chi^2=6.37$, $df=1$, $p<0.02$) said they felt more comfortable on the problem on which they used graphical definitions. As we would expect, the answers to these two questions were highly correlated (Fisher's exact test, $p=0.004$). See Table 6-6.

		More comfortable on program using			
		Graphical Definitions	Copying	(No answer)	Total
Would rather use on a third program	*Graphical Definitions*	10	0	1	11
	Copying	1	4	0	5
	No preference	4	0	0	4
	Total	15	4	1	20

Table 6-6. User reaction to graphical definitions.

6.5 Analysis of Programming Difficulties

In order to be able to perform a more detailed analysis of these results, we also looked at some of the difficulties the subjects encountered while working on the programs. One surprising result we discovered was that, in the population program, many of the subjects using graphical definitions did not take full advantage of the capabilities of graphical definitions. Although with graphical definitions the subjects could first define a black circle for cell *city* and then define the black circles for cells *town* and *village* entirely by direct manipulation (we refer to this as "cloning" in Table 6-7 below), only slightly more than half of the subjects (56%) used this technique. The others used graphical definitions to define a circle of the correct size, but then had to display the circle's definition form to define the formula for its *fillForeColor* cell. This required more work and time than cloning, and may have indicated a partial lack of understanding of graphical definitions. Subjects who mastered the cloning technique tended to do significantly better on the population program (Mann-Whitney test, $p<.01$). They also performed better on the tree program (Mann-Whitney, $p<.10$). This may be an indication that, even for built-in types, the benefits of graphical definitions may increase for advanced programmers as opposed to graphical definitions being a tool useful only to novices.

		Population program	
		Used cloning	*Didn't use cloning*
Tree program	*Correct*	4	0
	Incorrect	2	4

Table 6-7. Correlation between cloning on the population program and correctness on the tree program. The subjects who used cloning on the population program tended to do better on the tree program as well.

On the tree program, only one subject using graphical definitions (10%) had any problems in defining the formulas for cell *binTree*, and only one (10%) had difficulty with cell *left*. The subjects using the copying technique, on the other hand, encountered several difficulties. When instantiating the tree, several subjects made errors in defining the relationships between cells on the various copies of the tree definition form. Many of these errors were later corrected, in part because the subjects could see that the formulas they had defined were not producing the results they expected. However, in defining the formula for cell *left*, this continuous immediate feedback may actually have contributed to some errors: a few subjects defined incorrect formulas (such as creating a new tree with the elements 3 and 8) that *looked* correct but did not define the correct relationship. Table 6-8 contains a more detailed summary of the difficulties the subjects encountered on this problem.

In Table 6-9 we consider the number of subjects who made conceptual errors such as the ones described above on the tree program. Significantly more errors were made when the subjects used the copying technique than when they used graphical definitions (Fisher's exact test, p=0.001). These results, along with Table 6-2, suggest that in addition to helping programmers construct ultimately correct programs, graphical definitions can also help programmers make fewer errors along the way. This may be because graphical definitions are so closely related to the task-specific operations defined for the graphical type; they allow the programmer to concentrate on the task to be done rather than on the low-level details of copying forms and defining formulas for cells, activities in which the study subjects made numerous errors.

		binTree	**left**	**entire program**
		(number of subjects)		
Graphical Definitions	*none*	9	9	8
	minor	1	0	1
	major	0	1	1
Copying	*none*	3	3	2
	minor	3	2	3
	major	4	5	5

Table 6-8. Difficulties encountered on tree program. This table shows a categorization of the number of subjects who encountered difficulties as they were defining formulas for cells *binTree* and *left* in the tree program. Minor difficulties consisted of incorrect formulas that were quickly discovered and corrected. Major difficulties included numerous minor errors or errors that were never corrected.

	Programs with errors	**Programs without errors**
Graphical definitions	2	8
Copying	8	2

Table 6-9. Number of subjects who made conceptual errors on the tree program.

6.6 Summary of Results

We obtained the following results from the study. First, using graphical definitions subjects completed significantly more programs correctly than with the copying technique, and they had fewer difficulties in completing the programs. Second, subjects completed the programs faster using graphical definitions than with the copying technique. Third, most subjects seemed to prefer clicking on gesture icons to drawing gestures, although many subjects used both techniques. Fourth, the subjects were more comfortable using graphical definitions than the copying technique. These results demonstrate the usefulness of graphical definitions, and show that graphical definitions can be an effective technique for constructing programs with graphical types.

7. Implementation

The work described in this thesis has been implemented in our Forms/3 research prototype, which runs on Sun and Hewlett-Packard color workstations using Harlequin Common Lisp and the Garnet user interface development environment [16]. The gesture training facility is provided by Garnet's Agate package [10].

The major functions in the Lisp code used to implement graphical definitions in Forms/3 are shown in Appendix B.

8. 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.

The graphical definitions described in this thesis solve this problem. Among the other benefits of graphical definitions are the following:

- The contextual display of the gestures valid for the formula being edited and the use of gesture spaces to keep the number of such operations manageable eliminate the need for the programmer to memorize gestures, solving a common problem of gestural interfaces.
- Support for exploratory programming is enhanced, because graphical definitions have immediate effects and can be experimented with rapidly (using only a few mouse strokes) and reversibly.
- Program scalability is increased because fewer system resources are required, and because with graphical definitions programmers can specify fewer formulas and fewer cell references than are required without graphical definitions, sometimes reducing these actions by as much as a factor of *n*.

We have also evaluated graphical definitions in an empirical study. The results of this study show that programmers using graphical definitions can construct programs with graphical types faster and with fewer errors than with a less direct technique.

Most important, this work demonstrates 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, allowing graphical types to be promoted to first-class status in spreadsheet languages.

References

[1] Burnett, M. and A. Ambler, "Generalizing Event Detection and Response in Visual Programming Languages," *International Workshop AVI '92: Advanced Visual Interfaces,* Rome, Italy, May 27–29, 1992, 334-347.

[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] Carlson, P., M. Burnett, and J. J. Cadiz, "A Seamless Integration of Algorithm Animation into a Visual Programming Language," *ACM Proceedings of Advanced Visual Interfaces '96,* Gubbio, Italy, May 27–29, 1996 (to appear).

[4] Gross, M. and E. Do, "Ambiguous Intentions: A Paper-Like Interface for Creative Design," *ACM Symp. on User Interface Software and Technology,* Seattle, WA, Nov. 6-8, 1996, 183-192.

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

[6] 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.

[7] 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.

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

[9] Kurlander, D., "Chimera: Example-Based Graphical Editing," in *Watch What I Do: Programming by Demonstration* (A. Cypher, ed.), MIT Press, Cambridge, MA (1993), 271-290.

[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] Lewis, C., "NoPumpG: Creating Interactive Graphics with Spreadsheet Machinery," in *Visual Programming Environments* (E. Glinert, ed.), IEEE Computer Society Press, Los Angeles, CA (1990), 526-546.

[12] Lieberman, H., "Mondrian: A Teachable Graphical Editor," in *Watch What I Do: Programming by Demonstration* (A. Cypher, ed.), MIT Press, Cambridge, MA (1993), 341-358.

[13] *Microsoft Excel User's Guide,* Microsoft Corporation, 1993-1994.

[14] 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.

[15] 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.

[16] Myers, B., D. Giuse, R. Dannenberg, B. Vander Zanden, D. Kosbie, E. Pervin, A. Mickish, P. Marchal, "Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces," *IEEE Computer,* Nov. 1990, 71-85.

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

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

[19] Shneiderman, B., "Direct Manipulation: A Step Beyond Programming Languages," *IEEE Computer,* Aug. 1983, 57-69.

[20] Shneiderman, B., Designing the User Interface: Strategies for Effective Human-Computer Interaction, Addison-Wesley, Reading, MA (1992), 181-233.

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

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

[23] Vander Zanden, B. and B. Myers, "Demonstrational and Constraint-Based Technologies for Pictorially Specifying Application Objects and Behaviors," *ACM Trans. on Computer-Human Interaction,* Dec. 1995, 308-356.

[24] van Zee, P., M. Burnett, and M. Chesire, "Retire Superman: Handling Exceptions Seamlessly in Declarative Visual Programming Languages," *1996 IEEE Symposium on Visual Languages,* Boulder, Colorado, September 3–6, 1996 (to appear).

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

[26] 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.

[27] 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.

Appendices

Appendix A. Empirical Study Materials

A.1 Introduction to Forms/3

This is the script that was read to all subjects as their introduction to Forms/3. They were instructed to perform the actions described in the script.

This study has two parts in which you'll be doing some programming in the spreadsheet-based visual programming language Forms/3. Before each section, you'll be given instructions on how to complete the task that I'll ask you to do.

For the first part, I'll give you a brief hands-on introduction to programming in Forms/3, and then demonstrate one way of programming graphical objects such as boxes and circles. I'll then ask you to modify an existing program to use these graphical objects.

In Forms/3, programming is done by defining formulas for cells which are placed on windows called forms, such as the "demoForm" you see here. For these examples, you won't be creating any new cells, but you will be defining formulas for existing cells. To define a formula for a cell, double click on the formula tab shown in the bottom right hand corner of the cell. Go ahead and define the formula of cell *X* to be the number 15. (Post the formula)

We can reference this cell in another formula. To define the formula for cell *Y* to be *X*, either type the name of the cell, or click on the cell. Go ahead and do this. Now, if we change *X*'s formula, *Y*'s value changes as well. For instance, change *X*'s formula to 10. Press the "Clear" button to erase the current formula.

A.2 Group 1 (graphical definitions first)

A.2.1 Introduction to graphical definitions for built-in graphical objects

This was read to the subjects who programmed the population program using graphical definitions.

Graphical objects can be created and manipulated by gestures. For instance, to define the formula for cell *aBox* to be a box, first double click on the formula tab, and then either draw a gesture in the shape of a box or click on the box gesture icon. Go ahead and do this. The box can be resized by grabbing and dragging its corners.

Circles are resized by dragging the top right, bottom right, etc., and lines are stretched by dragging their endpoints.

These graphical objects are actually defined by built-in definition forms that contain cells. To display the definition form for an object, click on it with the middle mouse button. Then press the accept button to close the formula edit window. If we define new formulas for the cells on the primitiveBox form, the box changes. For instance, to make the box a solid color, define a formula for the *fillForeColor* cell. Make the box green. (Hit display or accept)

New objects can also be created from existing objects. For instance, to create a box that is just like some other box but smaller, click on the existing box while editing the new formula, and resize it to the desired size. This defines a new box that has all of the attributes of the original box except its size. Define the formula for *anotherBox* to be a green box that is smaller than the green box you just created.

Are there any questions before you start the first program?

A.2.2 Information sheet for population program using the copying technique

This was given to the subjects for their reference during the introduction above and while they were completing the population program.

- to define a formula for a cell:

  double click on the formula tab
- to refer to another cell in a formula:

  click on that cell while editing the formula
- to create a new box:

  double click on the formula tab to bring up the formula edit window.

  draw the box gesture or click on the box gesture icon.
- to resize a box:

  drag the corners of the box to the desired size.
- to display the box definition form:

  press the middle mouse button in the graphics area
- to make the box solid green:

  define a formula for the *fillForeColor* cell on the primitiveBox form.
- to create a new (different sized) box from another box:

  click on the box.

  stretch the box to the desired size.

A.2.3 Population program

This was read to all subjects prior to beginning the population program.

Now I'd like you to use the approach you've just learned to program circles in the following program. This is a visualization of population data for various cities. Different sized cities are represented by the strings "city", "town", and "village". I'd like you to change these strings to a graphical representation of solid black circles of varying sizes, as shown in this figure (see Figure A-1). You will need to replace the formulas for these three cells with formulas that represent black circles of approximately the size shown here, using the approach to programming graphical objects that I've just demonstrated.

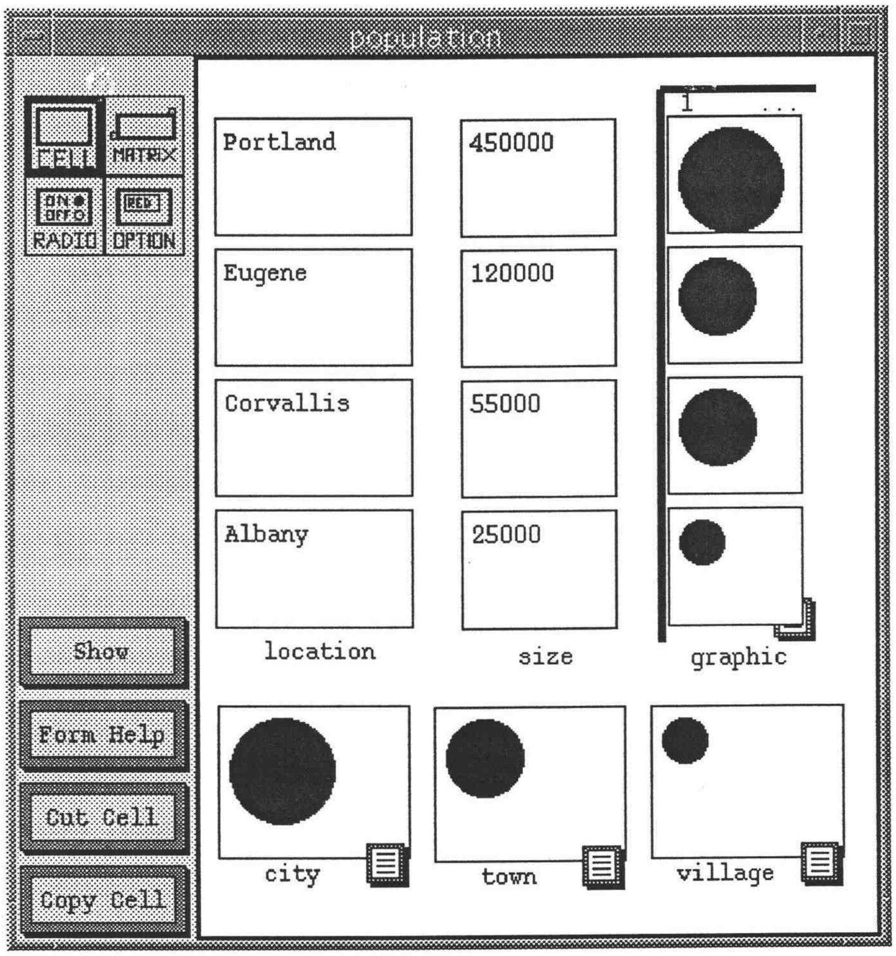


Figure A-1. The output of the completed population program. This handout was given to the subjects prior to beginning the population program.

A.2.4 Introduction to the copying technique for user-defined graphical objects

This was read to the subjects who programmed the tree program using the copying technique.

In this example, you're going to create a small binary tree, using a user-defined Tree data type that has already been created. This tree is defined by a form similar to the built-in primitiveBox form you saw earlier.

To define a new tree, the first thing you'll do is make a copy of the Tree form. To do this, first select the Tree form in the list and then press the "Copy Form" button. The Tree form contains several "abstraction boxes", which contain cells. These are used for data abstraction, and are similar to structs in C. To create a new tree, define a formula for the *newElement* cell. The abstraction box *newTree* now represents a tree containing the new element. This tree can be used in a program by referencing the *newTree* cell on this form. Let's do this now: create a tree with the element 10, and define the formula for *aTree* on the treeDemo form to be a reference to this tree.

We can create a new tree that is exactly like another tree but with some change (such as a new element) by defining a formula for the *inputTree* abstraction box. For instance, to create a tree that is the same as the first tree but with the element 5 added to it, first copy the Tree form, then define the formula for the *inputTree* abstraction box on this form to be a reference to the *newTree* cell on the previous tree's definition form. Now define the formula 5 for the *newElement* cell. The *newTree* abstraction box is now a tree just like the previous tree but with the element 5 added. We can now define the formula for *anotherTree* to be a reference to the *newTree* abstraction box on this form.

The tree form also contains other cells to perform operations on the tree. For instance, if we define a new tree just like the one in the *anotherTree* cell, we can access its top element and left and right subtrees. Define the formula for cell *treeTop* to be the top element of the tree defined in cell *anotherTree* by copying the Tree form, defining the formula for cell *inputTree* to be *anotherTree*, and defining the formula for cell *treeTop* to be the *top* cell on the Tree form.

Are there any questions before you start the second program?

A.2.5 Information sheet for tree program using graphical definitions

This was given to the subjects for their reference during the introduction above and while they were completing the tree program.

- to define a new tree with the element 10:

  select the Tree form

  press the "Copy Form" button

  define the formula '10' for the *newElement* cell

  reference the *newTree* cell of the Tree form

- to define a new tree with the element 5 added to an existing tree:

  select the Tree form

  press the "Copy Form" button

  define the formula for the *inputTree* cell to reference the existing tree

  define the formula '5' for the *newElement* cell

  reference the *newTree* cell of the Tree form

- to access the top element of an existing tree

  select the Tree form

  press the "Copy Form" button

  define the formula for the *inputTree* cell to reference the existing tree

  reference the *top* cell of the Tree form

A.2.6 Tree program

This was read to all subjects prior to beginning the tree program.

Now I'd like you to use the approach you've just learned to complete the following exercise. Define the formula for cell *binTree* on the treeExample form to be a binary tree with the elements 15, 3, and 8. Define the formula for *leftTree* to be the left subtree of *binTree* (see Figure A-2).

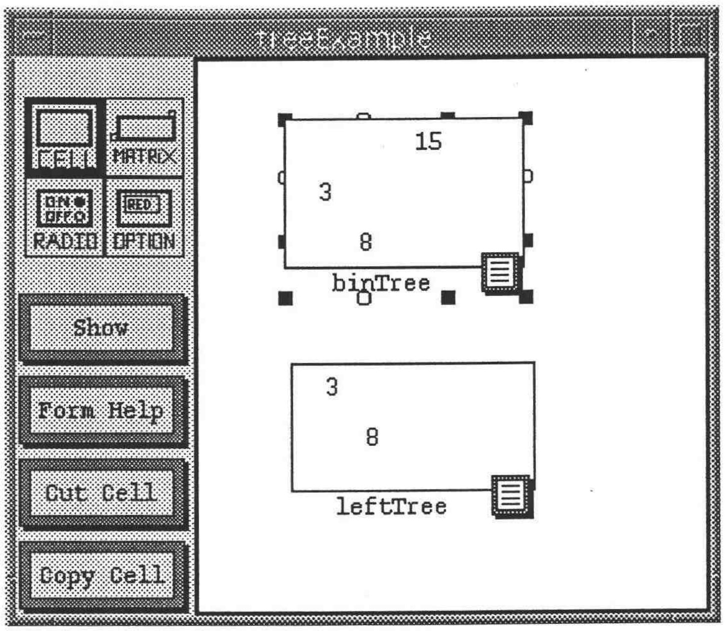


Figure A-2. The output of the completed tree program. This handout was given to the subjects prior to beginning the tree program.

A.3 Group 2 (copying technique first)

A.3.1 Introduction to the copying technique for built-in graphical objects

This was read to the subjects who programmed the population program using the copying technique.

Programming with graphical objects is done similarly, with a few new concepts. For instance, the first step in creating a box is to make a copy of the built-in box form. This is done by first selecting the primitiveBox form and then pressing the "Copy Form" button. Go ahead and do this now. The box in cell *someBox* is defined by the formulas of cells on this form. If we define new formulas for these cells, the box changes. For instance, make the box narrower by changing the formula for the *width* cell to 40. Go ahead and do this. The *width* cell changes to white to signify that it has been changed.

This box can be used in a program by referencing it in a formula. Define the *aBox* cell on the demoForm to be a reference to the box you have just created.

Other cells on the box form affect other attributes of the box. To make the box a solid color, define a formula for the *fillForeColor* cell. Make the box green.

Are there any questions before you start the first program?

A.3.2 Information sheet for population program using the copying technique

This was given to the subjects for their reference during the introduction above and while they were completing the population program.

- to define a formula for a cell:

  double click on the formula tab
- to refer to another cell in a formula:

  click on that cell while editing the formula
- to create a new box:

  select the primitiveBox form

  press the "Copy Form" button
- to make the box solid green:

  define a formula for the *fillForeColor* cell on the primitiveBox form.

A.3.3 Population program

The instructions for the population program for this group were identical to those given in section A.2.3.

A.3.4 Introduction to graphical definitions for user-defined graphical objects

This was read to the subjects who programmed the tree program using graphical definitions.

In this example, you're going to create a small binary tree, using a user-defined Tree data type that has already been created. Although a tree is defined by a form similar to the built-in primitiveBox form you saw earlier, we will be programming the tree using gestures rather than by copying forms.

To define a cell's formula to be a tree, first double click on the cell's formula tab as before. Then either draw the Tree gesture (a capital T), or click on the Tree gesture icon. This defines the formula for the cell to be an empty tree. You can now define a new formula that represents an operation on the tree. The displayed gesture icons signify the operations that have been defined for the tree.

To create a tree with a new element added, draw the New gesture or click on the New gesture icon. You will be asked to enter the new element. Enter the value 10. Press the "Accept" button. The formula for cell *aTree* is a tree with the single element 10.

A new tree can be created from an existing tree. For instance, to define cell *anotherTree* to be a new tree with the element 5 added to the tree you just created, first double click on the formula tab for *anotherTree*. Then click on cell *aTree*. Now draw the New gesture, and enter the value 5. And we can perform more operations on this new tree, such as adding another element to it, say 8.

Other operations on the tree are performed in the same manner. To define *treeTop* to be the top element of the tree you just created, click on the *anotherTree* cell while editing the formula for treeTop, then draw the Top gesture.

Are there any questions before you start the second program?

A.3.5 Information sheet for tree program using graphical definitions

This was given to the subjects for their reference during the introduction above and while they were completing the tree program.

- to create a new tree with the element 10:

  double click on the formula tab to bring up the formula edit window.

  draw the "Tree" gesture or click on the "tree" gesture icon.

  draw the "New" gesture or click on the "new" gesture icon.

  enter the value 10.

- to define a new tree with the element 5 added to an existing tree:

  double click on the formula tab to bring up the formula edit window.

  click on the tree

  draw the "New" gesture or click on the "new" gesture icon.

  enter the value 5.

- to access the top element of an existing tree:

  double click on the formula tab to bring up the formula edit window.

  click on the tree

  draw the "Top" gesture or click on the "top" gesture icon.

A.3.6 Tree program

The instructions for the tree program for this group were identical to those given in section A.2.6.

A.4 Questionnaires

Questionnaire 1 was given to the subjects after they completed the first program.
Questionnaire 2 was given to the subjects after they completed the second program.

A.4.1 Questionnaire 1

Subject #:___

Answer these questions after finishing the first part

1. How many years have you been at OSU?

____ 1st year ____ 2nd year ____ 3rd year ____ other (specify)

2. Are you a Master's or PhD student?

____ Master's ____ PhD ____ Unsure

3. Describe your experience with Forms/3.

____ Never heard of it ____ Know a little, but haven't used it

____ Have used it a little ____ Have used it a lot

4. Did you do the first program using gestures or by copying the primitiveCircle form?

____ gestures ____ copying

5. Did you think that your solution to the problem was correct?

____ yes ____ no ____ not sure

6. How confident were you of your results on this problem? Why?

____ very confident ____ somewhat confident ____ not confident

A.4.2 Questionnaire 2

Subject #:___

Answer these questions after finishing the second part

1. Did you do the second program using gestures or by copying the primitiveCircle form?

    ____ gestures ____ copying

2. Did you think that your solution to the problem was correct?

    ____ yes ____ no ____ not sure

3. How confident were you of your results on this problem? Why?

    ____ very confident ____ somewhat confident ____ not confident

4. Which problem did you feel more comfortable working on? Why?

    ____ population ____ tree

5. If you were able to choose either approach to use on a third problem, which would you use? Why?

    ____ gestures ____ copying

6. When you used gestures, did you prefer to draw the gesture or click on the gesture icon?

    ____ draw the gesture ____ click on the icon ____ used both

7. Any other comments?

THANK YOU VERY MUCH FOR YOUR PARTICIPATION!

Appendix B. Source Code

B.1 gesture.lisp

The file *gesture.lisp* contains the definitions of the *gesture* class and *gestureTableEntry* structure. These structures store information about user-defined gestures and the semantics of these gestures. This file also contains functions to manipulate these structures. The most important functions are shown below.

B.1.1 *gesture* class

```
(deferrals gesture (displayable)
  ((id :accessor displayable-id :initarg :id :initform nil)
   (name :accessor displayable-name :initarg :name
                                         :initform nil)
   (attributeList :accessor displayable-attributeList
              :initarg :attributeList :initform nil)
  )
  (:documentation "Gesture class.")
)
```

B.1.2 *gestureTableEntry* struct

```
(defstruct gestureTableEntry
  (vadtName nil)
  (gestureName nil)
  (returnCell nil)
  (items nil)
)
```

B.1.3 gesture-create-constr-list

```
;;; ------------------------------------------------------
;;; gesture-create-constr-list
;;;
;;; creates a constr-list and returns the selected cell from a
;; gesture that contains the appropriate cell fmlas for
;;; theVADTType.

; NOTE: only supports direct fmlas: cell defined by
; single attrib.
;; theAttribs is (list (cellName, logicalVal))
;; returns (list (cellID, absoluteVal))

(defun gesture-create-constr-list (theVADTType gesture)
  (let* ((theEntry (find-gestureTableEntry theVADTType
                              (displayable-name gesture)))

        (theAttribs (if theEntry
                        (gestureTableEntry-items theEntry)))
        (returnCell (if theEntry
                        (gestureTableEntry-returnCell theEntry)))
        (model (formtable-find theVADTType))
        tempConstrList val win)

     (setf win (car (gv (displayable-win model) :child)))
     (setf tempConstrList
         (mapcar #'(lambda (attrib)
                  (setf val (read-from-string (cadr attrib)))
                  (if (not (equalp 'same val))
                  (progn
                    (if (not (stringp val))
                        (if (equalp val 'ask)
                          (setf val (get-val-from-prompt
                                  (subseq (cadr attrib) 4)
                                  (car attrib)))
                        (setf val (displayable-getAttribute val
                            (displayable-attributeList
                                          gesture)))))
                    (if (null val)
                        (setf val (parse-formula-string
                                      (cadr attrib) win)))
                    (if (not (null val))
                        (cons (car
                              (displayable-getNameEntry model
                               (format nil "~a" (car attrib))))
                            (list '<- val)))
                    )))
              theAttribs))
     (values (remove-if #'null tempConstrList) returnCell)
))
```

B.1.4 edit-gesture-attributes

```
(defun edit-gesture-attributes (theVADTType gestureName)
  (let* ((title (format nil "Defining '~a' gesture for ~a"
                  gestureName theVADTType))
         (model (formtable-find theVADTType))
         (currentEntry (find-gestureTableEntry
                          theVADTType gestureName))
         ;; the currently defined attributes
         (currentAttributes (if currentEntry
                          (gestureTableEntry-items currentEntry)))
         (currentReturnCell (if currentEntry
                          (gestureTableEntry-returnCell
                                    currentEntry)))
         (allCells (remove-if #'null
                     (displayable-mapcells
                      #'(lambda (cellid cell)
                         (if (not (or (roobj-formulatab-hidden?
                                     (displayable-object cell))
                                    (string-equal cellid
                                          (displayable-absBoxId
                                                     model))))
                              (roobj-name (displayable-object cell))))
                      (displayable-celltable model))))
         (returnCells (sort
                      (stable-merge-no-dups
                     (list theVADTType)
                     (remove-if #'null
                        (displayable-mapcells
                         #'(lambda (cellid cell)
                            (declare (ignore cellid))
                            (if (roobj-formulatab-hidden?
                                  (displayable-object cell))
                                 (roobj-name (displayable-object
                                                     cell)))))
                         (displayable-celltable model)))
                     #'string-equal #'string-lessp) #'string-lessp))
         (allCellAttributes (mapcar #'(lambda (cellName)
                            (list (read-from-string cellname)
                                                    "same"))
                                 allCells))
         allAttributes returnCellListObj returnCell)

     (setf returnCell (if currentReturnCell currentReturnCell
                      (let* ((absBoxIds
                             (remove-if #'null
                                  (displayable-mapcells
                                   #'(lambda (cellid cell)
                                       (if (typep cell 'absBox)
                                         cellid))
                                   (displayable-cellTable
                                                     model)))))
                     (roobj-name (displayable-object
                                          (displayable-getCell
                           model
                           (if (= 2 (length absBoxIds))
                             (car (remove-if #'(lambda (cellid)
                                 (string-equal
                                       (displayable-absBoxId model)
```

```
                                          cellid)) absBoxIds))
                            (displayable-absBoxId model)))))))
    (setf currentAttributes (sort currentAttributes
                     #'(lambda (a b) (string-lessp (car a)
                                                          (car b)))))
    (setf allCellAttributes (sort allCellAttributes
                     #'(lambda (a b) (string-lessp (car a)
                                                          (car b)))))
    ;; combine both sets of attributes
    (setf allAttributes (stable-merge-no-dups
                    currentAttributes allCellAttributes
                    #'(lambda (a b) (string-equal (car a)
                                                          (car b)))
                    #'(lambda (a b) (string-lessp (car a)
                                                          (car b)))))

    ;; re-sort so that "same" comes last
    (setf allAttributes (stable-sort (copy-list allAttributes)
           #'(lambda (a b) (and (not (string-equal "same" a))
                          (string-equal "same" b))) :key #'cadr))

    (setf returnCellListObj (create-instance nil
                                        gg:option-button
                              (:items returnCells)
                              (:label "")
                              (:initial-item returnCell)))
    (setf allAttributes (cons `("Cell to be referenced"
                         ,returnCellListObj)
                         allAttributes))


    (s-value $GestureEditor :vadtType theVADTType)
    (s-value $GestureEditor :gestureName gestureName)
    (gg:pop-up-win-change-items $GestureEditor allAttributes
                                               200 200 title T)
    )
)
```

B.1.5 define-gesture

```
(defun define-gesture (an-interactor final-obj-over)
  (declare (ignore an-interactor))
  (agate:do-go
   :initial-classifier (gv final-obj-over :parent :classifier)
   :initial-examples (gv final-obj-over :parent :items)
   :final-function #'(lambda (filename classifier examples
                               saved trained)
                    (declare (ignore filename saved trained))
                    (let* ((win (gv final-obj-over :window))
                         (agg (displayable-aggregate win))
                         gestAgg)
                    (setf gestAgg
                              (displayable-gestureIconAgg win))
                    (opal:remove-component agg
                              (displayable-gestureIconAgg win))
                    (setf gestAgg
                     (create-instance nil opal:aggregadget
                                          (:parts
                                           `((:topLevel ,opal:text
                                              (:left ,(o-formula
                                                (+ 10
                                                    (opal:gvl-sibling :topLevelBorder
                                                                      :left))))
                                              (:top 10)
                                              (:font ,opal:default-font)
                                              (:string "Top-Level
                   Gesture")

                                              )
                                              (:toplevelborder ,opal:rectangle
                                               (:left ,(o-formula
                                                    (+ (opal:gvl-sibling :icons :left)
                                                       (opal:gvl-sibling :icons
                                                                         :width))))
                                               (:top 0)
                                               (:height 50)
                                               (:width 110))
                                              (:icons ,(create-instance nil
                                                          $GestureIconAgg
                                                          (:classifier classifier)
                                                          (:items examples)))))
                                          (:interactors
                                           `((:addTop ,inter:button-interactor
                                               (:window ,(o-formula
                                                    (gv-local :self :operates-on
                                                                         :window)))
                                               (:active T)
                                               (:start-event :any-mouseDown)
                                               (:start-where ,(o-formula
                                                          (list :in (gvl :operates-on
                                                                :topLevelBorder))))
                                             (:final-function
                                                   ,#'define-toplevel-gesture))))
                                          ))
                                        (displayable-set-gestureIconAgg win
                                                                gestAgg)
                                        (opal:add-component agg gestAgg)
                                        ))
                      )
                   )
```

B.1.6 define-toplevel-gesture

```
(defun define-toplevel-gesture (an-interactor final-obj-over)
  (declare (ignore an-interactor final-obj-over))
  (agate:do-go
   :initial-classifier $GestureClassifier
   :initial-examples $GestureExamples
   :final-function #'(lambda (filename classifier examples
                                         saved trained)
                       (declare (ignore filename saved trained))
                       (setf $GestureClassifier classifier)
                       (setf $GestureExamples examples))
   )
)
```

B.2 formulaDM.lisp

The file *formulaDM.lisp* contains functions to support the use of direct manipulation for editing formulas. The functions that are specifically related to graphical definitions are shown below.

B.2.1 handle-gesture

```
;;; ---------------------------------------------------------
;;; handle-gesture
;;;
(defun handle-gesture (gesture-name attribs)
  (if (or (null attribs)
        (< 9 (+ (abs (- (inter:gest-attributes-minx attribs)
                        (inter:gest-attributes-maxx attribs)))
                (abs (- (inter:gest-attributes-miny attribs)
                        (inter:gest-attributes-maxy attribs))))))
      (let* ((tempFormulaText (gv $FormulaGadget :formulaText))
             (newFormulaString nil))
      (gv tempFormulaText :string)
      (gv tempFormulaText :cursor-index)
      (case (gv $FormulaGadget :drawObj)
            (:userDefined
             (if (null gesture-name)
               (format T "Gesture not recognized")
             (let* ((tempFormDrawWindow
                     (gv (formula-tab-get-obj
                          (gv $FormulaWindow :formulaTab))
                         :parent :window))
                    (parsed-formula (parse-formula-string
                                     (gv tempFormulaText :string)
                                     tempFormDrawWindow))
                    modelForm theFmla theCell theVADTType
                    theGesture selectCell tempConstr)
               (if (cellref-p parsed-formula)
                   (progn
                   (multiple-value-setq
                        (modelForm theFmla theCell)
                    (formula-model parsed-formula
                                   tempFormDrawWindow))
                   (setf theVADTType
                        (displayable-name
                                   (formtable-find modelForm)))
            (setf theGesture
                        (make-gesture gesture-name attribs))
                   (multiple-value-setq (tempConstr selectCell)
                        (gesture-create-constr-list
                         theVADTType theGesture))

                   (if (typep (formtable-find modelForm)
                                               'VADTForm)
                    (if selectCell  ;; a recognized gesture
                     (progn
    ;; need to create a constr entry for the
    ;; distinguished absBox unless the formula
    ;; already refers to the model's
    ;; distinguished absBox
    (if (not (and (string-equal
              (cellref-form (car
       (cellrefs-replace-constr-names-by-ids
              (list parsed-formula))))
  (displayable-id (formtable-find modelForm)))
              (string-equal
               (cellref-cellID parsed-formula)
               (displayable-absBoxID
                (formtable-find modelForm))))
           )
;; need to get a ref to an absBox. if theCell
;; is an absBox or evaluates to an absDycon
;; (demand if necessary), then use it
;; otherwise, get the distinguished absBox
      (let* (absBoxConstr theValue)
        (if (typep theCell 'absBox)
            (setf absBoxConstr
                `((,(displayable-absBoxID
                   (formtable-find modelForm))
                       <- ,parsed-formula)))
          (progn
            (if (null
                 (displayable-object theCell))
              (setf theValue
                 (displayable-cell-demand
                 theCell
                 (formtable-find modelForm)))
            (setf theValue
              (roobj-value
               (displayable-object theCell))))
            (if (typep theValue 'absDycon)
              (setf absBoxConstr
                 `((,(displayable-absBoxID
                   (formtable-find modelForm))
                       <- ,parsed-formula)))
            (setf absBoxConstr
              `((,(displayable-absBoxID
                 (formtable-find modelForm))
                 <- ,(make-cellref modelForm
                      (displayable-absBoxID
                       (formtable-find
                             modelForm)
                       )))))
            )))
        (setf tempConstr
           (merge-constr-lists
            (constr-name-list
             (unravel-self-refs
        (make-constr-name :list absBoxConstr)
              (displayable-title
               (displayable-parent
                tempFormDrawWindow))))
            tempConstr))
        ))

    (setf newFormulaString
          (make-vadt-ref-fmla modelForm tempConstr
                                      selectCell)))
          (format T "Gesture not defined for ~a~%"
                theVADTType))
         (setf newFormulaString
             (make-prim-ref-fmla modelForm tempConstr
                        (gv tempFormulaText :string))))
        )
      ;; else
  (break "error - not a cellref in handle-gesture"))
     )))
 (T
  (if (null attribs)
    (setf newFormulaString
        (if (string-equal gesture-name "BOX")
            "box 50 50"
          (if (string-equal gesture-name "CIRC")
              "circle 25"
            (if (string-equal gesture-name "LINE")
              "line 50 50"
              (if (formtable-find gesture-name)
                (format nil "~a:~a" gesture-name
                   (roobj-name (displayable-object
                    (displayable-getcell
                     (formtable-find gesture-name)
                     (displayable-absBoxID
                (formtable-find gesture-name))))))
       (format nil "Form for gesture ~a not found"
                                 gesture-name))
              ))))
  ;; else, attribs not null
  (let* ((gesture
             (make-gesture gesture-name attribs)))
    (setf newFormulaString
        (if (string-equal gesture-name "BOX")
            (format nil "box ~a ~a"
              (displayable-getAttribute 'width
                (displayable-attributeList gesture))
              (displayable-getAttribute 'height
                (displayable-attributeList gesture)))
          (if (string-equal gesture-name "CIRC")
              (format nil "circle ~a"
              (displayable-getAttribute 'radius
                (displayable-attributeList gesture)))
            (if (string-equal gesture-name "LINE")
              (format nil "line ~a ~a"
              (displayable-getAttribute 'dx
                (displayable-attributeList gesture))
              (displayable-getAttribute 'dy
                (displayable-attributeList gesture)))
          (if (string-equal gesture-name "NIL")
        (format T "Sorry, gesture not recognized~%")
              (if (formtable-find gesture-name)
                (format nil "~a:~a" gesture-name
                  (roobj-name (displayable-object
                   (displayable-getcell
                    (formtable-find gesture-name)
                    (displayable-absBoxID
              (formtable-find gesture-name)))))))))))
    ))))
```

```
          (if newFormulaString
              (progn
                (s-value tempFormulaText :string newFormulaString)
                (s-value tempFormulaText :cursor-index
                       (length newFormulaString))
                (display-graphical-formula)
                ))
          )
        )
  )
```

B.2.2 create-graphical-formula

```
;;; ---------------------------------------------------------
;;; create-graphical-formula
;;;
;;; objType is :drawRect, :drawCircle, or :drawLine;
;;; return fmlaString

(defun create-graphical-formula (objType obj-being-changed
                                          final-points)
(let* (newFormulaString w h)
    (case objType
        (:drawRect
         (setf w (third final-points))
         (setf h (fourth final-points))
         (setf newFormulaString (format nil "box ~a ~a" w h)))

        (:drawCircle
         (setf w (third final-points))
         (setf h (fourth final-points))
         (setf newFormulaString (format nil "circle ~a"
                                  (round (/ (max w h) 2)))))

        (:drawLine
         (progn
           (setf w (- (third final-points)
                        (first final-points)))
           (setf h (- (fourth final-points)
                        (second final-points)))
           ;; move to upper left corner of gesture window
           (if (<= 0 (* w h)) ; both coordinates have same sign
            (s-value obj-being-changed :points
             (list 10
                (+ 10 (gv $FormulaGadget :gestureBorder :top))
                   (+ 10 (abs w))
                (+ 10 (abs h) (gv $FormulaGadget :gestureBorder
                                                      :top))))
             (s-value obj-being-changed :points
            (list 10
             (+ 10 (abs h)
                        (gv $FormulaGadget :gestureBorder :top))
             (+ 10 (abs w))
             (+ 10 (gv $FormulaGadget :gestureBorder :top)))))
           (setf newFormulaString
                                (format nil "line ~a ~a" w h))))
            )
        newFormulaString)
    )
```

B.2.3 handle-userDefined-formula

```
(defun handle-userDefined-formula (theForm theFmla
                                  obj-being-changed final-points)
;; the new formula for this cell is theFmla;
;; its model is theForm.
  (let* (newCopy newFormulaString)
  (case (car theFmla)
      (make-boxDycon
       (case (length theFmla)
             (3
            (setf newFormulaString (create-graphical-formula
                  :drawRect obj-being-changed final-points))
            )
             (4
            (setf newCopy (form-create-from-constr-list
                        theForm
                  `(("Box50" <- ,(gv obj-being-changed :width))
                  ("Box51" <- ,(gv obj-being-changed :height)))))

            (setf newFormulaString (format nil "~a:someBox"
                                        (displayable-id newCopy)))
            )
             (T
            (break "unknown box parameters"))
             ))
      (make-circleDycon
       (case (length theFmla)
             (2
            (setf newFormulaString (create-graphical-formula
                  :drawCircle obj-being-changed final-points)))
             (3
            (setf newCopy (form-create-from-constr-list
                   theForm
                   `(("Circle50" <-
                    ,(round (/ (gv obj-being-changed :height) 2))
                         ))))

            (setf newFormulaString (format nil "~a:someCircle"
                                        (displayable-id newCopy)))
            )
             (T
            (break "unknown circle parameters")))
      )
      (make-lineDycon
       (case (length theFmla)
             (3
            (setf newFormulaString (create-graphical-formula
                  :drawLine obj-being-changed final-points)))
             (4
            (setf newCopy (form-create-from-constr-list
                         theForm
                   `(("Line50" <- ,(- (gv obj-being-changed :x2)
                                          (gv obj-being-changed :x1)))
                             ("Line51" <- ,(- (gv obj-being-changed :y2)
                                          (gv obj-being-changed :y1)))
                                   )))
                  (setf newFormulaString (format nil "~a:someLine"
                                              (displayable-id newCopy)))
                  )
                   (T
                  (break "unknown line parameters")
                  ))
             )
             (abscompose
              (format T
             "Working on DM (RESIZE) of a VADT (Not yet implemented)")
              (let* ((theVADTType (displayable-name
                                             (formtable-find theForm))))
                (setf newFormulaString
                    (make-vadt-ref-fmla theForm
                  (DM-create-constr-list theVADTType obj-being-changed
                                           final-points)))
                ))
             (T
     (break "unknown formula type in handle-userDefined-formula")
              (setf newFormulaString "")))

       newFormulaString
     ))
```

B.2.4 form-create-from-constr-list

```
     (defun form-create-from-constr-list (theForm theList)
       (if (not (formtable-find theForm))
           ;; load it
           (file-load nil (concatenate 'string $FormsDir
                     (format nil "primitiveForms/~a.frm" theForm)))
         )
       (let* (newCopy tempConstrList tempConstr)
         (if (displayable-win (formtable-find theForm))
           (if (model-form-p
              (displayable-win (formtable-find theForm)))
             ;; copying a model; straightforward
             (setf tempConstr (make-constr-name
                            :name theForm
                            :list theList))
           ;; else, copying a (visible) copy
           (let* ((tempComponentList (gv (displayable-aggregate
                                   (car (gv (displayable-win
                             (formtable-find theForm)) :child)))
                                 :components))
                 (overriddenComponentList (remove-if-not
                                            #'(lambda (component)
                                  (roobj-override component))
                                            tempComponentList)))

             ;; create constrlist from all overridden cells
             (setf tempConstrList (mapcar #'(lambda (component)
                   (cons (displayable-id (roobj-formsRO component))
                              (roobj-formula component)))
                                   overriddenComponentList))

           ;; if theList has any values that are different from
           ;; tempConstrList, use them.  First, sort the lists.

           (setf theList (sort theList #'(lambda (a b)
                                   (string-lessp (car a) (car b)))))
          (setf tempConstrList (sort tempConstrList #'(lambda (a b)
                                   (string-lessp (car a) (car b)))))

           (setf tempConstrList (stable-merge-no-dups theList

tempConstrList
        #'(lambda (a b) (string-equal (car a) (car b)))
        #'(lambda (a b) (string-lessp (car a) (car b)))))

           (setf tempConstr (make-constr-name
                   :name (form-model-title (displayable-constr-name
                                     (formtable-find theForm)))
                        :list tempConstrList))
           ))
        ;; else parent isn't visible, so the constr-list is just a
        ;; merge of the parent's constr-list and theList.
        (progn
          (setf tempConstrList (constr-name-list
                                 (displayable-constr-name
                                       (formtable-find theForm))))

          (setf theList (sort theList #'(lambda (a b)
                                  (string-lessp (car a) (car b)))))
          (setf tempConstrList (sort tempConstrList #'(lambda (a b)
                                 (string-lessp (car a) (car b)))))

          (setf tempConstrList
                (stable-merge-no-dups theList tempConstrList
                  #'(lambda (a b) (string-equal (car a) (car b)))
                  #'(lambda (a b) (string-lessp (car a) (car b)))))

          (setf tempConstr (make-constr-name
               :name (form-model-title (displayable-constr-name
                                   (formtable-find theForm)))
                :list tempConstrList))
          ))

       ;; if a form with the given constr-name already exists,
       ;; use it rather than creating a new one.
       (if (null (setf newCopy (formtable-find tempConstr)))
         (setf newCopy (form-create-from-abs-ref tempConstr)))
newCopy         ; return the new form
   ))
```

B.2.5 display-graphical-formula

```
;;; -----------------------------------------------------------
;;; display-graphical-formula
;;;
```

```
(defun display-graphical-formula ()
(if $Gesture
  (let* ((tempFormula (gv $FormulaGadget :formulaText :string))
       tempOperator stringIndex w h
       (tempObj (formula-tab-get-obj
                          (gv $FormulaWindow :formulaTab)))
       (tempFormDrawWindow (gv tempObj :parent :window))
       (parsed-formula
          (parse-formula-string tempFormula tempFormDrawWindow))
       theValue tempImage modelForm cellFmla tempCell
       )

    ;;hjghjg may need to add destroy-objects call(s) as well.
    (opal:remove-all-components
                  (gv $FormulaGadget :drawAgg :userDefined))
    (display-gesture-icons nil)

    (if (cellref-p parsed-formula)
      ;; handle as cellRef
      (progn
        (multiple-value-setq (modelForm cellFmla tempCell)
             (formula-model parsed-formula tempFormDrawWindow))
        (if tempCell
            (if (null (displayable-object tempCell))
              (setf theValue (displayable-cell-demand tempCell
                                 (formtable-find modelForm)))
            (setf theValue (roobj-value
                               (displayable-object tempCell))))
          )

        (setf tempImage
            (if (not (or (typep theValue 'eventReceptorDycon)
                         (typep theValue 'glyphDycon)
                         (listp theValue)))
                (displayable-display theValue)
              (create-instance nil opal:text))
            )

        (if (and (is-a-p tempImage opal:text)
         (not (typep (formtable-find modelForm) 'VADTForm)))
            ;; not a graphical object
            (s-value $FormulaGadget :drawObj nil)
          (progn
            (move-to-drawing-area tempImage)
            (opal:add-component
                         (gv $FormulaGadget :drawAgg :userDefined)
                           tempImage)
            (s-value $FormulaGadget :drawObj :userDefined)
            ))

        (if (typep theValue 'absDycon)
            (progn
      (display-gesture-icons (displayable-absType theValue))
            (s-value $FormulaGadget :drawObj :userDefined))
          ;; else
          (if (typep (formtable-find modelForm) 'VADTForm)
            (display-gesture-icons (displayable-name
                                 (formtable-find modelForm)))
            ;; else
            (display-gesture-icons nil))
            )

          (if (and (cellref-form parsed-formula)
                (null (displayable-win (formtable-find
                              (cellref-form parsed-formula)))))
              (progn
              (roobj-set-general-formula (formula-tab-get-obj
                              (gv $FormulaWindow :formulaTab))
                              `(<- ,(make-cellref
                                   (expand-refs
                                    (displayable-constr-name
                                    (formtable-find
                                     (cellref-form parsed-formula)))
                                    (displayable-parentFormID
                               (roobj-formsRO (formula-tab-get-obj
                                                (gv $FormulaWindow
                                                 :formulaTab)))))
                                (cellref-cellid parsed-formula))))
              (format T "Got a ref to an undisplayed form; ")
              (format T "setting generalized formula to ~a~%"
                    (roobj-general-formula (formula-tab-get-obj
                              (gv $FormulaWindow :formulaTab))))
              ))
          )
        ;; else (not a cellRef)
        (if (or (equal tempFormula "")
    (search ":" tempFormula)) ; read-from-string bombs on ":",
                              ; and we want to ignore it anyway.
          (s-value $FormulaGadget :drawObj NIL)
        (progn
          (multiple-value-setq (tempOperator stringIndex)
                            (read-from-string tempFormula))
          (setf tempOperator (format nil "~a" tempOperator))
          ;; so we can treat it as a string

          (cond ((string-equal tempOperator "BOX")
                (multiple-value-setq
                 (w stringIndex)
                 (read-from-string tempFormula nil nil
                                              :start stringIndex))
                (setf h (read-from-string tempFormula nil nil
                                     :start stringIndex))

                (if (and (numberp w) (numberp h))
                (progn
                    (s-value $FormulaGadget :drawAgg :drawRect :box
                           (list 0 0 w h))
                      (s-value $FormulaGadget :drawObj :drawRect))
                  (s-value $FormulaGadget :drawObj nil))
                )

              ((string-equal tempOperator "CIRCLE")
                (setf w (read-from-string tempFormula nil nil
                                     :start stringIndex))
                (if (numberp w)
                    (progn
                     (setf w (* w 2))
                     (s-value $FormulaGadget :drawAgg
                                         :drawCircle :box
                           (list 0 0 w w))
              (s-value $FormulaGadget :drawObj :drawCircle))
                (s-value $FormulaGadget :drawObj nil))
              )

             ((string-equal tempOperator "LINE")
              (multiple-value-setq
               (w stringIndex)
               (read-from-string tempFormula nil nil
                                                 stringIndex))
              (setf h (read-from-string tempFormula nil nil
                                  :start stringIndex))
              (if (and (numberp w) (numberp h))
                (progn
                  (if (<= 0 (* w h))
                         ; both coordinates have same sign
                    (s-value $FormulaGadget :drawAgg
                                            :drawLine :points
                     (list 10
                   (+ 10 (gv $FormulaGadget :gestureBorder :top))
                         (+ 10 (abs w))
                         (+ 10 (abs h)
                      (gv $FormulaGadget :gestureBorder :top))))
              (s-value $FormulaGadget :drawAgg :drawLine :points
                   (list 10
                    (+ 10 (abs h)
                         (gv $FormulaGadget :gestureBorder :top))
                    (+ 10 (abs w))
              (+ 10 (gv $FormulaGadget :gestureBorder :top)))))
                  (s-value $FormulaGadget :drawObj :drawLine))
                (s-value $FormulaGadget :drawObj nil))
              )

             (T
              (s-value $FormulaGadget :drawObj NIL))
             )
         )))
 (update-formula-versions)
     )
 ))
```

B.2.6 prev-formula

```
;;; -----------------------------------------------------------
;;; prev-formula
;;;
(defun prev-formula ()
  (update-formula-versions)
  (let* ((tempFormulaText (gv $FormulaGadget :formulaText))
       newFormulaString)
    (if (and
      (not (equalp (car $PreviousFormulas) (car $NextFormulas)))
       (> (length $PreviousFormulas) 1))
      (progn
        (setf $NextFormulas (cons
                        (car $PreviousFormulas)
                        $NextFormulas))
        (setf $PreviousFormulas (cdr $PreviousFormulas))
        (setf newFormulaString (car $PreviousFormulas))
        (if newFormulaString (progn
             (s-value tempFormulaText :string newFormulaString)
                   (s-value tempFormulaText :cursor-index
                          (length newFormulaString))
                   (display-graphical-formula))))))
)
```

B.2.7 next-formula

```
;;; -----------------------------------------------------------
;;; next-formula
;;;
(defun next-formula ()
  (update-formula-versions)
  (let* ((tempFormulaText (gv $FormulaGadget :formulaText))
       (newFormulaString (car $NextFormulas)))
    (if $NextFormulas
      (progn
        (setf $PreviousFormulas (cons
                          (car $NextFormulas)
                          $PreviousFormulas))
        (setf $NextFormulas (cdr $NextFormulas))
        (s-value tempFormulaText :string newFormulaString)
        (s-value tempFormulaText :cursor-index
              (length newFormulaString))
        (display-graphical-formula))))
)
```