1992 IEEE Workshop on Visual Languages, Seattle, WA, Sept. 15-18, 1992, pp. 34-40.

A Declarative Approach to Event-Handling in Visual Programming Languages

Margaret M. Burnett
Michigan Technological University

Allen L. Ambler
University of Kansas

Abstract

In this paper, we address the question of event-handling for declarative visual languages. In the approach presented, system-level, interactive, and user-defined events are fully-supported, while still maintaining the property of referential transparency. An approach to time termed temporal assignment *provides a unifying mechanism for events to be defined as ordinary sequences of values, and conversely for ordinary sequences of values to be defined as events. This allows event-handling without additional concepts, and in particular provides a natural means for the user to define higher-level events of any kind.*

1. Introduction

Declarative programming languages effectively utilize visual programming techniques because declarative programming languages focus on the relationships between data as opposed to control flow, and these relationships tend to be inherently multi-dimensional. Yet, declarative programming languages have weaknesses in other areas which limit their usefulness for visual programming. One such weakness is their ability to program actions based on events. Previous declarative visual programming languages have either avoided the issue, limited event recognition capabilities to only a set of "built-in" events, or departed from the declarative model to reach some compromise solution.

The approach described in this paper addresses the problem of event-handling in declarative VPLs. The distinguishing features of the approach are that it is fully declarative; it is completely supported by the visual language, without requiring any other language or tool; it treats events as normal values no different than other values; it is general; and its abstraction facilities support programming at a high level, supporting both user-defined high-level events and composition of events and event-handlers with other kinds of data. This generality and support for high-level proramming seem well-suited not only for traditional point-and-click user interactions, but also for more complex, gestural interactions such as might be found in pen-based interfaces.

Consider the following problem. An object within a window can be moved by grabbing it and flicking it off in some direction. As long as the mouse button is down, the object is actually being dragged. But if the mouse button is released while the mouse is still moving, the object continues to move in the direction and speed used while it was being dragged. The movement of the object continues until it arrives at a border, or until it is again grabbed. This problem is indicative of the kind encountered in any sort of gestural interface, which generally requires the ability to detect and abstract arbitrary sequences of events. Such sequences may be quite complex, going far beyond the capabilities of systems able only to recognize discrete button clicks, mouse movements, and keystrokes.

This paper is organized as follows. Sections 2 and 3 begin with a discussion of related work and the VPL Forms/3 used in this research. The fundamentals of the approach to event-handling are presented in Sections 4 and 5. In Section 6, we present a solution in Forms/3 to the example above. Section 7 discusses issues raised by declarative event-handling.

2. Related work

In most traditional approaches to events and event-handling, an event is regarded as a stimulus requiring the machine to make a response and enter a new state. These approaches are inappropriate for declarative languages, and would conflict with the notion of referential transparency.

Although declarative languages traditionally have not provided for event-handling, a few have done so through declarative approaches involving time-based semantics [5-8]. However, these approaches are not visual, and do not address the issues raised by interactivity in a visual language, but rather concentrate solely on system-generated events and interrupts.

Much of the work on event-handling in declarative VPLs has been done in special-purpose visual languages specifically pertaining to user interfaces. InterCONS [12] is a visual dataflow language which has certain primitives associated with user interaction (e.g. buttons, sliders). These primitives accept interactive input events and generate integer outputs, which can be routed into other dataflow nodes for further calculation. However, this approach is not general enough for gestural interactions such as in our window example; it is intended only for the specific problem domain of user interface generation. A similar approach can be found in ConMan [9] and Prograph [4]. Prograph also provides dataflow connectibility to the low-level Macintosh Toolbox, allowing direct access and manipulation of the various Macintosh data structures. This provides generality, but leaves the realm of high-level programming to do so.

In the dataflow VPL Fabrick [10], primitive graphical objects called graphemes (e.g.: rectangles, bitmaps) can be sensitized to certain kinds of interactive events. The sensors that are incorporated into these graphemes provide continuous tracking information, such as the mouse's position. This is similar in concept to our approach, and represents a more general approach to event-handling than UIMS-oriented VPLs, but still ties event detection to specific, viewable objects. Fabrick also supports certain non-interactive event-related data such as the time of day.

NoPump II [13] is a spreadsheet-like visual language with an emphasis on support for interactive graphics. While it does not include a general approach to events, it does handle certain events, namely system clock ticks and the relocation of objects on the screen. These events are supported through the ability to reference the event-supporting cells, one being the clock cell and the others being pairs of coordinate cells for graphical objects. The authors also propose adding an additional distinguished cell to report the state of the mouse's button. However, generality has not been a goal of the NoPump system, and the generality of the event-handling proposed is insufficient for complex interactions.

The work with the strongest influence on our approach is Schoberth's proposal [11] for event-handling in Forms/2 [1]. In Forms/2, a form is a collection of cells each of which has a formula defining its value and a set of attributes, which can be formulated in cells themselves. In an extension of this fact, events are represented by special attribute-cells, whose values are maintained and updated automatically whenever events occur in the cells they describe. Attribute-cells' values can then be referenced by formulas in other cells, thus contributing to calculations as needed. Our current approach to events is different from the Forms/2 approach, but borrows from its declarative, time-based aspects.

3. Programming language aspects of Forms/3

In order to facilitate discussion of our approach to events in the later sections, we briefly describe the programming language aspects of the VPL Forms/3 [2]. Forms/3 supports procedural abstraction, visual data abstraction, polymorphic typing with type inference, files, error-handling, and both interactive and non-interactive events. A research prototype runs on Sun 3/160 and SPARCstation 2 color workstations using Lucid Common Lisp and the CLX interface to X Windows.

Programming in Forms/3 consists of arranging and defining one or more *forms*. In Forms/3, a form is a collection of *cells* and/or groups of cells called *matrices* and *abstraction boxes*. Each has a formula which defines its value. A formula is a functional expression, and hence no side-effecting is possible. Because the program's calculations are entirely determined by these formulas, and because the formulas explicitly state the dependencies inherent in the calculations, it is not necessary for the programmer to specify sequencing or control-flow information; all sequencing information can be automatically derived from the dependencies. The language consists of a combination of cells containing formulas using a visual representation of text and graphics including direct manipulation and concrete, immediate feedback. Most of the formula operators describe obvious calculation functions. Informal descriptions of other operations used in this paper follow:

<u>earlier A</u>: At time t, results in the value of A at time t-1.
<u>if A then B else C</u>: Functional if-then-else -- returns B if A is true, otherwise returns C.
<u>x fby A</u>: Specifies a sequence of values starting with x and followed by the values computed by A.

3.1 The role of form instances in generalizing calculations

After the formulas on a form have been defined, other instances (copies) of the form can be created and modified. A form is normally named something mnemonic such as "Factorial". Besides its role in helping the user to informally identify a form, the form's name is also used internally by the system to construct a unique identification of the calculations being performed. For example, a copy of Factorial in which cell *N*'s formula has been changed to 10 is known to the system by the internal name "Factorial(N←<10>)", denoting "Factorial in which N's formula is <10>". This copy's internal name describes exactly how it differs from the original form Factorial. This provides a mechanism by which the system can request instances of forms needed for calculations -- by naming the form, the system has exactly the information needed to create the instance if it does not already exist. The user does not know about these internal names, but may request a new instance via direct manipulation.

3.2 Evaluation and temporal assignment

The computational dependencies between cells constrains evaluation ordering. Evaluation is lazy, i.e. a

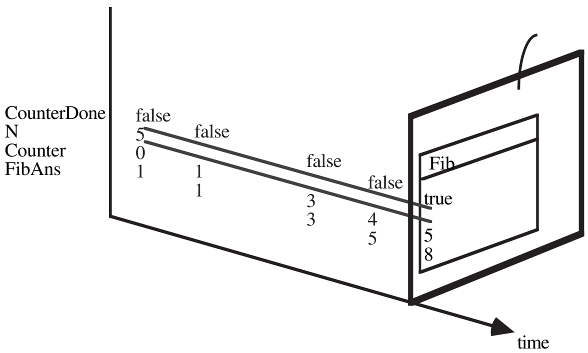

Figure 1

value is calculated only when it is required. A value may be required for one of two reasons. Either it is needed for the computation of another required value, or it is needed to produce a screen image. Screen images of all cells on the screen are by virtue of their presence required. Once a value has been calculated, the result may be saved for re-use in later calculations.

The formula for a cell actually defines a vector of values along a time dimension, rather than an atomic value. The idea of this notion of *temporal assignment* is that any given cell can define a single value or a sequence of values. If no cell's formula is expressed as a sequence over time, then only a single value for each cell will ever be computed. But if a cell's formula defines a sequence, each new computation in the sequence will also cause a cascade of further evaluations (if demanded) starting with those cells dependent on the original cell, and computing a new value for each dependent cell. This time-indexed model then produces for a given cell only a single value within each time interval, but over time it produces a vector of values called a *temporal vector*. A temporal vector is similar to a stream, but there is no notion of movement nor of consumption of its values.

Although it is usual for the user (and the system) to be interested mainly in the values of computations at the most recent unit in the time dimension, it is possible to view older values. When done interactively by the user, we term this *positioning the workspace*. Conceptually, the workspace is a large, strictly two-dimensional pane of glass with one-way visibility positioned somewhere in time, providing a vantage point from which forms and cells can be viewed, as depicted in figure 1.

4. A temporal approach to events

The temporal assignment principle provides the foundation for our approach. For simplicity, suppose for a moment that the formula for cell *X* is simply the event sequence in that cell. Since the sequence of events for cell *X* is a temporal vector of values from a unique queue of events, events do not compromise the referential transparency property. This will be discussed in more detail later.

We will consider first events not generated by user interaction. The other case, events relating to user interaction, will be addressed in section 6. Examples of non-interactive events include system clock ticks, input from a sensor port, printer-out-of-paper interrupts, etc. This kind of event fits naturally into the temporal model, because an event at time t can be defined as the presence of a value at time t in a temporal vector whose formula is some event queue.

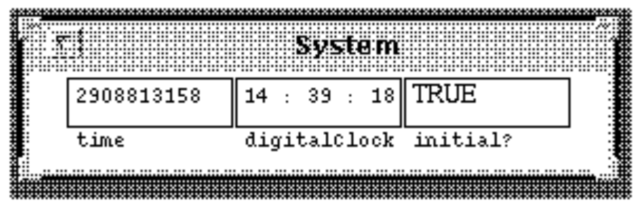

Figure 2

Cells with this event information are located on a distinguished primitive form called System (figure 2). System contains cells reflecting the state of the system and the environment. In theory, form System would contain a cell for each type of interrupt. (Our prototype at present supports only one type of system interrupt, namely clock ticks). Cell *time* defines a temporal vector of numbers, each of which reflects the (wall) time to the nearest second, in seconds past January 1, 1900. Form System also contains some additional cells for convenience

The cells on form System can be referenced in the usual way by other cells; hence the presence of a value in the temporal vector of one of these cells generates the needed event-handling automatically. To see why this is so, consider the evaluation model. If an event happens at time t, the information associated with it becomes the value of the temporal vector for the cell *E* associated with that event type at time t. Because of the dependency-driven nature of the language model, this implies that any cell *X* whose formula references cell *E* will also have an entry defined in its temporal vector at time t (unless *X* chooses to ignore selected events, as discussed in the next section). This is a declarative approach whose semantics are functionally equivalent to the more traditional imperative notion, expressed as:

> "whenever an event of interest to *X* occurs, *X* is activated and performs the actions needed."

5. Events as values and values as events

Although the previous discussion was conducted in the context of system-generated events, in this section we will see that there is no distinction between system-level events, ordinary calculations, and higher-level events.

These values can all be combined and composed as desired without restriction. This fact carries the implication that event-based programming and traditional calculation-based programming are indistinguishable from one another.

The ability to combine any combination of events and ordinary data may be used to create any arbitrary new kind of data, including higher-level events. A formula operator that is often used for this purpose is the if-then construct. It is similar to if-then-else, except that there is no "else", meaning that if the condition is not true at time t, then no new value is defined at time t. Thus, a formula can define a value on the condition that the cells to which it refers turn out to have "interesting" values. For example, events generated by the calculation of even numbers can be defined via the formula of some cell *Even?* as:

```
if answer/2 = truncated (answer/2)
        then "Even"
```

If cells *X* and *Y* depend on cell Even?, then if (and only if) a new entry in the temporal vector for *Even?* is defined, then an entry for *X* and *Y* (and everything dependent on them as well) will also be defined. This simulates the traditional notion of an event, because it allows the definition of a value only when something of interest has happened, providing the capability of higher-level events to be defined from sequences of lower-level events or any arbitrary values. This uniform treatment of data as events and events as data provides the user the flexibility to write programs using an event-driven philosophy even when no events (in the traditional sense) are involved.

6. Interactive events

Although user-interaction events could also be handled using the same mechanism as system events, such an approach would centralize interactive event-processing. Instead, we choose an approach which distributes the event-handling of interactive events to the objects for which those events were intended. To this end, we define a primitive data type to handle interactive events, called an *event receptor*. It is the only object in the language capable of receiving interactive events. Conceptually, an event receptor is an invisible window which when placed strategically on the screen will receive events over that portion of the screen. Like any data type, it can be composed with other types, allowing the user to define interactive objects with any desired behavior.

The essence of an event receptor is an invisible image and temporal vectors of pertinent event information. An event receptor is instantiated via a copy of the built-in visual abstract data type form shown in figure 3. Using this form, the user can specify the events of interest, the shape of the region, the desired amount of transparency, and the name describing the use of the event receptor. (All data types in Forms/3 are defined via visual abstract data type definition forms, and the types themselves are termed *visual abstract data types (VADT)*. A discussion of visual abstract data types is given in [2]). The capabilities and uses of event receptors are given in [3]. In this paper we focus on how it fits into the overall declarative approach, and in particular how it preserves referential transparency.

The example introduced earlier in which objects in a window can be moved via dragging and flicking gestures demonstrates many of the features of our approach. The higher-level events dragging and flicking are defined, using the integrated approach to event-handling and calculations. Although we have kept the example very simple, it gives some sense of the flexibility inherent in the approach.

We will use an event receptor to program a VADT which we will call an ImageMover. Instances of this new type can then be placed directly on a window and moved, or they can be incorporated into other types (customers, stacks, etc.), to give them the same movability properties.

Like all programming in Forms/3, defining the new VADT ImageMover involves placing cells, matrices, and abstraction boxes[1] on a form and defining their formulas. The formulas are entered by the user using a combination of direct manipulation (pointing at cell references, selecting forms from menus, etc.) and text. On a color workstation, they are displayed using a combination of text and color-encoding[2]; they are shown in this paper using a simulation of that notation without the color-encoding. The form in figure 4 defining an image mover refers to several cells in the EventReceptor form of figure 3. Formulas are shown superimposed over the screen dump. Cell *er*'s formula is a reference to cell *EventReceptor* in figure 3. The person eating ice cream is an instance of type Glyph. Glyphs are drawn using a normal bitmap editor provided by the window server.

Once the VADT ImageMover has been defined, new instances can be created as discussed in section 3. The behavior of these imageMovers is affected by the formulas defined for the parameter-like cells at the top of the figure. The actual implementation details of an instance of an image mover are encapsulated in the abstraction box *aMover*. The cells at the bottom of the form provide access to information about an image mover. Cell *image* is a distinguished cell which defines the appearance of an instance of an imageMover; thus any cell on any form which contains an image mover uses the user-defined

[1]The purpose of an abstraction box is to encapsulate the implementation of a data type. In the figure, *aMover* is an abstraction box. Its formula definition is simply the collection of cells it encloses.

[2]The implementation does not yet include this capability.

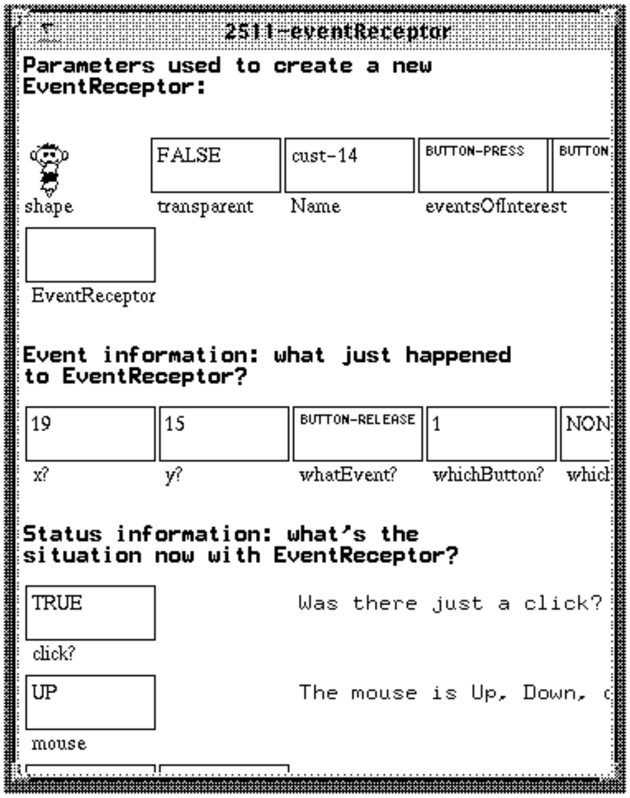

Figure 3

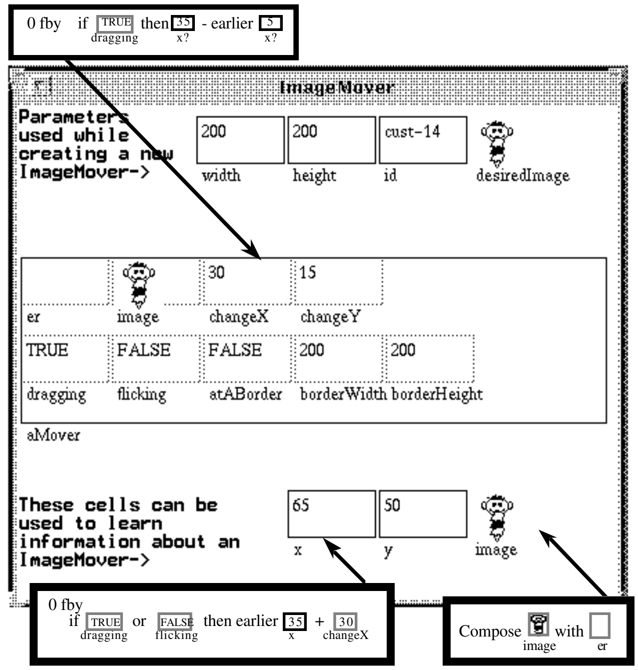

Figure 4

formula in cell *image* on an instance of form imageMover to determine how it is to be displayed.

Cells *dragging* and *flicking* demonstrate the ability to define higher-level events, as well as the use of values as events. *Dragging* becomes true if there is a Motion-Notify event on the object's event receptor while the mouse is down, and becomes false if there is a Button-Release event:

false fby
  if $\underset{\text{whatEvent?}}{\boxed{\text{MOTION-NOTIFY}}}$ = Motion-Notify and $\underset{\text{mouse}}{\boxed{\text{DOWN}}}$ = Down then true
  else if $\underset{\text{whatEvent?}}{\boxed{\text{MOTION-NOTIFY}}}$ = Button-Release then false

Thus, this cell defines new values only when the desired combination of events and computations mean that dragging is initiated or terminated. This is a higher-level event which has been defined by the user as an abstraction of a sequence of low-level interactive events and internal calculations. The representation of these events however, is just a simple boolean true or false. These values affect all "interested" cells in the same way as any event because any formula that references cell *dragging* will automatically define new entries in its own temporal vector whenever a new entry occurs in *dragging*'s temporal vector. As with all values under the lazy evaluation model, these temporal vector entries are defined, but not actually evaluated unless they are demanded.

Similarly, *flicking* becomes true if the button is released while there is still motion, and *dragging*'s previous value was true. It becomes false if there is a Button-Press event on the object, or its x-y position collides with a border. The remaining cells are straightforward. The cell *id* serves a parameter-like function, and requires a unique number passed in by the caller. This unique number is passed on to "name" the corresponding event receptor which is a part of each image mover, as discussed in section 7. The cells *width* and *height* describe the size of the window, and *desiredImage* tells the imageMover how it should appear on the screen. This *desiredImage* cell is also passed on to the event receptor to indicate its shape, as is the fact that it is to be sensitive to mouse and button movements.

The additional text which appears on the form, "Parameters used while..." and "These cells can be used...", are in fact cells with (constant) formulas also. The user can impose formatting characteristics (color, size, border, etc.) on any of the cells. These two particular cells have been formatted using the "8x13bold" font, and no border.

The window itself is also defined as a new VADT, as in figure 5. Its formulas are very simple. Its image cell (bottom of the figure) is simply the composition of a box with all the objects at their computed x- and y-positions. The objects are references to instances of type imageMovers (or of other types in which imageMovers have been composed), and the x- and y-positions are references to cells *x* and *y* on the corresponding instances of form imageMover. Because all instances of type

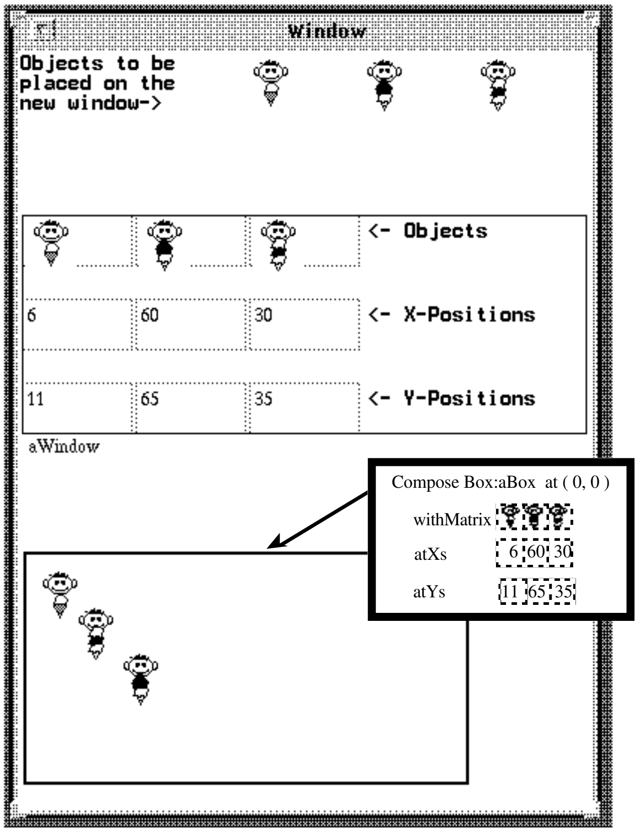

Figure 5

window are displayed via the formula in the image cell, the objects can be dragged and flicked around as desired.

7. Event receptors, evaluation, and referential transparency

The behavior of event receptors is relatively lazy. To see why this is true, recall that the only calculations that are demanded are those required to compute the images on the screen. Since it is impossible for the user to click on or otherwise interact with something that isn't on the screen, this evaluation rule serves interactive event processing well. An event receptor whose image is not on the screen will rest lazily in the background, and when on the screen will be continually demanded to respond to user interactions within the confines of its image.

The Name parameter on the event receptor's definition form, in addition to its informal documentation purpose, also plays a necessary role in maintaining referential transparency. Recall that under referential transparency, if all incoming parameter-like values to a form are the same (which are reflected in the internal form names), then they describe exactly the same calculation. Now suppose the user wants to construct two different event receptors (one to be composed with a button and one to be composed with a menu), both shaped identically and both responding to only Button-Press and Button-Release events. If there are no differing formulas, the two seemingly identical requests for such an event receptor (i.e. two requests with identical internal names) would have to have identical event sequences by the principle of referential transparency. This apparent difficulty arises because there is another "parameter" which is needed to describe the calculation completely–namely its input stream of events.

From a theoretical standpoint then, it is necessary to include the event sequence as part of the internal description of the calculation. From a practical standpoint this is not possible, since the values of the events are not known in advance. The Name cell allows the user to identify the forthcoming sequence of screen interactions in a concrete fashion, just as filenames in traditional languages allow naming of forthcoming external values and updates. In the example above, the user might choose to formulate one event receptor's Name cell as "ButtonEvents" and the other as "MenuEvents", forcing a difference in the description of the two calculations.

7.1 Evaluation of an ImageMover and Window

Suppose that an image mover exists and that its image is on the screen in an instance of type Window. This means that the image of an event receptor is on the screen too (since it is part of the image mover's image). If the user generates mouse events on the image mover, the event receptor will receive that information via its image, into its event queue.

Suppose the user's interaction so far has been a button press, motion, and button release while moving the mouse. Each of these events adds an entry to the temporal vector that is defined by the cell *whatEvent?*, which in turn affects the event receptor in cell *er*. Since *er* is part of the image mover, new values for it and for the containing image mover are defined. These in turn define new *changeX, changeY, x*, and *y* values. Since *x* and *y* are used by the window in calculating its composed image, a new value for it too will be computed. Since the most recent value of the window's image will be displayed, the appearance of the screen will be updated as a result of this event sequence. Figure 6 depicts the effects of these manipulations over time. The window form shows the values that would be displayed on the screen as of the most recent moment in time, and the values along the arrows show their progression over time. Notice that the only values that are defined along the time dimension are those affected by the user's interaction with the first object. Since that object includes an event receptor (with an event history), new values for that object, as well as its x- and y-position and the final image are defined.

Note that since an event receptor includes in its definition a mask of events to which it should respond, only relevant events will cause any new value definitions–

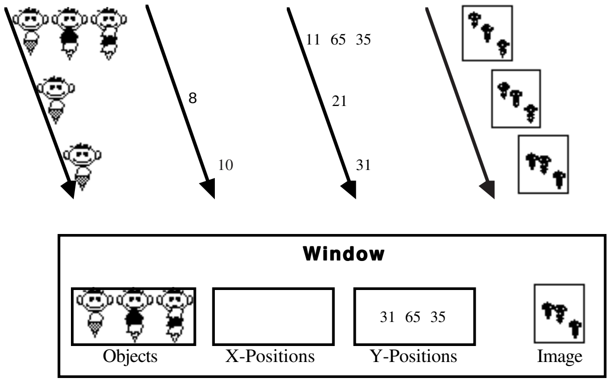

Figure 6

events not of interest to that event receptor do not affect its temporal vector. Also the user program can obscure an event receptor with some other object or cease displaying its image if the event receptor is no longer useful. When the event receptor is not visible, interaction with it cannot occur and its temporal vector cannot acquire new entries.

7.2 Multiple active images of an event receptor

Now suppose that, in addition to the image mover's image on the screen in some *cellX*, the form defining the particular image mover instance is also on the screen. This provides a more detailed view of the same object, which in particular has the same event receptor as one of its parts. This is an example of a single event receptor which has multiple active images on the screen -- imageMover:*image* and *cellX* both display an image of the same event receptor. The issue is what effects there are on the other if there is an event in one of the two images.

The answer becomes clear when we consider the way the event receptor is defined. An event is a value in the system-maintained event queue temporal vector for the event receptor. Because the event receptor contains a reference to that queue, any event in that event queue affects the event receptor, and resultantly the cells on form imageMover and *cellX*. Since the event receptor's image is the conduit from the user to the event queue, and since these two event receptor images on the screen are both conduits to the same event queue, then an event in either of the two images affects the event queue and all the objects that depend on it.

8. Conclusion

Although interactivity plays a major role in the <u>process</u> of programming with visual languages, most declarative visual languages have not fully supported interactivity in the programs <u>produced</u> by those visual programming systems. In this paper we have addressed that problem by presenting an approach to event handling in declarative visual languages. The approach is consistent with the high-level, visual process of programming that is the goal of many visual languages, and is consistent with the declarative model. Referential transparency is preserved, as is a lazy philosophy of evaluation.

A significant feature is that programming with events is no different than any other kind of programming. Since events are simple values and conversely simple values are events, additional concepts are unnecessary to a user wishing to perform event-related programming. This allows a consistent, visual, high-level approach to event-related programming.

References

[1] A. Ambler, M. Burnett, Visual forms of iteration that preserve single assignment, Journal of Visual Languages and Computing, 1(2), 159-181, 6/90.

[2] M. Burnett, Abstraction in the demand-driven, temporal-assignment, visual language model, Ph.D. Thesis, Univ. Kansas, 8/91.

[3] M. Burnett, A. Ambler, Generalizing event detection and response in visual programming languages, Proc. Advanced Visual Interfaces, Rome, Italy, May 27-29, 1992. (CS-TR 91-02, Mich. Tech. Univ., 4/92).

[4] P. Cox, F. Giles, T. Pietrzykowski, Prograph: A step towards liberating programming from textual conditioning, 1989 IEEE Workshop on Visual Languages, Rome, Italy, 150-156, Oct. 4-6, 1989.

[5] R. Dannenberg, Arctic: A functional language for real-time control, Conf. Record 1984 ACM Symp. on Lisp and Func. Prog., Austin, 96-103, 8/84.

[6] J. Darlington, L. While, Controlling the behaviour of functional language systems, Proc. Func. Prog. Langs. & Comp. Arch, Portland, 9/87.

[7] A. Faustini, E. Lewis, Toward a real-time dataflow language, IEEE Software, 29-35, 1/86.

[8] T. Gautier, P. LeGuernic, SIGNAL: a declarative language for synchronous programming of real-time systems, Proc. Func. Prog. Langs. & Comp. Arch., Portland, 9/87.

[9] P. Haeberli, ConMan: a visual programming language for interactive graphics, Computer Graphics, 8/88.

[10] D. Ingalls, S. Wallace, Y.-Y. Chow, F. Ludolph, K. Doyle, Fabrik, A visual programming environment, Proc. OOPSLA 88, San Diego, 176-190, 9/88.

[11] A. Schoberth, Event handling in a demand-driven visual language preserving single assignment, Master's Thesis, Univ. Kansas, 1990.

[12] D. N. Smith, The interface construction set, in *Visual Languages and Applications*, (T. Ichikawa, E. Jungert, R. Korfhage, eds.), Plenum Pub., NY, 1990.

[13] N. Wilde, C. Lewis, Spreadsheet-based interactive graphics: from prototype to tool, Proc. CHI '90, 4/90