ALGORITHM ANIMATION
IN A
DECLARATIVE VISUAL PROGRAMMING LANGUAGE

Paul Carlson

Margaret M. Burnett

Department of Computer Science

Oregon State University

Technical Report #95-60-2

April 18, 1995

ABSTRACT

How might capabilities for algorithm animation be seamlessly integrated into a programming language that is both visual and declarative? Until now, visual programming language researchers have not attempted to anwer that question, making the fruits of algorithm animation available only to users of textual programming languages. Users of visual programming languages (VPLs) have been deprived of the unique semantic insights algorithm animation offers, insights that would foster the understanding and debugging of visual programs.

We have answered the question by seamlessly integrating algorithm animation capabilities into Forms/3, a general-purpose, declarative VPL. Our results show that such a VPL can support algorithm animation without leaving the declarative, visual model, without adding new concepts to the language or how to program in it, and without deviating from the uniform representation established for the language. In addition, our research shows that the characteristics of declarative VPLs result in some interesting algorithm animation features not found in other systems.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

Algorithm Animation in a Declarative Visual Programming Language

CHAPTER 1
INTRODUCTION

1.1 Goals and Definitions

The goal of this research is to discover if algorithm animation can be supported by a declarative visual programming language without compromising the fundamental characteristics of that language. We believe that the union of algorithm animation with visual programming will improve both algorithm animation programming and the visual programming language itself.

For purposes of this thesis, animation is defined to be the illusion of continuous change produced by rapidly displaying a sequence of graphical images having incremental changes between adjoining images. Algorithm animation is the use of animation to illustrate the semantics of a program. The images of algorithm animation often have no direct mapping to either a program's data structures or to its explicit operations, but instead portray the abstract operations of the program. Furthermore, to aid in understanding these abstract operations, algorithm animation transforms the discrete transition between program configurations resulting from an operation into an animation of that transition. These transitional images have no semantic meaning, since they correspond to some intermediate state between valid program configurations.

Visual programming languages (VPLs) are programming languages that use a visual syntax.[1] They allow the programmer to express relationships among or transformations to data simply by sketching them, pointing at them, or demonstrating them -- not by translating them into sequences of commands, pointers, and abstract symbols. The

1. Visual programming environments for traditional textual languages are specifically excluded from this category.

primary objectives of VPLs are to simplify the programming process, thereby reducing the cost and effort required to program, and to improve the understandability of programs, thereby improving their reliability and maintainability.

1.2 Why Integrate Algorithm Animation into a Visual Programming Language?

It is not difficult to see that integrating algorithm animation with visual programming is a win-win situation. Three benefits of this integration are returning algorithm animation programming to its natural visual domain, improving the understandability of visual programs, and simplifying the process of constructing an algorithm animation.

1.2.1 Rescuing a Visual Process from Textual Approaches

While animation is inherently a visual process, the traditional approaches to algorithm animation programming are textual, requiring the programmer to translate his or her visual imagery of animation into textual, one-dimensional code. One way of freeing the programmer from the cognitive burden of translation is to make algorithm animation programming itself a visual process.

Algorithm animation system designers have recognized that animation programming is best done visually. Stasko says that visual programming of animation “supports simplified animation design by freeing developers from writing code” and “animation developers can shift their focus from the details of design language syntax to the more important matter of creating interesting, visually informative algorithm animations” [26]. Many designers have implemented visual interfaces to their textual algorithm animation systems [12] [17] [26].

But such visual interfaces still produce textual code as output, and often require the use of a trapdoor to the underlying textual language for more complex animations. Our work takes the next step in the direction pointed out by these visual interfaces to textual animation systems. By integrating animation capabilities into a VPL, algorithm animation programming becomes entirely visual, and exploits the benefits of visual programming, such as reduction in the number of concepts required to program, concreteness in the programming process, explicit depiction of the relationships embedded in the program, and immediate visual feedback during the process of entering or changing a program.

1.2.2 Illustrating the Semantics of a Visual Program

Animation can make a contribution to software development regardless of the type of programming language used. Duisberg observes that “animation can be seen as an integral part of a software design and development environment, with particular utility for debugging, process monitoring, and documentation” [12]. Mukherjea and Stasko say that algorithm animation “can be critical for determining why a program is not performing in its desired manner” [19].

Since the execution of a visual program is already seen visually, it might seem that animating a visual program does not provide the same benefit as animating a textual program. But the semantics of a visual program can be just as difficult to comprehend as those of a textual program, particularly when visual changes are sudden and discrete. For example, when visual changes to a user interface are sudden and unexpected, the user is often unable to detect any clear causal connection between previous state and current state, and is often uncertain about which changes are directly related to his or her actions and which are incidental [9]. But a well-constructed algorithm animation focuses the user’s attention to the important abstract operations of an algorithm, and improves comprehension by transforming sudden visual change to smooth, continuous visual change.

1.2.3 Simplifying the Process of Constructing Algorithm Animation

Animating a textual algorithm can be a laborious chore, and the benefits of using animation are often not great enough to offset the costs of constructing the animation [11] [16] [19] [29]. The task may involve special tools, different environments, and multiple files. For example, even Tango [23], whose research contribution was simplifying the process of constructing algorithm animation, requires two code files, one control file, and a support graphics platform for the display.

Many VPLs have inherent characteristics that simplify the process of constructing an algorithm animation. In particular, since the software development environment of many VPLs is often indistinguishable from the VPL itself, animation display, animation programming, and algorithm programming all can be done in this one common environment with a single uniform representation. It is our conjecture that this use of one environment and a uniform representation lowers the cost of constructing algorithm animations, frees the user from the burden of learning new tools, and thus makes algorithm animation attractive even for throw-away tasks such as debugging.

1.3 Guide to this Thesis

This thesis describes how we integrated a visual and declarative extension of an algorithm animation model called the path/transition paradigm [24] into Forms/3, a declarative VPL. Our goal was not a special-purpose visual algorithm animation system for animating algorithms implemented in other languages. Rather, our goal was to be able to use a VPL to animate the algorithms that are themselves programmed in that VPL.

At first we simply wanted to see if a declarative VPL could support algorithm animation without compromising the fundamental characteristics and concepts of that VPL. What we discovered was that not only are those characteristics and concepts not compromised, but that they can be exploited to produce algorithm animation capabilities with unique and intriguing features.

After Chapter Two's discussion of related work, we introduce animation programming in Forms/3 in Chapter Three by examining the construction of a simple animation example, the rolling wagon wheel, an example that will be referred to throughout the thesis.

One of the contributions of our research is generalizing the way animation is driven, from the traditional time-based approach to an approach based on generalized events. Although Forms/3's support for generalized events is not original to this thesis, because of their importance to our approach, Chapter Four contains an in-depth look at events and event-handling in Forms/3, and explains their influence on animation in Forms/3.

Chapter Five contains two examples of algorithm animation, animation of a selection sort and animation of a sum.

Some insights and other discussion points revealed by our research are the topic of Chapter Six. We begin by examining the unique features that the live and declarative characteristics of Forms/3 bring to animation in Forms/3. Next we describe how we extended the path/transition paradigm to a declarative, visual domain. Then we characterize algorithm animation in Forms/3 using a taxonomy developed by one of the pioneers of algorithm animation. We conclude by discussing how 'nested logical time', a future work task of Forms/3, will make algorithm animation a better fit into Forms/3.

Implementation issues are discussed in Chapter Seven. We describe Garnet, a set of user interface tools used in the present implementation of Forms/3, then discuss some implementation groundwork that was needed in order to implement animation and was thus done as part of the work for this thesis. The last section of the chapter discusses some issues related to the implementation of animation itself. Chapter Eight contains the conclusion, which is followed by the bibliography.

CHAPTER 2
RELATED WORK

Balsa (Brown ALgorithm Simulator and Animator) [1] was a pioneering textual algorithm animation system. Using Balsa, a programmer hand-crafts animations by writing graphical software and then inserting procedure calls, called interesting events, into the algorithm being animated. Many later approaches, including Stasko's path/transition paradigm, were modeled after the fundamental approach devised for Balsa.

Animus [13], also a textual system, simplifies the process of constructing algorithm animations developed for Balsa. The animation programmer avoids low-level graphics coding and instead specifies animation in a declarative manner through the use of temporal constraints. By extending a constraint language to include time, Animus offers a kit of components with graphical and temporal behavior built in; these components are then composed using constraints.

Animus' creator did some early research on the visual specification of animation [12]. Called 'animation by demonstration', the user draws objects, demonstrates their motion, and indicates the triggering points in the code through the use of gestures. To produce the desired animation, the system retools the textual source code based on the visual specifications.

ALADDIN (ALgorithm Animation Design and Description using INteraction) [16] takes this approach further. Using ALADDIN, all specifications of an animation are done visually. Graphical drawing facilities are provided for both the design of objects and the construction of animation steps that operate on those objects. The system then annotates the textual algorithm code by inserting graphical objects that represent animation operations. ALADDIN is used to generate instructional animations of Modula-2 programs, and an example of an annotated program is shown in Figure 2-1. The icons represent animation operations.

Figure 2-1. An animation-annotated algorithm using ALADDIN

Stasko recognized that the ad hoc approaches of earlier systems made algorithm animation needlessly difficult, and developed a model with precise semantics called the path/transition paradigm [24]. The focus of the path/transition paradigm is creating smooth, continuous image movement. This is accomplished by conceptually viewing all types of animation as an image moving along a path of incremental changes. For example, images are moved along paths, images are colored along paths, and the visibility of images is changed along paths. Since the path/transition paradigm was the starting point of animation in Forms/3, it is explained in greater detail in later chapters.

To implement his path/transition paradigm, Stasko developed a textual algorithm animation system called Tango [23]. Tango is based upon Balsa's concept of identifying interesting events in an algorithm. The programmer inserts animation operations into the algorithm being animated either through manual editing or by using a graphical editing tool. The algorithm animation design language consists of a package of C routines; the Tango system controller coordinates the animation window display with the algorithm and animation code. An example of an algorithm with Tango animation function calls inserted at key points of interest is shown in Figure 2-2. Animation function calls start with the keyword 'TANGO'. An animation function, which the programmer also codes, is shown in Figure 2-3.

```
#include <stdio.h>
#include "xtango.h"

void ANIMInit(), ANIMInput(), ANIMDraw(), ANIMExchange();

static NAME_FUNCT fnc[] = {
               {"Init",      1, {VOID, (FPTR)ANIMInit}},
               {"Input",     1, {VOID, (FPTR)ANIMInput}},
               {"Draw",      1, {VOID, (FPTR)ANIMDraw}},
               {"Exchange",  1, {VOID, (FPTR)ANIMExchange}},
               {NULL,        0, NULL, NULL} };

main()
{
 int n,i,j;
 int temp;
 int a[300];

 int count;

 TANGOalgoOp(fnc, "BEGIN");

 printf("Input number of elts in array\n");
 scanf("%d",&n);

 TANGOalgoOp(fnc, "Init");

 printf("Enter the elements\n");
 for (count=0; count<n; ++count)
    {  scanf("%d",&a[count]);
       TANGOalgoOp(fnc, "Input", a, count, a[count]);
    }

 TANGOalgoOp(fnc, "Draw", a, n);

 for (j=n-2; j>=0; --j)
    {  for (i=0; i<=j; ++i)
       {  if (a[i] > a[i+1])
             {  temp = a[i];
                a[i] = a[i+1];
                a[i+1] = temp;
                TANGOalgoOp(fnc, "Exchange", a, i, a, i+1);
             }
       }
    }
 TANGOalgoOp(fnc, "END");
}
```

Figure 2-2. An animation-annotated algorithm using Tango

```
void
ANIMExchange(p1,p2,p3,p4)
   intp1;
   intp2;
   intp3;
   intp4;
{
   TANGO_LOC loc1, loc2;
   TANGO_IMAGE rect1, rect2;
   TANGO_PATH onepath, path1, path2;
   TANGO_TRANS move1, move2, flip;

   rect1 = (TANGO_IMAGE) ASSOCretrieve("ID", p1, p2);
   rect2 = (TANGO_IMAGE) ASSOCretrieve("ID", p3, p4);
   loc1 = TANGOimage_loc(rect1, TANGO_PART_TYPE_C );
   loc2 = TANGOimage_loc(rect2, TANGO_PART_TYPE_C );
   onepath = TANGOpath_null(1);
   path1 = TANGOpath_example(loc1, loc2, onepath);
   path2 = TANGOpath_example(loc2, loc1, onepath);
   move1 = TANGOtrans_create(TANGO_TRANS_TYPE_MOVE, rect1, path1);
   move2 = TANGOtrans_create(TANGO_TRANS_TYPE_MOVE, rect2, path2);
   flip = TANGOtrans_compose(2, move1, move2);
   TANGOtrans_perform(flip);
   ASSOCstore("ID", p3, p4, rect1);
   ASSOCstore("ID", p1, p2, rect2);
   TANGOpath_free(3, onepath, path1, path2);
   TANGOtrans_free(3, move1, move2, flip);
}
```

Figure 2-3. Portion of Tango animation code

Stasko developed a direct-manipulation, by-demonstration visual interface to the Tango system called Dance (Demonstration ANimation CrEation) [26]. Using Dance, a programmer graphically creates and edits instances of the path/transition paradigm's abstract data types. Each data type has its own unique visual representation. The programmer can also visually invoke operations on the data types, via pull-down menus; the operations' parameters are specified by pointing with the mouse. After the user demonstrates an animation scenario, the system generates textual animation design code which is then used as input to Tango. The only similarity between Dance and animation in Forms/3 is using the trace of the mouse to specify a path.

Mukherjea and Stasko have begun the development of a visual debugging tool called Lens [19]. Lens' focus is allowing the programmer to rapidly develop algorithm animations of textual programs for debugging purposes. Lens' implementors say that algorithm animations built with Lens can be constructed in minutes. Use of Lens doesn't require any textual programming of animation code, as all specification and design of the animation is done in a visual environment. Lens is implemented in conjunction with a debugger, allowing it to support incremental development and refinement of algorithm animations.

Declarative visualization models, which map program state to geometric objects, have been extended to include an animation component [22] [29]. The programmer specifies animation through textual, rule-based notations; it is only the declarative characteristics that provide some similarity to animation in Forms/3. Pavane [22], a visualization environment for concurrent computations, implements one such extended declarative visualization model. An unusual aspect of animation in Pavane is that animation is triggered not by computational events, but rather by visual events. In other words, changes in the configuration of abstract geometrical objects trigger the animation.

CHAPTER 3
INTRODUCTION TO ANIMATION PROGRAMMING IN FORMS/3

In order to support algorithm animation, a VPL must first support animation programming. This chapter describes how animations are programmed in Forms/3, and Chapter 5 builds on this introduction by describing two examples of how animation programming is used to animate an algorithm.

3.1 Introduction to Forms/3

Forms/3 is a general-purpose, form-based language. Programming in Forms/3 follows the spreadsheet paradigm; the programmer uses direct manipulation to place cells on forms, and then defines a formula for each cell. Such a formula may include constants, references to other cells, or references to the cell's own value at a previous moment in time. Cells are referenced by clicking on them. A program's calculations are entirely determined by these formulas. A complete description of programming in Forms/3 can be found in [5].

3.2 Animation in Forms/3 From the User's Point of View

Writing a program to display a simulation of a spoked wagon wheel rolling down a ramp whose slope can be varied by the user was one of the problems from the Visual Languages Comparison at the 1994 IEEE Symposium on Visual Languages [30]. Figure 3-1 shows the user's view of the Forms/3 program that produces that animation.

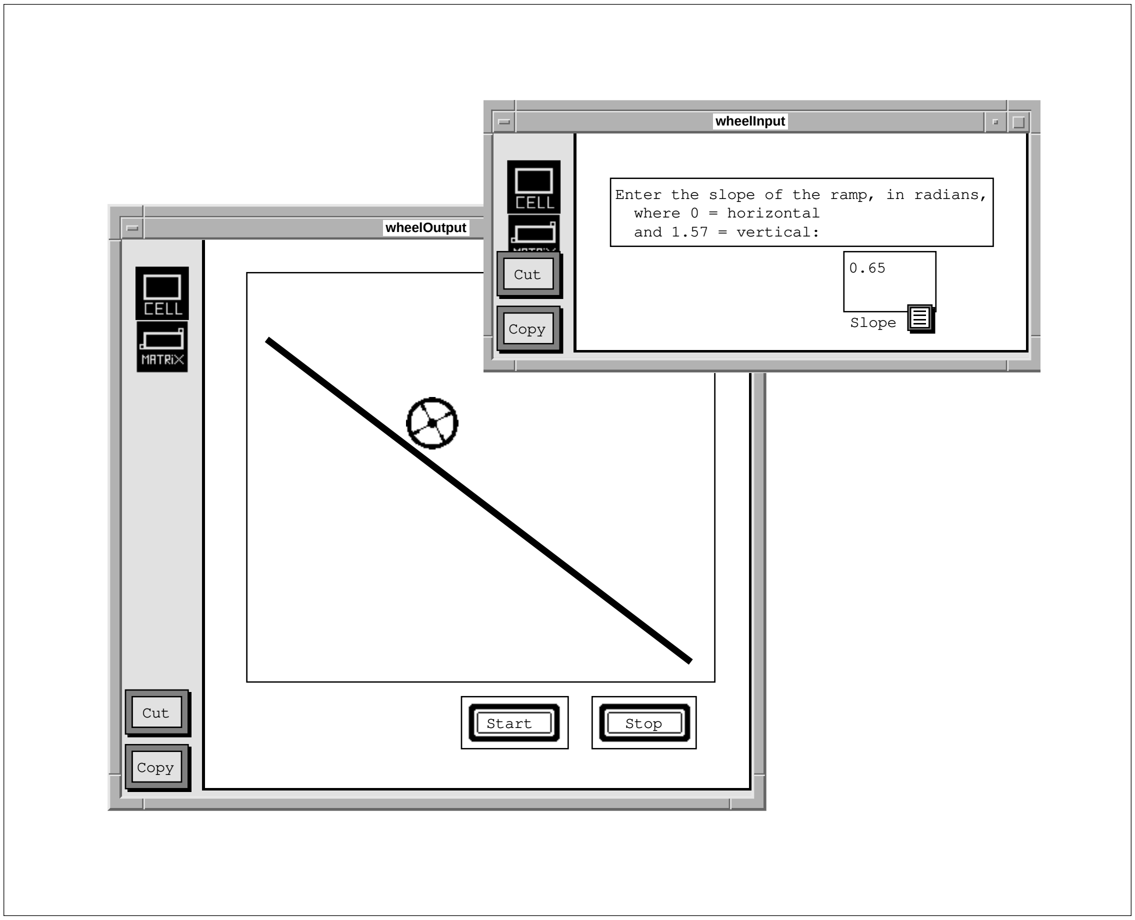

Figure 3-1. Rolling Wagon Wheel Program

The user changes the slope of the ramp by double-clicking on the Slope cell's formula tab, and then entering the desired slope as the cell's formula. The unlabeled cell above the Slope cell has a text string for its formula, and its formula tab is hidden. A programmer can hide a formula tab to prevent a cell's formula from being changed or viewed by the user. For the wheelInput form, the user is permitted to change the slope, but is not allowed to modify the textual comment.

The wheelOutput form of Figure 3-1 shows the wagon wheel rolling down the ramp. The formula of the cell that contains the display composes the wagon wheel with the ramp. The ramp's formula includes a reference to the Slope cell; this dependency ensures that whenever the slope is changed by the user, the ramp is recomputed based on the new slope. The user clicks on the Start and Stop buttons to control the animation. Forms/3's support for interactive events, which enables the programming of such buttons, is described in Chapter Four. All three cells on the wheelOutput form have their formula tabs hidden, because changing the formulas of these cells is not part of the way the programmer wants the user to interact with the wagon wheel example.

In addition to hiding formula tabs, the programmer can also make cell borders invisible. Figure 3-2 shows the wagon wheel example after the programmer has made four of the cell borders invisible. Since the visibility of cell borders has no semantic meaning, a decision on visibility is made solely from a standpoint of aesthetic tastes.

3.3 Animation Programming in Forms/3 in a Nutshell

Following Stasko's path/transition paradigm, animation programming in Forms/3 is founded on the concept of animating an object along a path. A path consists of a finite sequence of x-y coordinate pairs; each pair is interpreted as an offset from a previous value. Each step of the animation along the path is called an 'animation frame'. The object's traversal from the start of the path to the end of the path is called a 'transition'. An object and a path are two of the five parameters to a transition.

The third parameter to a transition is the transition type, such as Movement, Intensity, Visibility, and Color. Although it is most natural to think of the animation path as a sequence of graphical x-y coordinate pairs for the object to physically move along, all transition types use a path to animate an object. For example, an intensity transition animates an object along a path of intensity changes. The path still consists of a sequence of x-y coordinate pairs, but since the intensity transition is one-dimensional, only the y-coordinate is significant (either the x-coordinate or the y-coordinate would work equally well; we chose the y-coordinate because it fit better with our visual image of an intensity path).

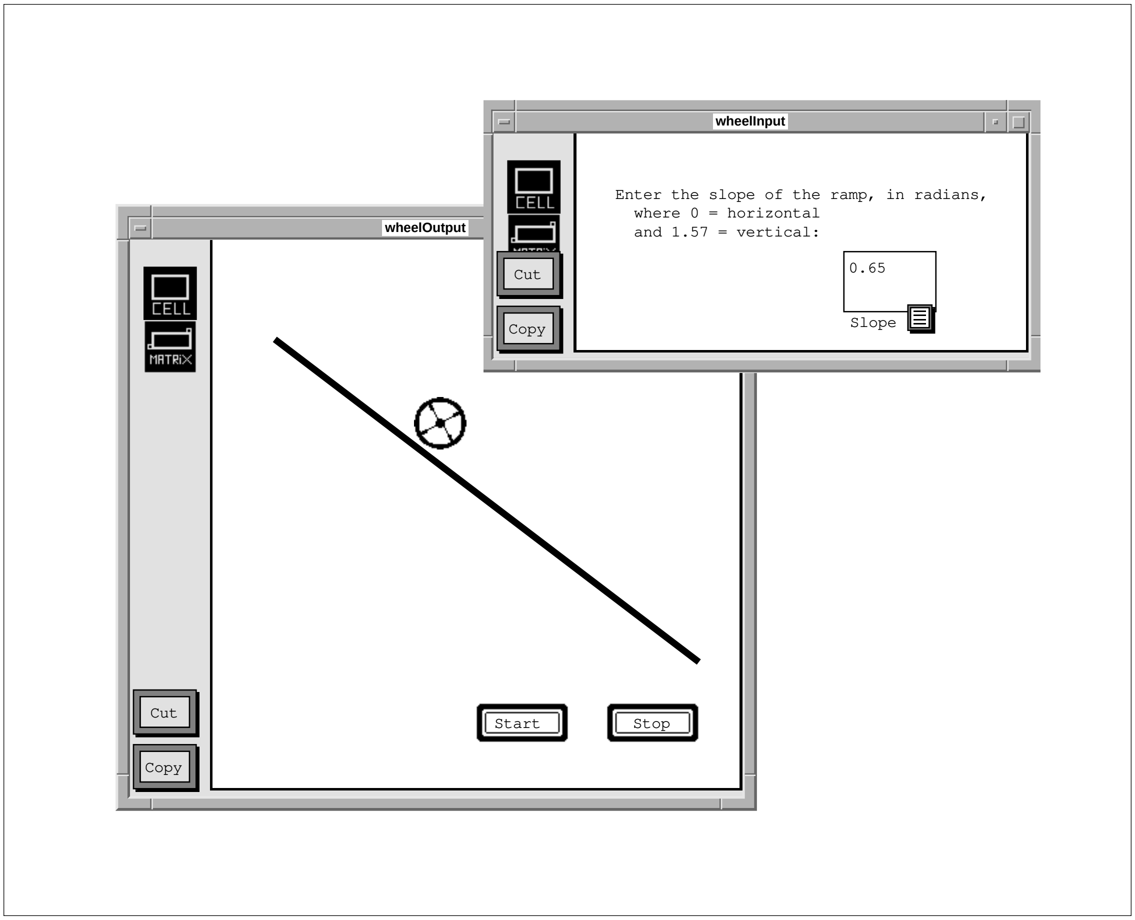

Figure 3-2. Rolling Wagon Wheel Program with cell borders hidden

The sequence of x-y coordinate pairs is a relative path. The entire path is relative to the object's starting value, and each coordinate pair specifies an offset to the object's value from the previous animation frame. Thus, two objects with different starting intensities, for example, that follow the same path will retain the same intensity difference as at the beginning of the path.

The last two parameters to a transition are resetEvent and continueEvent. The resetEvent parameter is used to cause the object to return to the beginning of the path, and the continueEvent parameter is used to cause the object to take the next step along the path.

3.4 Specifying Animation Using the Animation Form

Animation is visually programmed in Forms/3 by specifying the animation parameters on a form called the Animation Form. Figure 3-3 shows the Animation Form for the rolling wagon wheel program. The Object matrix and the Type, resetEvent, continueEvent, and Path cells are the inputs to the animation, and the Animation cell at the bottom of the form uses this input information to compute and display the appropriate output. In other words, the transition of the object along the path is rendered in the Animation cell.

The Object matrix contains the object(s) to be animated. An object can be any type, including primitive objects such as boxes, lines, bitmaps, and text strings; user-defined objects such as people or stacks; and arbitrary complex objects resulting from other calculations. When the Object matrix contains more than one object, as in Figure 3-3, the animation repeatedly cycles through the matrix, displaying one object per animation frame. This allows animation types (such as rotation) that are not one of the primitive transition types (Movement, Intensity, Visibility, and Color). For example, the Object matrix in Figure 3-3 contains a matrix of glyphs, or bitmaps. The differences between the three glyphs cause the wagon wheel to rotate as it moves down the ramp.

The Type cell contains the transition type of the animation. The programmer uses the radio buttons to select from the four available transition types.

When the resetEvent cell is true, the animation is restarted from the beginning of the path. When the continueEvent cell is true, the next animation frame in the transition is rendered in the Animation cell. The resetEvent cell has precedence over the continueEvent cell. The formulas of the resetEvent cell and the continueEvent cell for the wagon wheel example will be shown and explained in the next chapter, as part of a discussion about events in Forms/3.

The value of the Path cell is the relative path that the animation follows. The programmer indicates, by selecting the appropriate radio button, whether the path is to be computed or drawn. In Figure 3-3, the programmer has selected Computed Path.

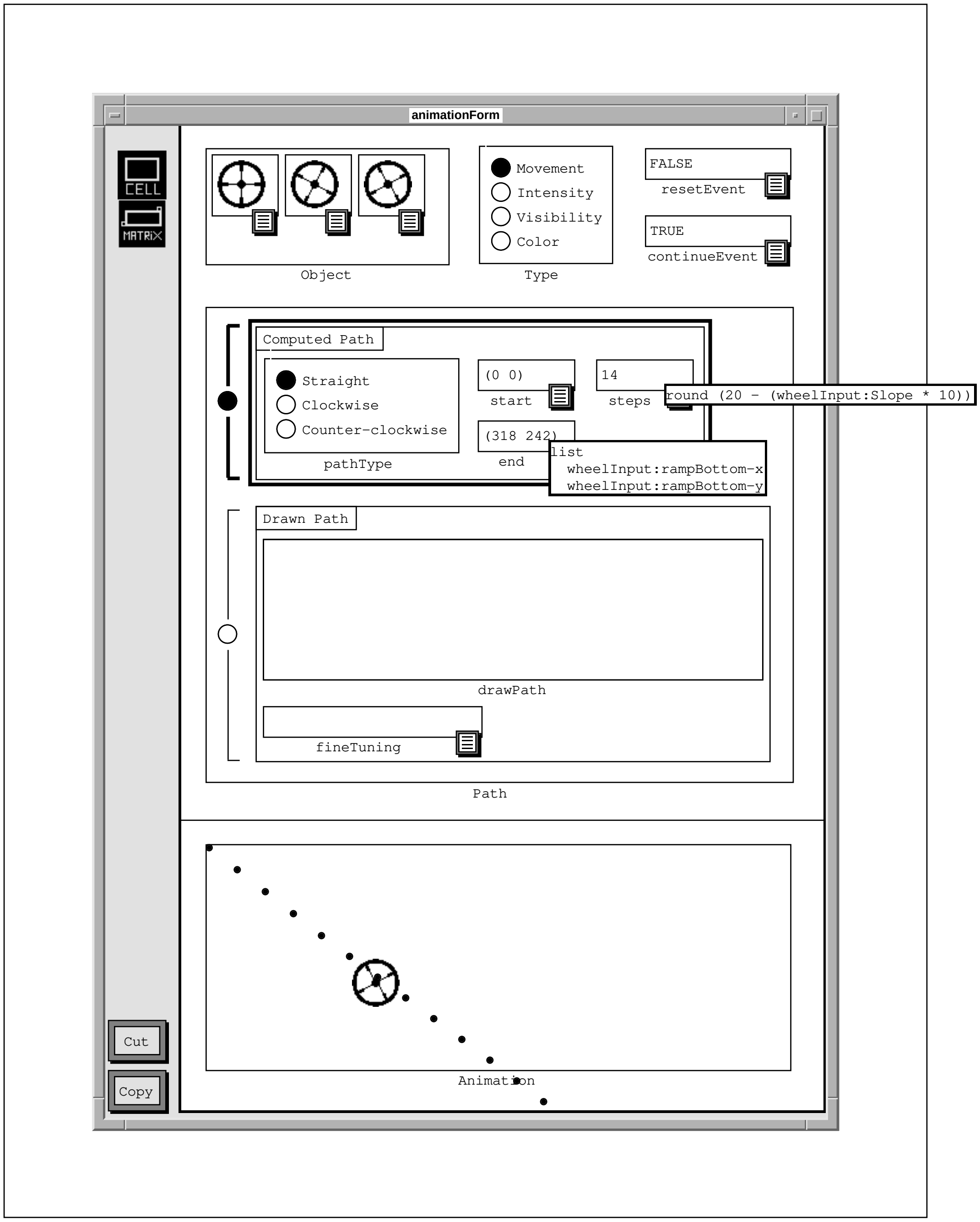

Figure 3-3. The Animation Form for the Rolling Wagon Wheel Program

A path is computed from four inputs: the path type, starting point, ending point, and number of steps between the points. For the wagon wheel example, the starting and ending points of the path should match the start and end of the ramp. Both the path and the ramp are anchored at (0 0). The formula for the path's ending point is shown in Figure 3-3 (displaying a cell's formula over its formula tab is a feature of the Forms/3 user interface, and is done by single-clicking the cell's formula tab). The formula shows that the ending point of the path depends on the ramp's ending point, so if the user changes the ramp's slope, the wheel's path will change accordingly. (Although the formula has a textual appearance, this is only an artifact of the present implementation. The formula was visually entered through direct manipulation, by pointing and clicking at the referenced cells.)

The formula for the Steps cell is also shown in Figure 3-3. Since the wheel should move faster down a steeper slope, and since the animation appears to move faster when the animation path has fewer steps, the number of steps between the starting and ending points decreases when the slope increases. The formula does indeed effect an inverse relationship between the number of steps and the slope. In general, a larger value in the Steps cell results in a smoother animation.

The path type in this example is “straight”. The other path types, “clockwise” and “counter-clockwise”, produce either a V-shaped or inverted-V-shaped path.

If Drawn Path is selected, the programmer draws the path by moving the mouse, while any mouse button is down, inside the drawPath cell. When the mouse button is released, the list of offsets of the path just traced is placed in the fineTuning cell, where the programmer can make precise adjustments to the drawn path. An example of a drawn path will be seen in Figure 3-4 and Figure 3-5.

The Animation cell renders the resulting animation. (The path of the animation is also rendered, by a different cell under the Animation cell.) If another cell's formula references this cell, it too will render the animation. The Animation cell in Figure 3-3 shows the wagon wheel after it has rolled a few steps along the path. The display of ramp and animated wheel in Figure 3-1 is obtained by composing the ramp with a reference to this Animation cell.

3.5 Combinations of Animations

Different types of animations can be combined to operate on the same object; this is done by building one animation using another animation as an input. For instance, a circle might change intensity while it is moving. The Animation Form in Figure 3-4 shows the intensity animation, and the Animation Form in Figure 3-5 shows the movement animation. As seen in Figure 3-5, the movement form's Object matrix contains a cell which references the result of the animation form set up for the intensity animation. The object of the movement animation is thus a circle with dynamically-changing intensity, and the result of the animation form set up for the movement animation will be a circle that changes in both intensity and location.

Forms/3's 'compose' operator can be used to produce a scene in which more than one object is being animated. Figure 3-6 shows such a scene, and Figure 3-7 shows the same scene a short time later. As the displayed formulas show, the intensity animation from Figure 3-4 is in the same scene with the movement animation from Figure 3-5. (The formula dependency between the two Animation Forms has been broken; each now produces an independent animation.) The intensity animation is changing the intensity of its object from light to dark and back to light again, according to the path drawn in Figure 3-4. The intensity animation is done in place, since only movement animation physically moves an object. The movement animation is moving its object down and to the right, according to the path drawn in Figure 3-5.

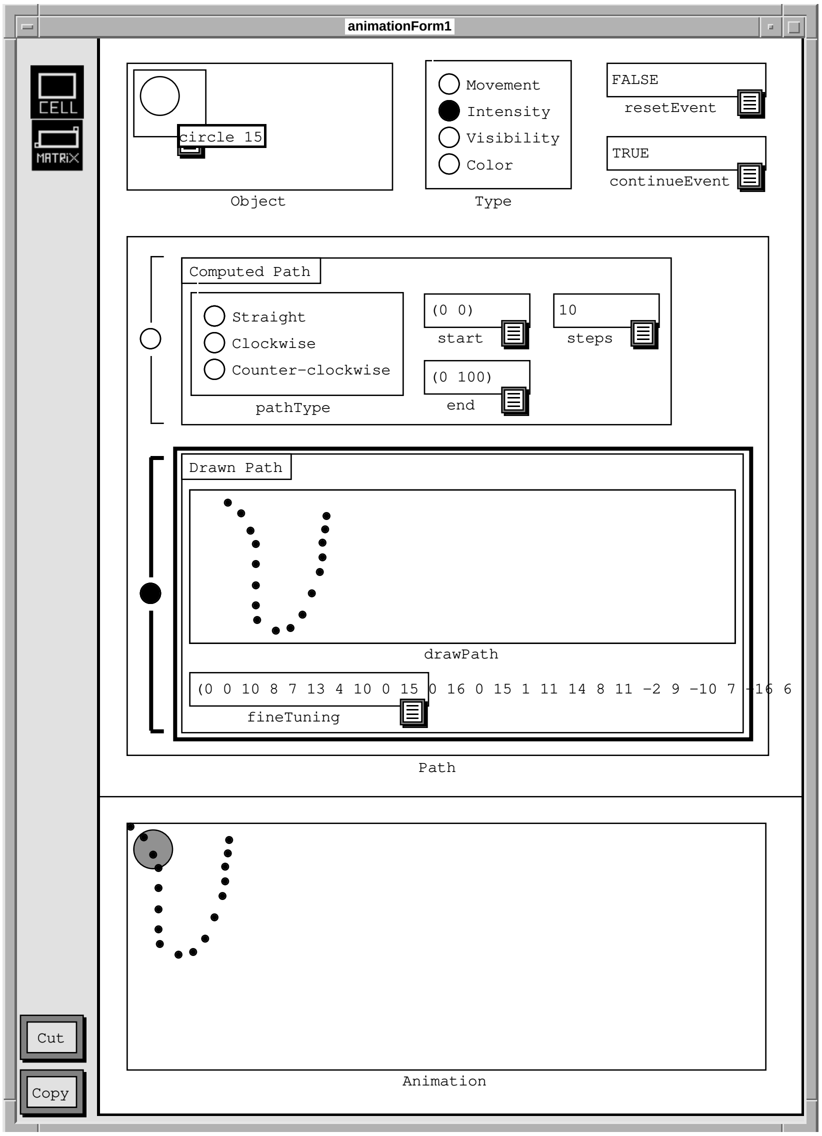

Figure 3-4. Animation Form showing intensity animation

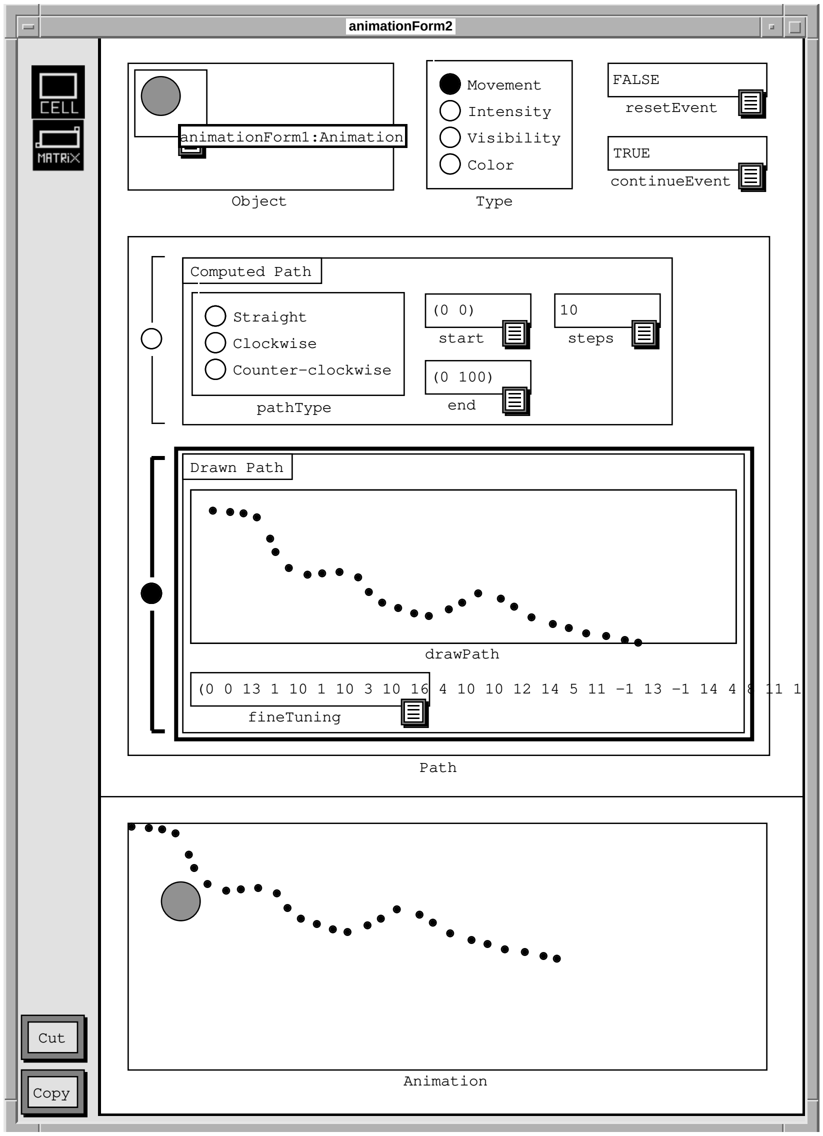

Figure 3-5. Animation Form showing movement animation

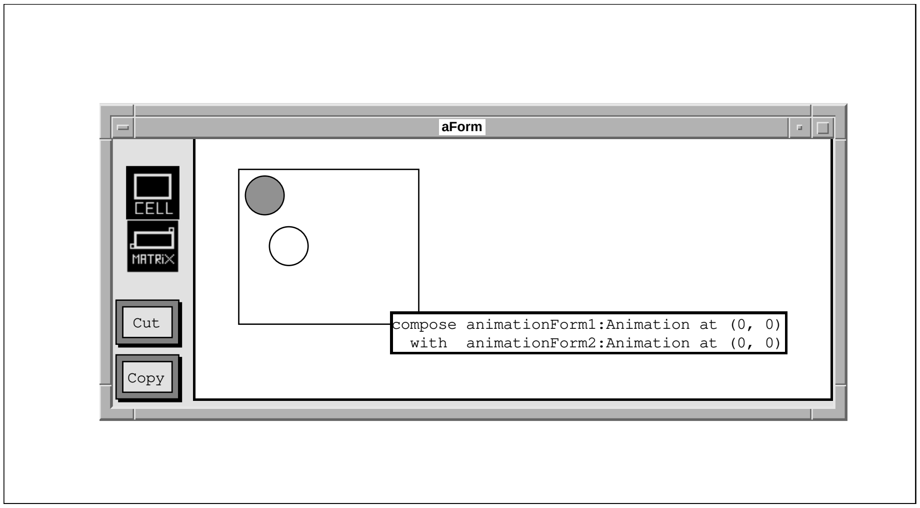

Figure 3-6. Composing two separate animations

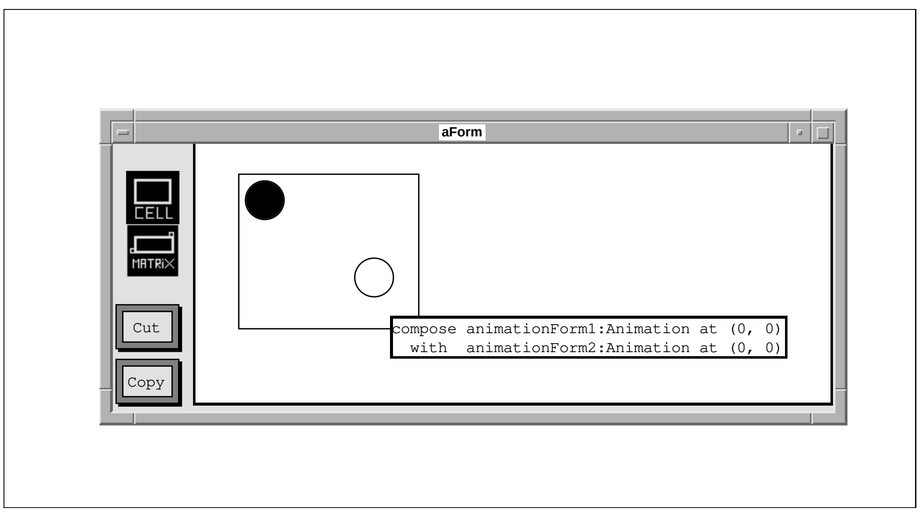

Figure 3-7. The composition a short time later

CHAPTER 4
GENERALIZED EVENTS IN ANIMATION IN FORMS/3

4.1 Motivation

Animation is produced by rendering an ordered collection of images one at a time, where each image has incremental changes from the previous image. The traditional approach to advancing animation is to delay an arbitrary time interval between each of these images, or animation frames. For example, if a ball moving between two points travels along a path of ten steps, there would be ten animation frames, each showing the ball at a step along the path. The animator might decide to delay a half-second between each frame, in which case the entire transition will take five seconds.

Despite this time-based approach to advancing animation, traditional algorithm animation systems are thought of as being ‘event-driven’ [23]. They bear such a label because events are used to *activate* animation. When constructing an algorithm animation, the programmer identifies key points called ‘events of interest’ in the algorithm being animated, and inserts functions calls to animation code at those key points. When the events of interest points are reached, the function calls execute and animation is activated.

Like all algorithm animation systems, our approach to activating animation is also event-driven. But we go one step further and generalize the traditional time-based approach to advancing animation by using events to *advance*, as well as to activate, animation. The key to understanding this generalization is the realization that the traditional time delay between animation frames is really just the occurrence of a specialized event -- the passage of a time interval -- which can be replaced by the occurrence of any general event.

This new generalized approach to advancing animation requires that the underlying language provide full support for generalized events and event-handling, and Forms/3 meets this requirement. The research and original implementation of events and event-handling in Forms/3 is described in [7]; part of the work of this thesis was re-implementing events in the present version of Forms/3.

4.2 Generalized Events and Event-Handling in Forms/3

4.2.1 Introduction

Forms/3 provides full support for all types of events, ranging from system-level events to user-interactive events to user-defined, high-level events. Since events are associated with change of state, support for events is often thought to be outside the domain of declarative languages. But Forms/3, a declarative VPL, uses a technique called ‘temporal assignment’ to treat events and data in a uniform way. Events and data are not only interchangable in Forms/3, but can also be composed together to produce high-level events, which are also simply another form of data.

Forms/3’s support for generalized events allows interactivity in the *programs produced* by programming in Forms/3, a step up from just supporting interactivity in the *process of programming*, which is a characteristic of many VPLs including Forms/3. Forms/3 was the first declarative VPL to provide support for general, programmer-defined interactivity in programs produced using the language [7]. An example of interactivity in a Forms/3 program was seen in the wagon wheel example of Chapter 3, in which the user of the program can interactively start and stop the animation via the Start and Stop buttons.

4.2.2 Temporal Assignment

The principle of temporal assignment allows Forms/3 to support generalized events. The idea of temporal assignment is that the formula for a cell does not define just a single value, but rather defines a vector of values along a time dimension. This vector of values is called a ‘temporal vector’, and the time dimension, called ‘logical time’, ranges from one to positive infinity. At any particular point in logical time, a cell does have a single value, which is the appropriate entry from the temporal vector for that logical time. For example, consider Figure 4-1. The formula for the Integers cell uses the temporal operators ‘fby’ (pronounced “followed-by”) and ‘earlier’, and means “1 followed by my earlier value plus 1." The temporal vector for the Integers cell consists of the positive integers starting with 1, and the Integers cell's value at logical time one is 1, at logical time two is 2, etc. Each value in a temporal vector is defined at a certain logical time, and expires at a certain logical time.

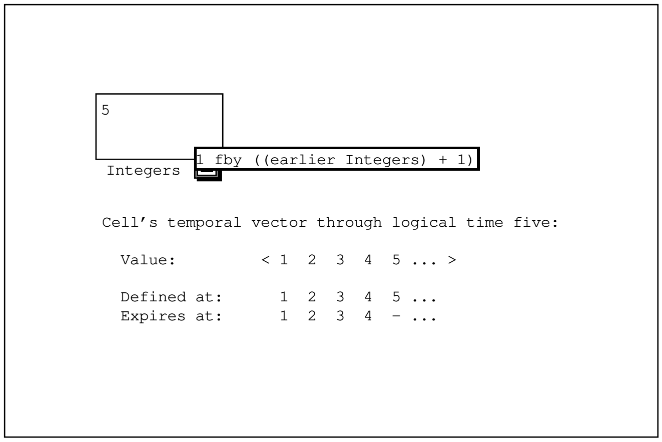

Figure 4-1. The Integers cell at logical time five

4.2.3 Filtering

Now consider the lastOddInteger cell from Figure 4-2, whose formula references the Integers cell. Since the conditional clause in the formula will evaluate to true only when the value of the Integers cell is odd, the temporal vector for the lastOddInteger cell consists only of the odd integers. The cell's value at logical time one is 1, at logical time three is 3, etc. What are the cell's values when logical time is even? *When a cell's formula does not produce a new value at a certain logical time, the logical range of the cell's previous value is extended to include that logical time.* So the lastOddInteger cell's value at logical times one and two is 1, at logical times three and four is 3, etc. Notice that the lastOddInteger cell's temporal vector has half as many entries as the Integers cell's temporal vector. A reasonable alternative approach, not used by Forms/3, is shown in Figure 4-3.

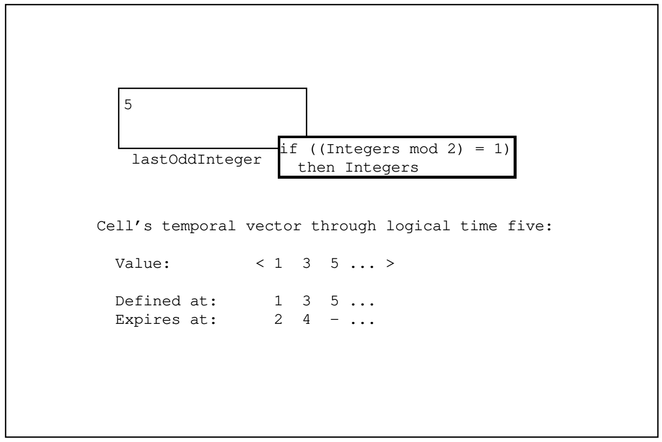

Figure 4-2. The lastOddInteger cell at logical time five

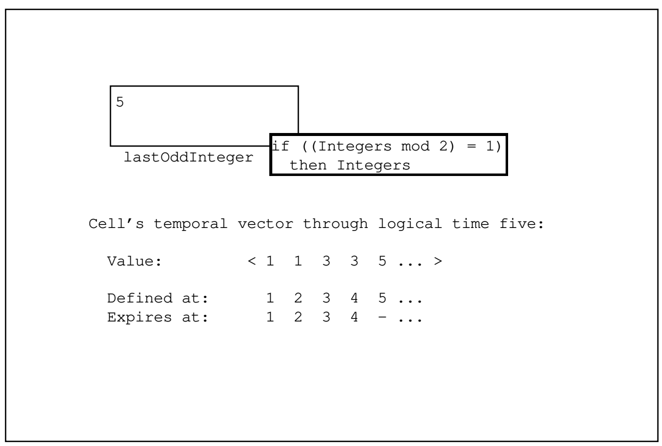

Figure 4-3. An alternative that Forms/3 does not use

Figure 4-2 is an example of *filtering*; we say that the formula for the lastOddInteger cell filters the Integers cell's temporal vector. Forms/3 uses the filtering approach over the approach in Figure 4-3 for more than just efficiency reasons. More importantly, the use of filtering to limit the presence of new values in a temporal vector is the key to treating data as events. Before examining the important concept of data-as-events, we first examine how temporal assignment allows the converse -- events-as-data -- and how event-handling is done in Forms/3.

4.2.4 Treating Events As Data

Suppose that the value of a cell in a Forms/3 program is a button for the user to press with the mouse (the wagon wheel example included two such cells). A description of such a cell's behavior would be "Display the image of a button and capture the occurrences of the user pressing that button." Now suppose that the user presses the button at logical time three, and again at logical time nine. The time line and temporal vector for that sequence are shown in Figure 4-4.

```
Time line

  Event:              Press Release            Press  Release
                          |    |                   |     |
  Logical time:  1    2    3    4    5    6    7    8    9    10    11    12  ...


Temporal Vector through logical time twelve:

  Value:           < NO-EVENT  PRESS  RELEASE  PRESS  RELEASE  ... >

  Defined at:              1       3        4        9       10      ...
  Expired at:              2       3        8        9        -      ...
```

Figure 4-4. Time line and temporal vector for button-presses

A comparison of the temporal vectors from Figure 4-4 and Figure 4-2 shows how Forms/3 uses temporal assignment to treat events just like ordinary data. Since an event at time t is simply the presence of a component in a temporal vector at time t, the only difference between the two temporal vectors is the type of their values. The temporal vector from Figure 4-2 contains values of type ‘integer’, and the temporal vector from Figure 4-4 contains values of type ‘event’.

4.2.5 The Event Receptor Form

To capture interactive events such as the user pressing a button, Forms/3 provides a primitive form called eventReceptor. An eventReceptor form from the wagon wheel example is shown in Figure 4-5. An event receptor has four parameters: shape, transparent, name, and events of interest. The Shape cell defines the area over which the event receptor will be sensitive to user interactivity. The Transparent cell is used to determine how overlapping event receptors detect events (the functionality of that cell is not currently implemented). The Name cell is used for identification purposes, both by the user and by the system. The eventsOfInterest matrix contains those events that the event receptor should detect.

The value of the eventReceptor cell of the eventReceptor form is an invisible image the shape of the Shape cell and sensitive to the events contained in the eventsOfInterest matrix. (The buttons from the wagon wheel example are a composition of the invisible event receptor with the Shape cell.) Associated with each event receptor is a special system cell called an event queue (not shown). The temporal vector of the event queue contains the events captured by the event receptor; the temporal vector in Figure 4-4 is actually from an event queue. Most of the remaining cells on the eventReceptor form provide more detailed information about the most recently captured event.

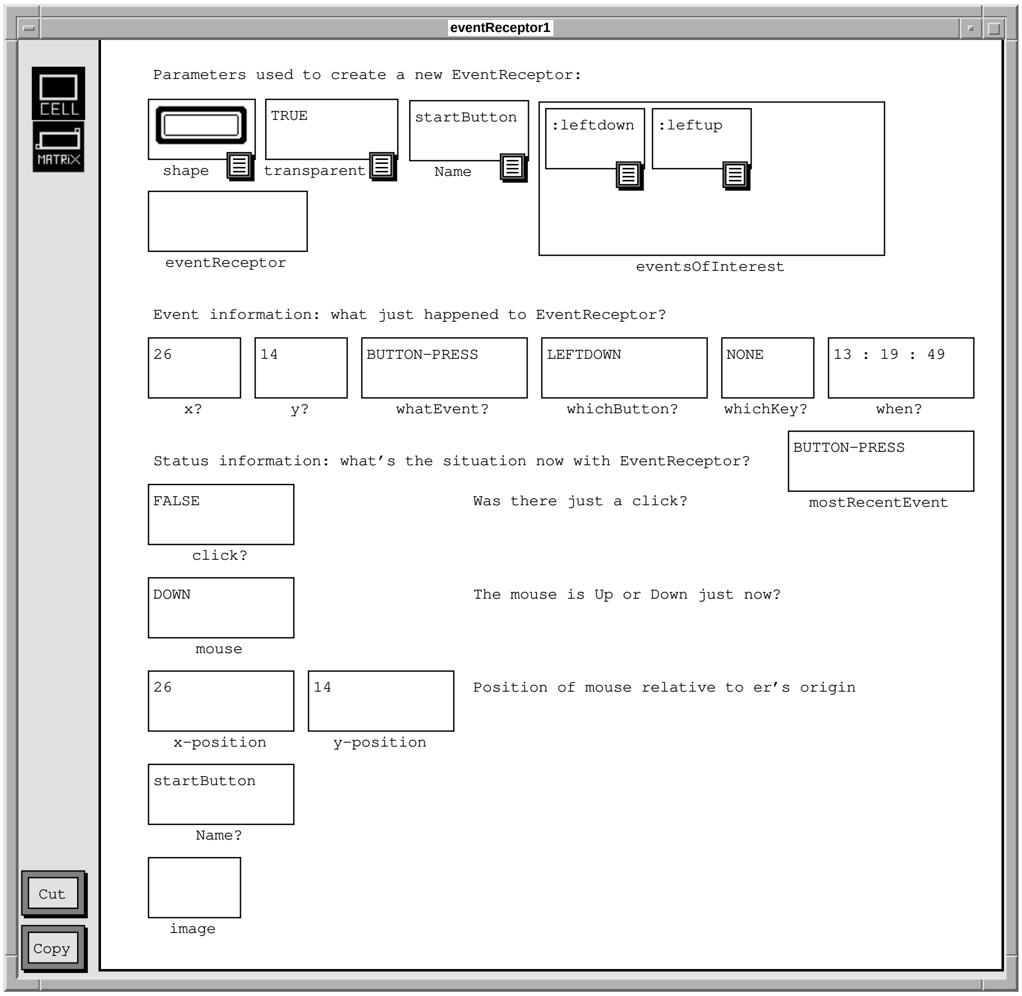

Figure 4-5. The eventReceptor form

4.2.6 Event-Handling in Forms/3

An event receptor's only task is to detect the occurrence of events; it does not respond to events in any way. Instead, event handling is done by defining the formulas of other cells to include a dependency on the event receptor.

For example, consider a button created from the event receptor in Figure 4-5. Suppose that the desired response to a button press is to increment the value of a certain cell by one. Figure 4-6 shows such a cell, its formula, and its temporal vector. The reference in the cell's formula to eventReceptor1:whatEvent? refers to the whatEvent? cell from Figure 4-5. The 'initially' construct is used to establish an initial condition for the 'earlier' construct.

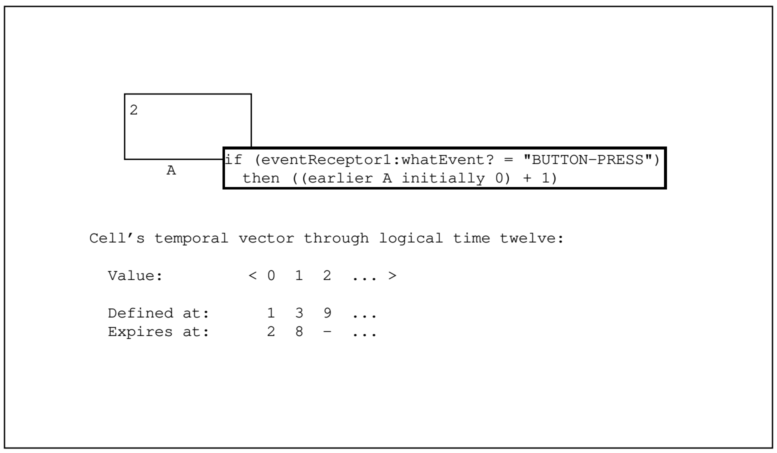

Figure 4-6. A cell that "handles" the event of button press

Suppose that the user follows the time line from Figure 4-4. In other words, the user has pressed the button at logical time three and logical time nine. Then cell A will receive a new value at those times, *and only at those times*. The functionality of cell A is to count the number of button presses. Since its behavior constitutes the response to the button being pressed, we say that cell A handles the button-press event. We are now ready to understand one of Forms/3's primary research contributions -- how data can be treated like events.

4.2.7 Data As Events

In the previous subsection, we saw an example of event-handling of a user-interactive event: a cell's value was incremented by one each time a button-press occurred. This was done by making the occurrence of the interactive event a condition for the cell to receive a new value. In the same manner, if a cell receives a new value only when some other cell's value satisfies a certain condition, then the first cell can be thought of as the event-handler of the *data event*[1] that occurs when the second cell's value causes the condition to be true.

For example, consider Figure 4-7. The data event here is the occurrence of an odd value in the Integers cell. Since oddCounter cell receives a new value only when that data event occurs, it is the event handler of the data event. Its functionality is to count the number of odd integers encountered thus far.

Earlier we saw how events are treated in the same uniform way as data in Forms/3, and a comparison of Figure 4-6 and Figure 4-7 shows the same uniform treatment to data-as-events. The cell in Figure 4-6 counts the occurrences of a user-interactive event, and the cell in Figure 4-7 counts the occurrences of a data event. The formula for each cell is identical in structure; only the conditional in the 'if-then' construct varies.

4.3 Applications of Generalized Events to Animation

4.3.1 High-Level Events and Their Use in Animation

The events-as-data and data-as-events relationship in Forms/3 provides the power to compose events with each other and with other kinds of data to create new high-level events. To illustrate high-level events, we return to the wagon wheel example and look at

1. Events of this type are called data events to distinguish them from user-interactive events and system events (e.g., clock ticks). However, the distinction is solely for clarity of explanation; in Forms/3 all three are first-class events.

the formulas of the continueEvent and resetEvent cells from that example (the Animation form containing those cells can be found in Figure 3-3). The formulas for these two cells are shown in Figure 4-8.

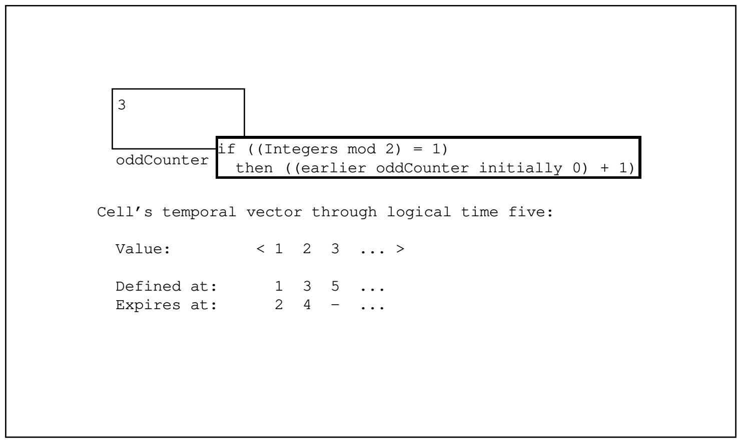

Figure 4-7. An example of handling a data event

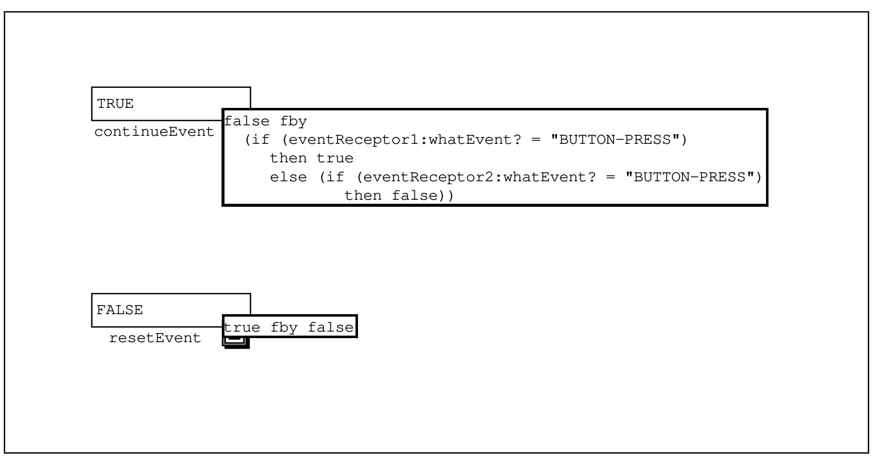

Figure 4-8. continueEvent and resetEvent cells from the wagon wheel example

We say a cell's formula defines a high-level event when two conditions are satisfied. First, the formula must transform one or more lower-level events or data into a new event of higher abstraction. In other words, on the continuum from primitive events (lower) to application-specific events (higher), the new event is higher on the continuum than the lower events it is built from. Second, one or more cells must handle the occurrence of the putative high-level event, through formula dependencies on the original cell.

For example, the formula of the continueEvent cell defines a high-level event. The first condition is satisfied because the formula transforms two lower-level interactive events into a high-level event that is a request to continue or discontinue the animation. The second condition is satisfied because the animation itself has a formula dependency on the continueEvent cell. In other words, it is the rendering of the animation that constitutes the desired response, or event-handling, of the high-level event defined and captured by the continueEvent cell.

The formula of the resetEvent cell also defines a high-level event, since it transforms low-level data (boolean values) into requests to restart the animation. Like continueEvent, the animation itself is the event-handler of resetEvent.

If we examine the formula for the continueEvent cell a little more closely, we see that it initializes the cell's value to false at logical time one, and from that point on the cell's temporal vector receives a new value only when one of the buttons is pressed. When the Start button is pressed, 'eventReceptor1:whichEvent? = BUTTON-PRESS' evaluates to true, continueEvent gets a new value of true, and the animation is started. When the Stop button is pressed, 'eventReceptor2:whichEvent? = BUTTON-PRESS' evaluates to true, continueEvent gets a new value of false, and the animation is stopped.

If none of the if-clauses evaluate to true, continueEvent will retain its previous value, because the logical range over which that previous value is defined will be extended to include the present logical time. This allows the animation to continue even after the user has released the Start button, or the animation to stay stopped even after the user has released the Stop button. This effect is due to the functionality of Forms/3's if-then construct: if the condition is not true at a given time and no else-clause is given, then no new value is defined at that time.

The formula for the resetEvent cell produces a temporal vector of two values: true at logical time one, and false at logical time two and beyond. This produces the desired effect of resetting the animation at logical time one but at no other time.

4.3.2 Contributions to Algorithm Animation

Using generalized events to advance animation allows the animation programmer to give the animation user control over the animation. We saw this in the wagon wheel example, where the user had control over starting and stopping the animation. If the passage of time were the only means of advancing animation, animation would always run to completion without any interactive control. Designers of existing algorithm animation systems recognize the importance that pausing and restarting an animation has on comprehending the algorithm being animated, and often include such interactive controls in their systems. The difference is that interactive control of animation in Forms/3 can be customized for each particular animation, rather than being limited to the hard-coded controls found in existing algorithm animation systems.

For example, the programmer may not want the user to have any interactive control for a particular animation, and thus use a time-based method to advance animation. But for a different animation program, the programmer may opt to include interactive control, and can choose from a range of choices, from some user-based interactive method to some method involving the use of programmer-defined high-level events.

4.3.3 Contributions to Graphics Animation

When the animation itself is one of the primary components of a program, we call that graphical animation programming. While algorithm animation is a means to an end -- an aid to increased comprehension of the algorithm being animated -- graphical animation is either an end in itself, or at least an important enough component that the program falls apart without the animation. The wagon wheel example is such an program, since the sole focus of the program is to animate the wagon wheel, and thus without the animation there is no program.

The use of generalized events in animation in Forms/3 can simplify the programming of graphical animation in Forms/3. As an example, imagine the wagon wheel program being modified slightly so the wheel is being pushed up the ramp by the image of a stick figure. Advancing the wheel up the ramp this way gives the impression of progress being made due to work. Such an animation might fit in nicely with an educational application that asks the user increasingly difficult questions. For each correct answer, the wheel gets pushed higher up the ramp, with perhaps some delightful surprise when the top is reached. A example screen shot of such a program (the program has not been implemented) is shown in Figure 4-9.

When creating this program, the Animation form acts as a template, simplifying and minimizing the amount of programming. The object, path, transition type, and resetEvent are similar to the wagon wheel example, except that the path is reversed and the object now includes the stick figure. The formula for continueEvent is simply a reference to the event of a correct answer.

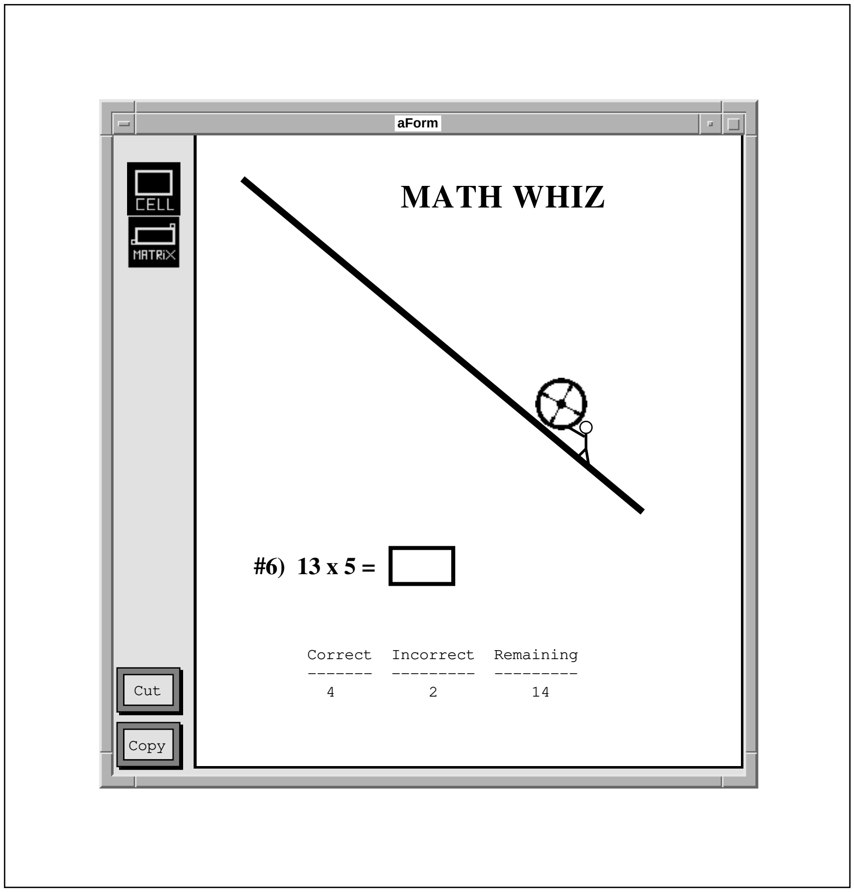

Figure 4-9. Screen for animated educational program (unimplemented)

CHAPTER 5
ALGORITHM ANIMATION EXAMPLES

5.1 Algorithm Animation of a Selection Sort

Figure 5-1 shows an animation of an in-place selection sort. The sort swaps the minimum of the remaining unsorted elements with the element in the first unsorted position. Such a sort is possible in the declarative Forms/3 due to the language's inclusion of the explicit time dimension discussed in the previous chapter. Each box of the animation corresponds to an element in the list being sorted, and the height of each box is directly proportional to its corresponding element's value. The animation shows the second and fifth elements being swapped.

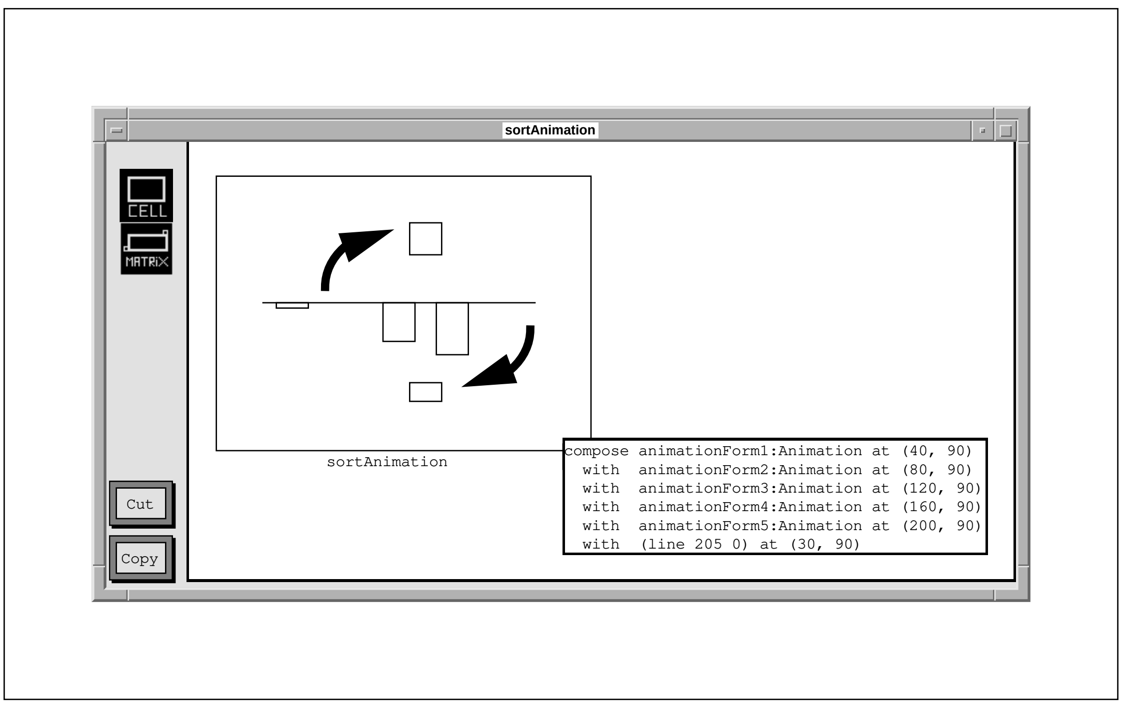

Figure 5-1. Animation of an in-place selection sort (the arrows are superimposed on the screen shot to show the direction of motion)

Figure 5-2 shows the user's view of the sort itself. (The sort form is implemented as a language primitive.) The list to be sorted is defined by the user in the initialList cell. The in-place sort occurs in the group of cells in the lower-left portion of the form. The algorithm does not know about its animation, and executes properly even if the sortAnimation form is not present.

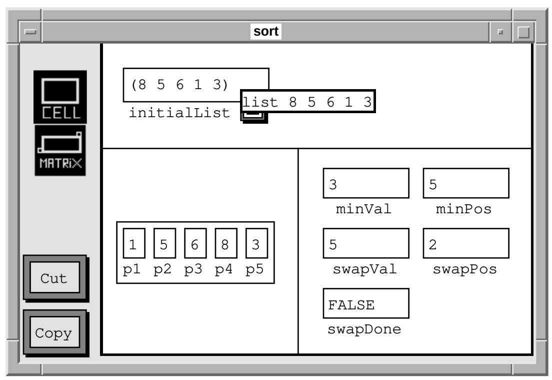

Figure 5-2. The in-place selection sort during execution

The programmer constructed this algorithm animation by programming one Animation form for each position in the list. The only other task was specifying the formula shown in Figure 5-1.

Figure 5-3 shows the Animation form for the second position, which presently holds the 5 element that is part of the current swap. The object is a box whose width is 25, and whose height depends on the value of the second element. continueEvent is true whenever the second element is part of the swap, and resetEvent is true after each swap is completed. When the present swap is completed, the list's second position will contain a

3, which causes the box to change size. This object change occurs at the same time that resetEvent is true, causing the new object to be rendered at the beginning of the path.

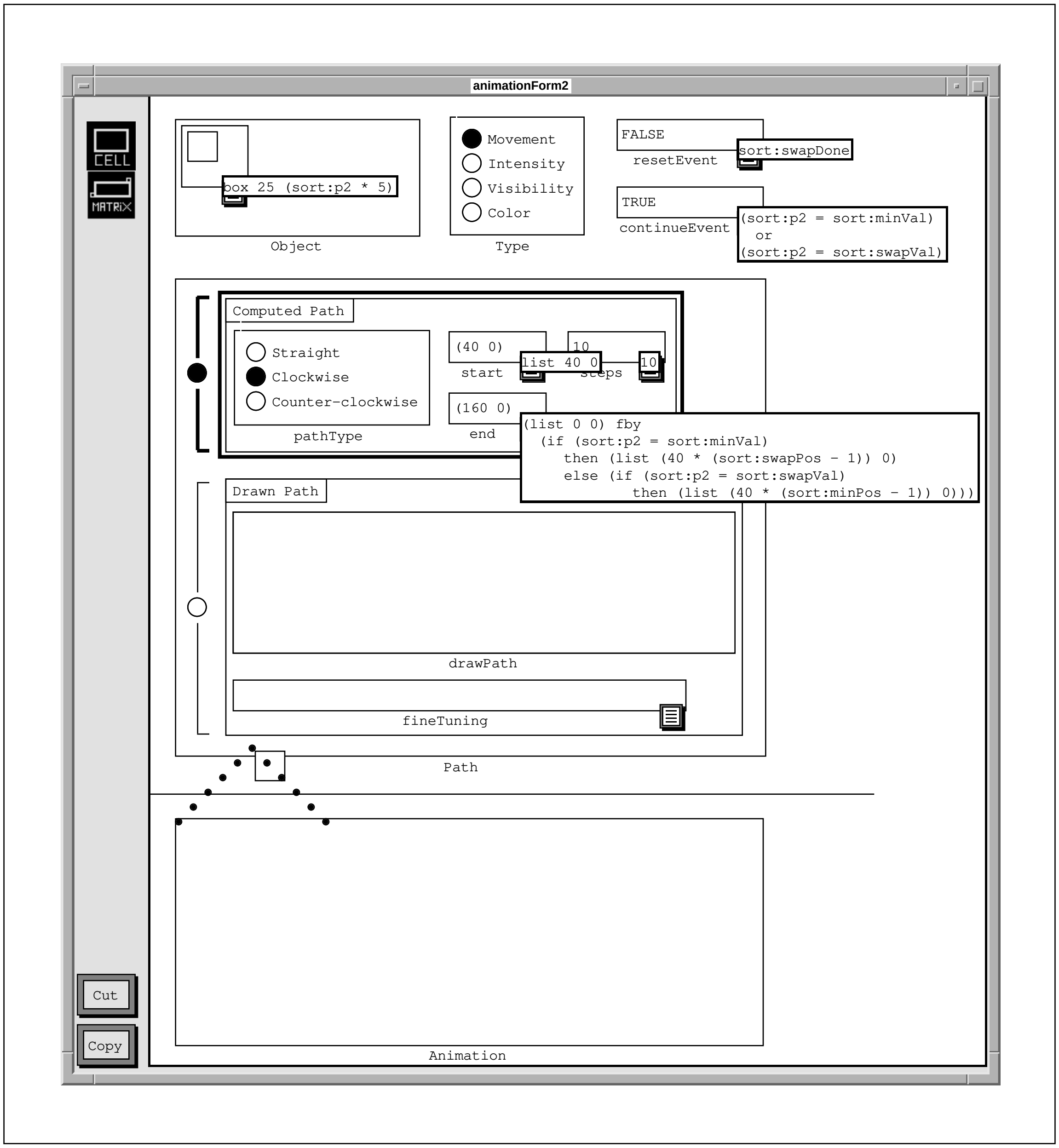

Figure 5-3. A representative Animation form from the in-place selection sort animation

As seen in Figure 5-3, the clockwise path is computed. Its starting point is the (constant) location of the second position and its ending point is the location of the destination position, in this case the fifth position. Note that if the second position is the minimum element, then the destination position is that of the 'swapped' element, and vice versa. The computed path is relative to those location points, which are in turn relative to the on-screen location of the sortAnimation cell (see Figure 5-1). The animation path and the rendered animation are displayed in the Animation cell.

5.2 Algorithm Animation of a Sum

Figure 5-4 shows the animation of a sum. The user enters the three numbers to be summed on the sumInput form. On the sumOutput form, the numbers use movement animation to travel to the total cell. Figure 5-4 shows the 5 moving towards the total cell, to join the 7 and 2 already summed there. During the animation, the number being animated appears inside a circle, instead of a box. We can imagine an animation of this type being used on a spreadsheet-like form that consists of many numbers and cells. The total cell might be the summation of three cells from scattered locations on the form; the animation assists the user in determining exactly which cells are included in the total. For example, a tax form might use income, deduction, and withholding totals to compute the tax due, and those totals might be located on different areas of the form.

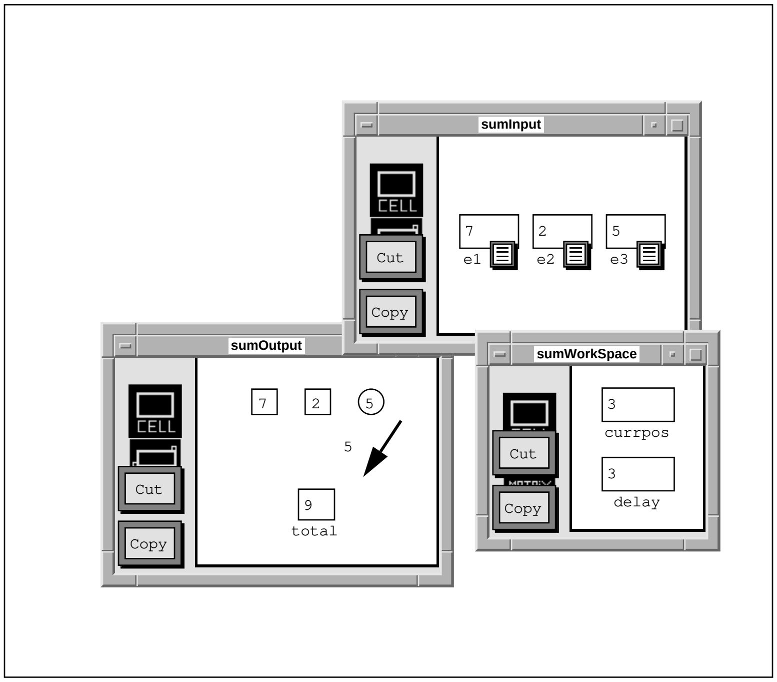

Figure 5-4. Input, output, and work space forms of the sum animation (the arrow is superimposed on the screen shot to show the direction of motion)

Figure 5-5 shows the Animation form used to program this example. Its construction is much like the Animation form for the sorting example. The delay and currpos cells from the sumWorkSpace form provide information to the animation. The currpos cell tells the animation which number it should be animating. The delay cell handles the timing information; that is, it instructs the animation when to reset. This necessity of explicitly tracking time is an artifact of the present Forms/3 implementation; the next chapter discusses a future improvement to solve this timing problem.

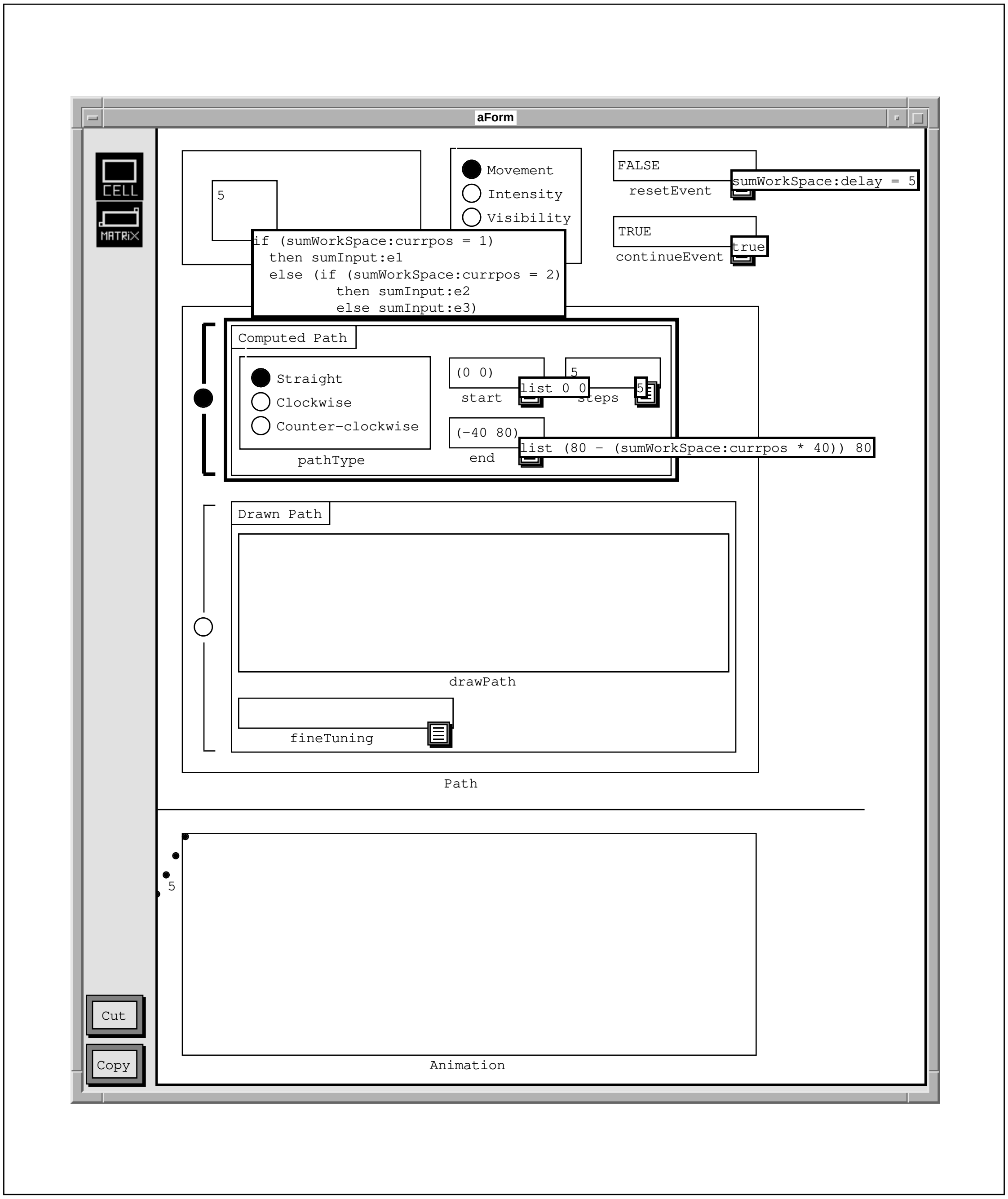

Figure 5-5. The Animation form of the sum animation

CHAPTER 6
INSIGHTS AND DISCUSSION POINTS

6.1 Unique Features of Algorithm Animation in a Declarative, Continuously-Responsive VPL

The fundamental characteristics of Forms/3, such as its declarativeness and its liveness, give algorithm animation in Forms/3 some unique features, which we explore in this section.

6.1.1 On-The-Fly Exploratory Programming

We call a VPL live, or continuously responsive, if, after changes are made to a program, the programmer is given immediate feedback about the effects of those changes. In other words, when using a continuously-responsive VPL, the process of programming is indistinguishable from the process of executing a program. Since Forms/3 meets the liveness requirements through a programming environment that is incremental and interpretive, algorithm animation programming in Forms/3 does not have the edit-compile-restart loop of imperative algorithm animation programming. Instead, a programmer can modify an algorithm animation during its execution (a refinement characteristic of growing importance in scientific programming known as 'steering'), and will receive immediate feedback about the effects of those modifications. For example suppose that something affecting the animation, such as an input parameter of the animation, a component of a combination animation, or even the animated algorithm itself, is modified at some point of the animation's execution. The liveness of Forms/3 means that the new animation resulting from this change will be automatically redefined, and the screen display will be updated to reflect the same point in logical time that the animation was at before the modification was made.

6.1.2 Changing the Direction of Execution

Not only can the programmer modify the animation (or algorithm) during runtime for exploratory or debugging purposes, but the new program -- in fact any Forms/3 program -- can be executed in either a forward or backward direction. The direction can be easily toggled at any time by simply clicking a button in the programming environment, permitting the programmer to examine any portion of the execution as many times as desired without restarting the entire program.

While some algorithm animation systems allow the *animation* to run in both directions, it is usually impossible for the *algorithm* to execute in reverse. In Forms/3, both animation and algorithm execute synchronously in either direction. This is possible because a cell's formula declaratively defines its complete history of values.

6.1.3 Separating the Algorithm from the Animation

In Forms/3, as in many declarative systems, programming can be thought of as defining a mapping from computational states to a desired outcome. The advantage of this declarative mapping approach is that the algorithm being animated is neither altered nor augmented with animation code. Instead, the interesting events of the algorithm are defined in the *animation* code. This allows the same algorithm program to be used both with and without its animation.

Any declarative approach to algorithm animation, whether visual or textual, can offer this feature; algorithm animation in Forms/3 is not the first. But the best-known algorithm animation systems are imperative, and require the programmer to construct an animation-annotated algorithm by inserting animation function calls at key points in the original algorithm. Under this approach, if the algorithm is not always to be animated, two versions of the same program file are needed -- one with the unaltered algorithm and one with the animation-annotated algorithm. Unfortunately, this leads to the undesirable situation where modifications to the algorithm must be made in two places -- surely the bane of software maintenance.

6.2 The Path/Transition Paradigm in a Declarative, Visual Setting

As mentioned earlier, Stasko's path/transition paradigm appealed to us as a starting point for devising and implementing animation in Forms/3 because it is sound conceptual model with precise semantics, which we prefer over more ad-hoc approaches. Other attractive characteristics of the path/transition paradigm are its support for smooth, continuous image movement and its support for animation types other than simply movement. Its use of a relative path to indicate changes between animation frames fits in nicely with the declarative model.

However, although the path/transition paradigm is a model and not an implementation, it is a model intended for implementation in a programming language that is imperative and textual. Thus, our task was to modify the model to make it suitable for implementation in a language that is both *declarative* and *visual*.

The path/transition paradigm consists of four abstract data types: the graphical images on the screen, the locations that images occupy, the transitions that the images make, and the paths that dictate the images' transitions. Animation in Forms/3 uses images and paths without modification, but transitions are treated slightly differently and locations are not used at all. Our approach also introduces two new parameters: resetEvent and continueEvent.

In Stasko's imperative textual implementation of the path/transition paradigm, a transition takes three parameters: a transition type, an image, and a path. The transition is then explicitly executed with a 'perform' command. In our approach, a transition also takes a transition type, an image, and a path, but further takes a resetEvent and a continueEvent. In effect, we replace Stasko's perform command with resetEvent and continueEvent. This is necessary because in the live, responsive visual domain of Forms/3, the transition (represented by the Animation cell on the Animation form) is continuously demanded,[1] meaning it does not need to be explicitly triggered per se. Rather, it is told when it is allowed to begin (resetEvent is true) and when it is allowed to advance (continueEvent is true).

The location abstract data type of the path/transition paradigm is used for positions of interest within the animation coordinate system. Typically the value of a variable is denoted by an image at a particular location. For instance, the animation of a sorting algorithm might represent an array element by a rectangle at location (x,y). This rectangle can be queried for its location, and the response to that query is often used as the starting point of a path for the rectangle to follow.

In animation in Forms/3, locations are also used to denote a particular variable in an algorithm. For instance, the selection sort animation of the previous chapter used the first rectangle to denote the value of the first array element, etc. But in order to maintain referential transparency in Forms/3, an image is not able to return its absolute screen location.

A language is referentially transparent if the value of a function depends only on the values of its parameters, and not on any other factors, such as the order of evaluation of the parameters or when the function is called. In Forms/3, a cell's location on a form has no semantic meaning; that is, moving the cell to a different location on a form does not change the program's outcome. If a Forms/3 image could return its absolute location, moving the cell that contains that image, while not changing the program, would produce different values of the location function, violating referential transparency.

We avoid the need for absolute locations through the use of relative paths. Not only are the steps within the path interpreted as incremental changes to the image's previous location (like in the path/transition paradigm), but the overall path itself is relative to the image's starting location (unlike the path/transition paradigm, which asks the image for its location and starts the path from that absolute point).

1. A cell in Forms/3 is continuously demanded as long as the cell is visible, or on the screen. This aspect of evaluation in Forms/3 is not discussed in this thesis. See [5] for more information.

6.3 Characterizing Algorithm Animation in Forms/3

Brown's taxonomy of algorithm animation display approaches [4] allows us to characterize algorithm animation in Forms/3. The taxonomy has three dimensions, which are Content, Persistence, and Transformation.

The Content dimension ranges from direct displays to synthetic displays. Direct displays are a direct representation of a program's data structure. The isomorphism between display and data structure means that the data structure could be constructed from the display. An example of a direct display is a circle whose radius represents the magnitude of a variable. Synthetic displays are not produced by a mapping to a data structure, but instead either show the abstract operations of a program or an abstraction of the program's data. Synthetic displays often convey information that is not directly available from an algorithm. Algorithm animations will often use both direct and synthetic displays in the same view. For instance, the selection sort animation from the previous chapter uses a synthetic display, since the swap operation is an abstract operation of the algorithm, and it also uses a direct display, since the magnitude of an element is directly mapped to the height of the box corresponding to that element. From this example, we see that both direct and synthetic displays are possible in animation in Forms/3.

The Persistence dimension ranges from 'current' to 'history' and indicates whether a display shows only current information about an algorithm or whether it can illustrate some history of the algorithm's execution. This historical information need not necessarily be part of the animation process, but rather can be part of the overall display. Displaying a list of all the element comparisons made in a sort algorithm is an example of a historical display. Since the algorithm animation might only animate the swap operation, this history contains different information than that displayed in the animation. Since our animations are implemented in a general-purpose programming language, animation in Forms/3 spans this entire range and there is no limit to the complexity and depth of historical information we can display.

The Transformation dimension measures how the objects in an animation transform, and ranges from discrete, sudden changes to smooth, incremental transitions. Referring again to the sorting example of the previous chapter, when two elements switch positions in the array, the swap occurs incrementally and the viewer can clearly see which elements are involved. A sudden, discrete change would swap the elements instantaneously, and the viewer would only know which two elements were swapped by comparing the screen to his or her memory of the previous screen. It is the animation path step size that determines the degree of smoothness; in animation in Forms/3, this step size can be set to produce any level of smoothness. Brown states that "unless a good animation package is available, incremental transitions are often tedious and difficult to program" [4]. Stasko's path/transition paradigm simplifies the programming of incremental transitions, and by using it as our foundation, animation in Forms/3 also achieves the same simplicity.

We conclude that animation in Forms/3 spans the complete range of all three dimensions of Brown's algorithm animation taxonomy.

Another dimension found in the literature [28] is the Automation dimension, which measures the level of assistance (through automation) provided when developing an algorithm animation. Animation in Forms/3 requires explicit programmer specification, but lack of automation is not considered a deficiency for algorithm animation systems. Since identifying the abstract operations of an algorithm almost always require programmer assistance and direction, abstraction and automation usually have an inverse relationship [28].

6.4 Nested Logical Time

One area of future work that will improve animation in Forms/3 is the concept and functionality of 'nested logical time'. Nested logical time allows a logical time unit to be broken into smaller units, much as a minute is broken into seconds. This functionality will allow a greater decoupling between algorithm and animation than that which exists using the current implementation.

For example, consider the animation of a sort. The swap operation takes just one unit of logical time. But if a ten-step path is used to animate the swap, like the sort example in

Chapter 5, executing the animation takes ten units of logical time. Synchronizing the algorithm and the animation thus requires that the algorithm delay some amount of logical time for the animation, and thus the algorithm is no longer unaware of its animation. Nested logical time will solve this problem, allowing the animation to run ‘faster’ than the algorithm.

CHAPTER 7
IMPLEMENTATION ISSUES

7.1 Introduction

This chapter provides some implementation background and discusses some implementation groundwork done as part of this thesis. First we offer a brief introduction to Garnet, a set of tools used to construct the present user interface to Forms/3. Then we discuss two implementation tasks done as part of this thesis: the What-Affects-What Table (WAWTable), an efficient method of tracking dependencies in Forms/3, and event receptors, the sole language construct to provide user interactivity in programs produced using Forms/3. Both the WAWTable and event receptors had been implemented in a previous version of Forms/3, but neither had been converted to the current version. The chapter concludes with details of the implementation of animation and a discussion of whether our approach to animation could be entirely implemented just using Forms/3 itself, an important measure when trying to avoid the addition of new concepts to the language.

7.2 Garnet's Role in the Implementation of Forms/3

The current version of Forms/3 is running under HP-UX 9.0 and SunOS 5.2 using Lucid Common Lisp, and its user interface is implemented using Garnet [14] [20]. Earlier versions of Forms/3 had no user interface, and the addition of a user interface has raised many questions and issues. One of the most important of these issues is determining what features of Garnet can be utilitized in the user interface without changing the underlying Forms/3 language.

Garnet is a set of tools that aid the design and implementation of highly interactive, graphical, direct manipulation user interfaces. The user interface to Forms/3 is implemented using Garnet's low-level tools, called the Garnet Toolkit. The Garnet Toolkit supplies an object-oriented graphics system and constraints, a set of techniques for specifying the objects' interactive behavior in response to the input devices, and a collection of interaction techniques.

For example, a Forms/3 cell consists of a number of Garnet objects. The cell's border is a rectangle and its value is displayed using a text string or other appropriate graphical object. The cell's formula tab is a Garnet interactor (interactors are encapsulations of input device behaviors). When the formula tab is double-clicked, it responds by displaying a Garnet window containing another interactor whose purpose is to accept the new formula from the programmer.

Interactors, which are parameterized to allow precise control of their action, are widely used throughout the current Forms/3 implementation to capture user input. For instance, on the Animation form it is a Garnet interactor that captures the mouse trace and turns it into a relative path. The widespread use of interactors came about only after it was determined that their use does not change the underlying language of Forms/3, and that the functionality that interactors provide can also be provided in other ways. For instance, a previous implementation of Forms/3 provided the same functionality with xlib calls via the CLX interface to XWindows. Since Garnet's interactors do not change the underlying language in any way, we choose to use them because they allow programming at a higher level, and fit in with the rest of Forms/3's use of Garnet. The upcoming section on animation implementation will show how the same functionality provided by the interactors used in animation can also be provided using Forms/3 itself.

7.3 The WAWTable

We have noted that programming in Forms/3 is done by placing cells on a form and defining a formula for each cell. A programmer edits a program by defining a new formula for one or more cells. Because of the dependencies between cells, defining a new formula (and thus a new value) for one cell can cause a cascading change of values of other cells. For example, consider Figure 7-1. Cell B has a formula dependency on Cell A, and Cell C has a formula dependency on Cell B. Suppose that Cell A's formula is changed from the constant value '5' to the constant value '8'. The immediate feedback characteristic of Forms/3 requires that the values of Cell B and Cell C be instantaneously updated when Cell A receives a new formula (and thus a new value). Although the dependencies among these cells can easily be obtained in one direction from their formulas (e.g., "What cells affect cell B?"), another data structure is needed to obtain them efficiently in the other direction (e.g., "What cells does cell B affect?").

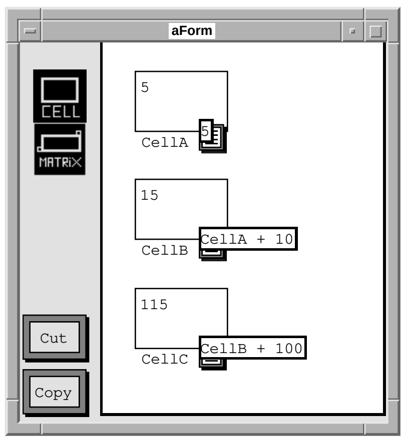

Figure 7-1. Cells having formula dependencies

Forms/3 uses an internal hash table called the What-Affects-What Table (or WAWTable) to maintain the formula dependencies necessary for efficient immediate feedback[1]. For example, when the formula for Cell B of Figure 7-1 was initially defined, an addition was made to Cell A's entry in the WAWTable, stating that Cell A affects Cell

1. A brute force sequential search was used prior to this thesis, but it was significantly slower.

B. Similarly, when the formula for Cell C was defined, an addition was made to Cell B's WAWTable entry noting that Cell B affects Cell C. Now, when Cell A receives a new formula, the WAWTable is used to determine the closure of dependencies on Cell A, and all cells in the closure are re-evaluated in light of the new formula. To maintain correctness, entries are also removed from the WAWTable when appropriate. For instance, if the formula for Cell B were changed to no longer depend on Cell A, Cell A's WAWTable entry would be adjusted accordingly.

In addition to formula dependencies, the WAWTable also tracks what are called 'copy dependencies'. Copy dependencies, which occur when forms are copied as part of Forms/3's code reuse scheme, are beyond the scope of this thesis and thus not discussed further.

7.4 Event Receptors

The event receptor, which was discussed in detail in Chapter 4, is the only programming construct in Forms/3 that can receive interactive events. Thus, a program produced using Forms/3 that includes interactive behavior must use event receptors. Event receptors are implemented using Garnet interactors; these interactors are customized for each event receptor based on the input parameters to the event receptor. The interactor for an event receptor captures <u>all</u> user events that occur over its sensitive area, but since each interactor is customized based on the events of interest for that event receptor, the interactor places only the significant events, as determined by the events of interest, into the event queue for the event receptor. The present implementation defines twelve event queues, an arbitrary number that can be extended if needed. The event queues physically reside on a special System form, shown in Figure 7-2, along with some other system cells. An event receptor is automatically mapped to an event queue based on the value of the event receptor's Name cell. Thus, multiple event receptors having the same name will contribute events to the same event queue.

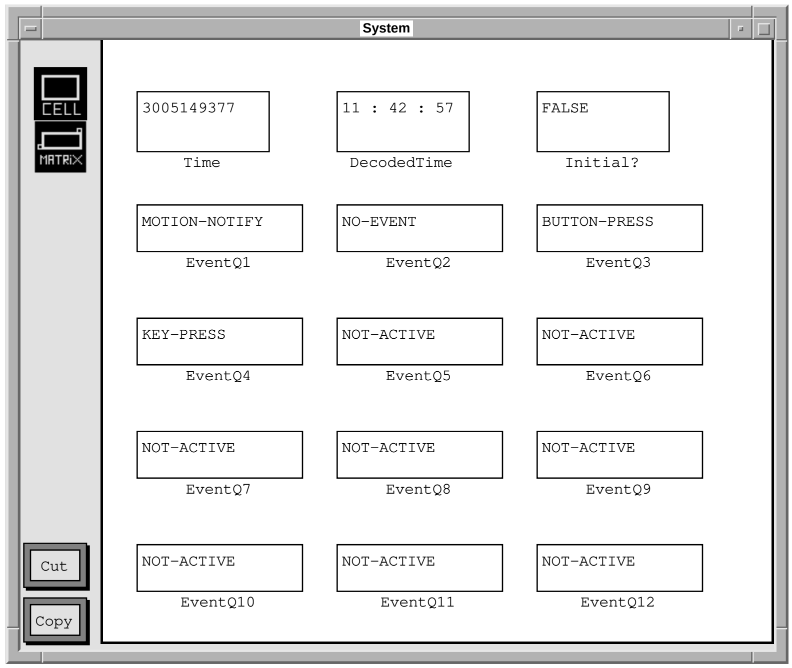

Figure 7-2. The System form

7.5 Animation

7.5.1 Implementation of the Animation Form

The three sets of radio buttons on the Animation form (see Figure 3-3) are implemented using event receptors. For example, consider the transition type cell. Behind the four circles posing as buttons rests an event receptor whose shape is the four circles and whose events of interest are left-down and left-up. It is the y-coordinate of the user's actual mouse click (as reported through the event receptor) that determines both which circle is darkened for feedback purposes, and which transition type is passed to the rendering of animation.

Paths are drawn and displayed using Garnet interactors. When a path is traced in the drawPath cell, the Garnet interactor capturing that trace causes the path to be both displayed in the drawPath cell, via a Garnet object, and defined as the fineTuning cell's formula, in the form of a list of coordinates. The display of the animation path in the Animation cell is also done using a Garnet object; the coordinates of that path are taken from the fineTuning cell if Drawn Path has been selected or are computed from the parameters given in Computed Path, if that option has been selected. So the drawPath cell (apparently) handles both input and output of a path, and the Animation cell (apparently) displayed both the animation and the path of animation. This apparent dual functionality of both these cells is achieved by superimposing one cell on top of another, with each cell handling a single task.

7.5.2 The Animation Cell's Temporal Vector

The external parameters necessary to produce animation are the five inputs from the Animation form -- Type, Object, Path, resetEvent, and continueEvent. Since the path is a relative path (i.e., each step indicates how to modify the previous animation frame), the previous value of the animation is used as an internal parameter. Once the animation has been defined, through formula definitions of the input cells, these external parameters are constant but the internal parameter representing the animation's previous value changes over logical time. Thus, each entry in the temporal vector for an animation sequence contains the animation's value at a point in logical time, which is then accessed when computing the value at the next point in logical time.

The value of a single entry in the animation's temporal vector is a tuple containing an image and an index, where the image is how the original object appears at that index along the relative path. To compute the next animation frame, the tuple corresponding to the previous animation frame is retrieved, and the incremental change indicated by the (index+1)st (x,y) pair in the path is applied to the image of the tuple. The new image and the new index are then combined to form a new tuple, which is then saved in the animation's temporal vector for the current logical time.

7.5.3 Implementing Animation Using Forms/3

Since we wanted to implement animation without adding new language concepts to Forms/3, our goal was to implement (or be able to implement) animation in Forms/3 using Forms/3 itself wherever possible. As small proof-of-concept examples have shown, it turns out that all portions of animation in Forms/3 except one -- fine-tuning the visually-drawn path -- can be implemented without leaving Forms/3. In other words, although we provided animation as a language primitive for simplicity and efficiency reasons, when doing so we were not changing the computational power of Forms/3 in any way (except for the noted exception).

The first proof-of-concept example we tried, movement animation, can be implemented using Forms/3's built-in 'compose' operator in a straightforward way. Essentially, a running total of the x- and y-coordinates of the path traversed thus far is kept, and the object is displayed (using the compose operator) at that point.

The second proof-of-concept example is intensity animation. The implementation of this example is similar to that of movement animation, except the simplicity gained by using the compose operator is no longer available. There are two apparent ways of overcoming this loss. The first is creating an 'intensityCompose' operator. Whereas the compose operator changes the location attribute of an object, the intensityCompose operator would change the intensity attribute of an object. This solution requires adding additional operators to the language[2], but would not change the underlying principles of the language at all. The second solution builds on the existence of a cell associated with the intensity of each primitive object; that cell is found on the primitive form for each

2. In addition to an intensityCompose operator, visibilityCompose and colorCompose operators would also be needed.

Forms/3 primitive object. If the formula for this intensity cell includes a reference to an intensity value computed by the animation, intensity animation is achieved.

The final proof-of-concept example uses event receptors to capture the user's drawn trace of the animation path. It turns out that event receptors can indeed be used for this task, but with a significant decrease in simplicity, efficiency, and precision when compared to using a Garnet built-in interactor to accomplish the same task. Specifically, the Garnet interactor is faster, captures more path points, and perhaps most importantly at this stage, can complete the task in one unit of logical time.

Lastly, we examine why the technique used to fine-tune a visually-drawn path does change the underlying language. Recall that when the user draws an animation path, the resulting list of coordinates is made available in the fineTuning cell. In other words, the fineTuning cell's formula is modified to become the list of coordinates. The theoretical Forms/3 language says that only a cell's value, and not its formula, may be affected by other cells. Our present technique violates this principle, since the drawPath cell changes the formula of the fineTuning cell, through a Lisp function. One solution is to implement *visual fine-tuning*, which would both solve the present problem and add another dimension of visualness to animation in Forms/3. It seems feasible that Garnet interactors and/or event receptors could be combined to produce such functionality, but this possibility has not been explored. However, fine tuning is a convenience only, and not an essential feature of the approach's functionality. Thus omitting it entirely would not impact the computational power of animation in Forms/3.

CHAPTER 8
CONCLUSION

The movement in algorithm animation has always been toward approaches that simplify the process of algorithm animation specification and design. Algorithm animation in Forms/3 takes another step along that path by extending Stasko's path/transition paradigm in a visual and declarative manner. Animation programming in Forms/3 is done visually through the Animation form, on which the user specifies the animation's inputs.

By seamlessly integrating algorithm animation into Forms/3, without leaving the declarative and visual model and without adding new concepts to the language, we have shown that algorithm animation need not compromise the characteristics of a declarative VPL, nor add complexity to the language. Rather, we can exploit the language's responsive and declarative characteristics to produce the following unique features of algorithm animation:

- on-the-fly exploratory algorithm animation programming (steering);
- the capability to change the direction of execution, of both the animation and the algorithm;
- separation of algorithm and animation code; and
- the generalization of the traditional time-based approach to advancing animation.

BIBLIOGRAPHY

[1] M.H. Brown and R. Sedgewick, A System for Algorithm Animation, *ACM Computer Graphics (Proc. SIGGRAPH'84, Minneapolis, MN)*, Volume 18, Number 3, July 1984, pp. 177-186. Reprinted in [15].

[2] M.H. Brown and R. Sedgewick, Techniques for Algorithm Animation, *IEEE Software*, Volume 2, Number 1, January 1985, pp. 28-39. Reprinted in [15].

[3] M.H. Brown. Exploring Algorithms Using Balsa-II, *IEEE Computer*, Volume 21, Number 5, May 1988, pp. 14-36. Reprinted in [15].

[4] M.H. Brown. Perspectives on Algorithm Animation, *Conference Proceedings, CHI'88: Human Factors in Computing*, Washington D.C., May 15-18, 1988, ACM Press, New York, pp. 33-38. Reprinted in [15].

[5] Margaret M. Burnett, *Abstraction in the Demand-Driven, Temporal-Assignment, Visual Language Model*, Ph.D. Dissertation, Department of Computer Science, University of Kansas, Lawrence, KS, August 1991.

[6] Margaret M. Burnett and Allen L. Ambler, Generalizing Event Detection and Response in Visual Programming Languages, *Proc. Advanced Visual Interfaces*, Rome, Italy, May 27-29, 1992, pp. 334-337. Also available as CS-TR 91-02, Michigan Technological University, April 1992.

[7] Margaret M. Burnett and Allen L. Ambler, A Declarative Approach to Event-Handling in Visual Programming Languages, *Proc. 1992 IEEE Workshop on Visual Languages*, Seattle, WA, September 15-18, 1992, pp. 34-40.

[8] Margaret M. Burnett and Allen L. Ambler, Interactive Visual Data Abstraction in a Declarative Visual Programming Language, *Journal of Visual Languages and Computing*, Volume 5, Number 1, March 1994, pp. 29-60.

[9] Bay-Wei Chang and David Ungar, Animation: From Cartoons to the User Interface, *Proc. ACM SIGGRAPH Symposium on User Interface Software and Technology*, Atlanta, GA, November 3-5, 1993, pp. 45-55.

[10] Kenneth C. Cox and Gruia-Cataling Roman, A Characterization of the Computational Power of Rule-Based Visualization, *Journal of Visual Languages and Computing*, Volume 5, Number 1, March 1994, pp. 5-27.

[11] Robert A. Duisberg, Animated Graphical Interfaces using Temporal Constraints, *ACM SIGCHI Special Issue, Proc. of Human Factors in Computer Systems*, Boston, MA, April 13-17, 1986, pp. 131-136.

[12] Robert A. Duisberg, Visual Programming of Program Visualizations: A Gestural Interface for Animating Algorithms, *Proc. 1987 Workshop on Visual Languages*, Linkoping, Sweden, August 1987, pp. 55-66. Reprinted in [15].

[13] R.A. Duisberg, Animation Using Temporal Constraints: An Overview of the ANIMUS System, *Human-Computer Interaction*, Volume 3, Number 3, 1987/1988, pp. 275-307. Reprinted in [15].

[14] *Garnet Reference Manual, Version 2.1*, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, November 1992.

[15] Ephraim P. Glinert (ed.), *Visual Programming Environments: Paradigms and Systems*, and *Visual Programming Environments: Applications and Issues*, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[16] Esa Helttula, Aulikki Hyrskykari, and Kari-Jouko Raiha, Graphical Specifications of Algorithm Animations with ALADDIN, *Proc. of the 22nd Hawaii International Conference on System Sciences*, Kailua-Kona, HI, January 1989, pp. 892-901.

[17] Esa Helttula, Aulikki Hyrskykari, and Kari-Jouko Raiha, Principles of ALADDIN and other Algorithm Animation Systems, in [18], pp. 175-187.

[18] Tadao Ichikawa, Erland Jungert, Robert Korfhage (eds.), *Visual Languages and Applications*, Plenum Press, NY, 1990.

[19] Sougata Mukherjea and John T. Stasko, Applying Algorithm Animation Techniques for Program Tracing, Debugging, and Understanding, *Proc. of 15th International Conference on Software Engineering*, Baltimore, MD, May 17-21, 1993, pp. 456-465.

[20] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal, Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces, *IEEE Computer*, Volume 23, Number 11, November 1990, pp. 71-85.

[21] G.-C. Roman and K.C. Cox, A Declarative Approach to Visualizing Concurrent Computations, *IEEE Computer*, Volume 22, Number 10, October 1989, pp. 25-36.

[22] G.-C. Roman, K.C. Cox, C.D. Wilcox and J.Y. Plun, Pavane: A System for Declarative Visualization of Concurrent Computations, *Journal of Visual Languages and Computing*, Volume 3, Number 2, June 1992, pp. 161-193.

[23] John T. Stasko, Simplifying Algorithm Animation with TANGO, *1990 IEEE Workshop on Visual Languages*, Skokie, IL, October 4-6, 1990, pp. 1-6.

[24] John T. Stasko, The Path-Transition Paradigm: A Practical Methodology for Adding Animation to Program Interfaces, *Journal of Visual Languages and Computing*, Volume 1, Number 3, September 1990, pp. 213-236.

[25] John T. Stasko, Tango: A Framework and System for Algorithm Animation, *IEEE Computer*, Volume 23, Number 9, September 1990, pp. 27-39.

[26] John T. Stasko, Using Direct Manipulation to Build Algorithm Animations By Demonstration, *CHI'91 Conf. Proceedings*, New Orleans, LA, April 27-May 2, 1991, pp. 307-314, published as Special Issue of *SIGCHI Bulletin*, ACM Press, May 1991.

[27] John T. Stasko and Doug Hayes, *XTANGO Algorithm Animation Designer's Package*, College of Computing, Georgia Institute of Technology, Atlanta, GA, October 1992.

[28] John T. Stasko and Charles Patterson, Understanding and Characterizing Software Visualization Systems, *Proc. 1992 IEEE Workshop on Visual Languages*, Seattle, WA, September 15-18, 1992, pp. 3-10.

[29] Shin Takahashi, Ken Miyashita, Satoshi Matsuoka, and Akinori Yonezawa, A Framework for Constructing Animations via Declarative Mapping Rules, *Proc. 1994 IEEE Symposium on Visual Languages*, St. Louis, MO, October 4-7, 1994, pp. 314-322.

[30] The 1994 Visual Languages Comparison, Wilfred J. Hansen, Moderator, *Proc. 1994 IEEE Symposium on Visual Languages*, St. Louis, MO, October 4-7, 1994, pp. 90-97.