To appear in ACM Proceedings of AVI'96: International Workshop on Advanced Visual Interfaces, Gubbio, Italy, May 27-29, 1996.

A SEAMLESS INTEGRATION OF ALGORITHM ANIMATION INTO A VISUAL PROGRAMMING LANGUAGE

Paul Carlson, Margaret Burnett[†]*, and Jonathan Cadiz*

Department of Computer Science, Oregon State University
Corvallis, Oregon 97331-3202 USA
E-mail: burnett@cs.orst.edu

ABSTRACT

Until now, only users of textual programming languages have enjoyed the fruits of algorithm animation. 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. To begin solving this shortcoming, we have seamlessly integrated algorithm animation capabilities into Forms/3, a declarative VPL in which evaluation is the continuous maintenance of a network of one-way constraints. Our results show that a VPL that uses this constraint-based evaluation model can provide features not found in other algorithm animation systems.

1 : INTRODUCTION

Algorithm animation is a type of software visualization of growing importance. It is a dynamic visualization of the main abstractions of a program's underlying algorithm. The value of algorithm animation lies in its ability to portray the essence of the program's logic, avoiding the obscuring of this essence that comes from detailed visualization of a program's data structures and variables. With this in mind, it is clear that a visual programming language (VPL), although it may already contain numerous visualization details, may have as much to gain as do textual programming languages from algorithm animation. Surprisingly however, our literature search revealed no prior attempts to bring algorithm animation capabilities to VPLs.

This paper describes how we integrated a visual and declarative extension of an algorithm animation model called the path/transition paradigm [Stasko 1990b] into the VPL Forms/3 [Burnett and Ambler 1994], a declarative VPL in which evaluation is done by maintaining a network of one-way constraints. Our goal was not a special-purpose visual algorithm animation system for animating algorithms implemented in other languages. Rather, our goal was to use this type of declarative VPL to animate the algorithms that are themselves programmed in that declarative VPL.

At first, we simply wanted to see if a declarative VPL following a constraint-based evaluation model could support such an integration with algorithm animation without compromising fundamental characteristics of a constraint-oriented VPL. What we discovered was that not only are those characteristics not compromised, but that combining a visual system with a constraint-oriented evaluation model can produce an approach to algorithm animation with unusual features.

1.1: Organization of this Paper

After a discussion of related work in Section 2, we introduce our constraint-based approach through two examples in Section 3. The first example shows how an animation is programmed using one-way constraints in our VPL, and the second shows how this approach to animation is used to animate algorithms. Section 4 discusses the issues and implications of the approach. We conclude after a discussion of the current status and future work.

2 : RELATED WORK

Balsa (Brown ALgorithm Simulator and Animator) [Brown and Sedgewick 1984] was a pioneering textual algorithm animation system which influenced many later approaches. Other early algorithm animation systems included Animus [Duisberg 1987/88], which is a textual system that incorporates temporal constraints to simplify the programmer's task of constructing animations of algorithms programmed in a textual language; another system by Duisberg that incorporates a gestural interface in a system for animating textual programs [Duisberg 1987]; and ALADDIN [Helttula et al. 1989], in which all inputs to the animation of textual programs are specified visually.

[†]This work has been supported in part by the National Science Foundation under grant CCR-9308649 and an NSF Young Investigator Award.

Stasko contributed an animation model with precise semantics called the path/transition paradigm [Stasko 1990b]. 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. Since the path/transition paradigm is the basis of animation in Forms/3, it is discussed further throughout this paper and in particular detail in Section 4.1.

To implement the path/transition paradigm, Stasko developed a textual algorithm animation system called Tango [Stasko 1990a]. Algorithm animation construction using 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. Figure 1 provides an example of this textual imperative approach to algorithm animation. The top half of the figure shows a portion of an algorithm with a Tango animation function call, and the bottom shows the associated animation function, which the programmer also codes.

Stasko developed a direct-manipulation, by-demonstration interface to the Tango system called Dance (Demonstration ANimation CrEation) [Stasko 1991]. 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 our approach to animation is using the trace of the mouse to specify a path.

```
...
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);
           }
...
```

```
void  ANIMExchange(p1,p2,p3,p4)
  int    p1, p2, p3, p4;
{
  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 1: Animation code is unshaded. (Top): A portion of an animation-annotated algorithm to swap two array elements. (Bottom): The associated Tango animation function.

A more recent research project from Stasko's research group is a visual debugging tool called Lens [Mukherjea and Stasko 1993]. Lens' focus is allowing the programmer to rapidly develop animations of textual programs for debugging purposes. Use of Lens doesn't require any textual programming of animation code; all specification and design of the animation is done in a visual environment. Other research projects that use visualization techniques to debug textual languages include Provide [Moher 1988] and ZStep 94 [Lieberman and Fry 1995].

The Garnet system [Myers et al. 1990] was the among the first to use one-way constraints to declaratively control behavior. However, Garnet's support is for user interface construction, and does not extend to algorithm animation. (Indeed, the user interface for our system is implemented in Garnet, although the algorithm animation aspect of our system was hand-implemented in Lisp.) More recently, a few researchers have extended declarative visualization models, which map program state to geometric objects, to include an animation component [Roman et al. 1992; Takahashi et al. 1994]. The programmer specifies animation through textual, rule-based notations; it is only the declarative characteristics that provide some similarity to animation in Forms/3.

3: ANIMATION PROGRAMMING IN A VPL VIA ONE WAY CONSTRAINTS

In order to support algorithm animation, a language must first support animation programming. We introduce our approach to algorithm animation through two examples. The first example in this section describes animation programming using one-way constraints in Forms/3. The second example shows how this approach to animation programming is used to animate an algorithm.

3.1: One-Way Constraints in Forms/3

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. (In our current prototype, these references are reflected textually in the formulas.) A program's calculations are entirely determined by these formulas. The formulas combine into a network of one-way constraints, and the system continuously ensures that all values displayed on the screen satisfy these constraints.

As this brief description shows, Forms/3 is declarative and responsive. By declarative, we mean that programming is a matter of defining the relationships between the inputs and the desired outputs. By responsive, we mean that whenever any new piece of information enters the system—such as when the system computes new data or the programmer changes a constraining formula—the effects are immediately and automatically reflected in the displayed portion of the program's results. These two properties, declarativeness and responsiveness, are important to the way animations are programmed under our approach, because the programmer needs to worry only about *specifying* the animation, and may ignore details such as how to update changing parts of the animation on the screen and how to synchronize with the algorithm being animated.

3.2: An Animation Example From the User's Point of View

Algorithm animation uses a variety of effects to communicate the essence of an algorithm such as smoothly moving bitmaps, gradual color changes, and fading images. The first example, a rolling wheel bitmap, will provide a fine-grained look at how Forms/3 can be used to produce such animated effects.

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 [Hansen 1994]. Figure 2 shows the user's view of the Forms/3 program that produces that animation.

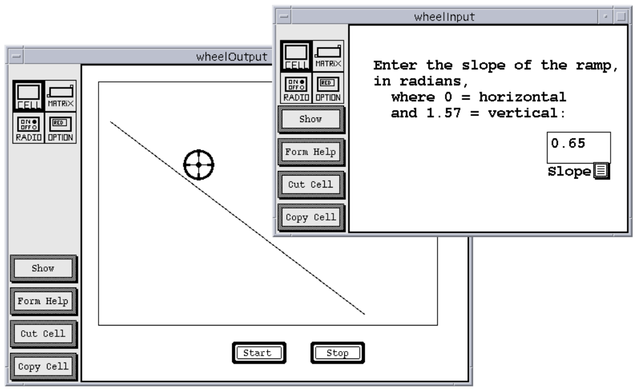
The user changes the slope by clicking on the Slope cell's formula tab and then entering a new slope as the cell's formula. (In general, the user inputs to a Forms/3 program are (1) cells whose formula tabs are visible and (2) interactive devices such as buttons.) The wheelOutput form shows the wagon wheel rolling down the ramp. The user clicks on the Start and Stop buttons to control the animation. Forms/3 supports generalized events, as described in detail in [Burnett and Ambler 1994], which enables the programming of such buttons.

3.3: Fundamentals of Animation Programming in Forms/3

Our approach to animation programming follows the path/transition paradigm's concept of animating an object along a path. In our adaptation of this paradigm, a path defines a finite sequence of x-y coordinate pairs; the pairs are interpreted as an offset from a previous value. 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 specification parameters for a transition.

The third parameter to a transition is the transition type, such as movement, intensity, visibility, or 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 last two parameters to a transition are the reset event and the continue event. The reset event specification gives the constraints that must be satisfied for the object to (re-)start at the beginning of the path; the continue event specification gives the constraints that must be satisfied for the object to traverse the next step along the path.



Enter the slope of the ramp,
in radians,
where 0 = horizontal
and 1.57 = vertical:

0.65

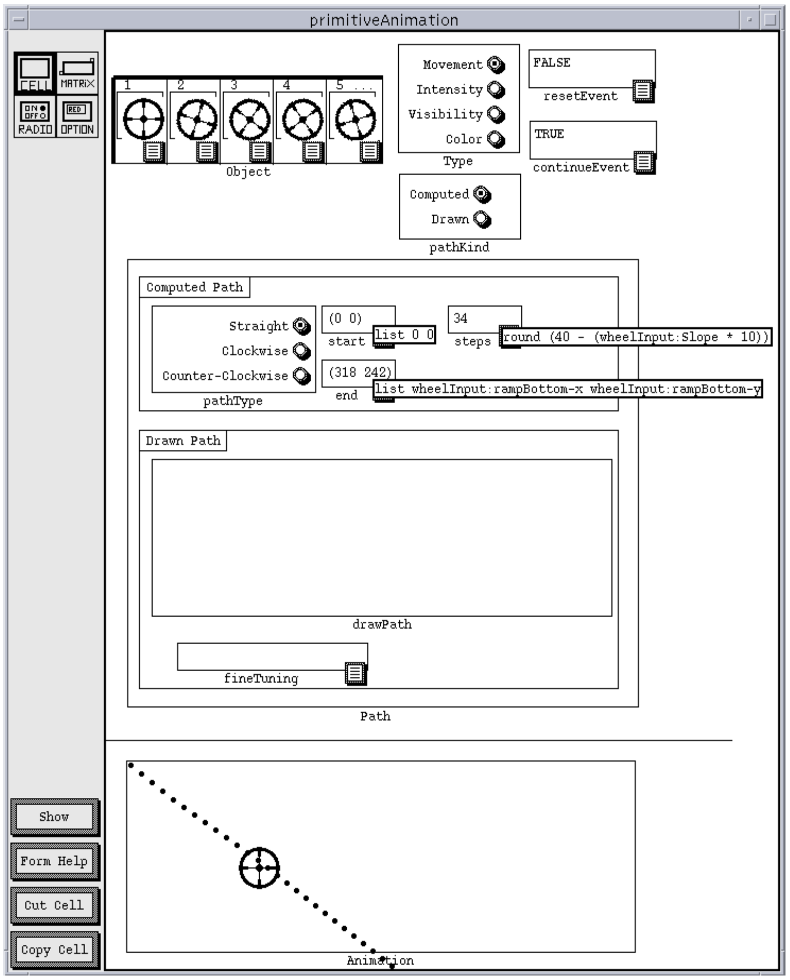
Figure 2: Rolling wagon wheel program.

3.4: The Wheel Example From the Programmer's Point of View

In Forms/3, animation is visually programmed following the above principles by specifying the animation parameters on a form called the Animation form. Figure 3 shows the Animation form for the rolling wagon wheel program. The Object matrix and the Path, Type, resetEvent, and continueEvent cells specify the constraints on the animation; the Animation cell at the bottom of the form is the resulting rendering of the transition of the object along the path. We will discuss each of these cells as it pertains to the rolling wheel.

The Object matrix defines the object(s) to be animated. An object can be any type: primitive objects such as boxes, glyphs, and text strings; user-defined graphical objects such as people or stacks; or arbitrarily complex objects resulting from other calculations. When the Object matrix includes more than one object (as in Figure 3) the animation repeatedly cycles through the matrix, displaying one object per animation frame. This allows animation types that are not built in, such as rotation. For example, the differences between the five glyphs of the Object matrix cause the wagon wheel to rotate as it moves down the ramp. The programmer can set up the Object matrix to be any size desired, and if the matrix is too large to fit nicely, the remaining elements are truncated and replaced with an ellipsis.

The programmer has selected a Computed path that is Straight via radio buttons with these names. The starting and ending points of the wheel's path need to match the start and end of the ramp. Thus, like the ramp (not shown in the figure), the path is defined to start at (0 0). The formula for the path's ending point constrains the ending point of the path to the ramp's ending point. (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 on the referenced cells.) The formula for the number of steps in the path is also shown in Figure 3. The formula effects an inverse relationship between the number of steps and the slope, to make the wheel move faster down a more steeply sloped ramp. Alternatively, instead of a computed path, the programmer could have drawn the path with the mouse inside the drawPath cell, as in Figure 4. When the mouse button is released, the list of offsets constituting the path is placed in the fineTuning cell, where the programmer can make precise adjustments.



Type: [✓ Movement] Intensity Visibility Color

resetEvent: FALSE

continueEvent: TRUE

pathKind: [✓ Computed] Drawn

pathType: [✓ Straight] Clockwise Counter-Clockwise

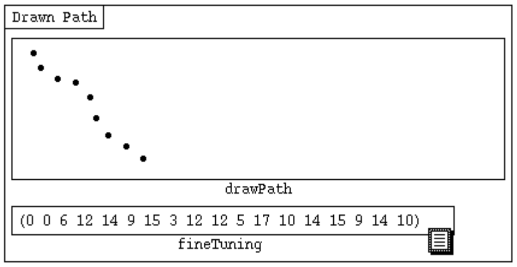
start: (0 0) — `list 0 0`

steps: 34 — `round (40 - (wheelInput:Slope * 10))`

end: (318 242) — `list wheelInput:rampBottom-x wheelInput:rampBottom-y`

fineTuning:

Figure 3



fineTuning: (0 0 6 12 14 9 15 3 12 12 5 17 10 14 15 9 14 10)

Figure 4

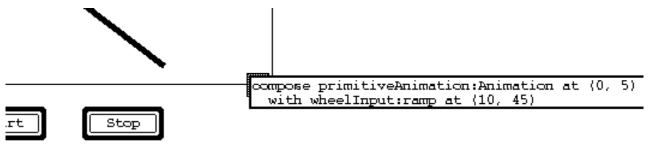
Figure 3 shows the Animation form for the rolling wagon wheel program. If the programmer had chosen to draw the path instead, the middle portion of the form might have looked as shown in Figure 4.

The remaining constraints on the animation are the Type cell, which is specified by selecting a radio button at the top of the form, and the resetEvent and continueEvent cells, whose formulas constrain the animation's initiation and advancement to the proper combination of time passing and button presses; this concept will be discussed in Section 4. The Animation cell in Figure 3 shows the wagon wheel after it has rolled a few steps along the path. In the figure, resetEvent's displayed value is currently false because it is past the time to start the animation at the beginning; continueEvent's displayed value is currently true because the animation is in progress.

3.5: Combining Animations

Different types of animations can operate on the same object at once. This is done by constraining the input of one animation (in the formula for a cell in the Object matrix) to be the output of another animation, by referring to another Animation form's Animation output cell. Using this technique, for example, a circle might change intensity while it is moving.

Because the result of an animation is a sequence of ordinary values, these sequences can be combined through formula references with other value sequences into a single scene using



```
compose primitiveAnimation:Animation at (0, 5)
  with wheelInput:ramp at (10, 45)
```

Figure 5: This portion of the wheel form shows the formula used to specify Figure 2's combined display of the animated wheel with the ramp.

the normal operators of the language. For example, Forms/3's 'compose' operator can be used to define combinations of animated (and non-animated) value sequences, as in the formula in Figure 5. The specifications in such a network of constraining formulas provide the constraint satisfaction mechanism with enough information to automatically synchronize multiple animations in a single scene.

3.6: Algorithm Animation Example: Animating a Sort

The same technique for creating animations is used to create algorithm animations. For example, Figure 6 shows an animation of a selection sort, and Figure 7 shows a user's view of the sort itself. Each bar in Figure 6 corresponds to an element from the input matrix from Figure 7. The programmer has constrained the height of a bar to be directly proportional to its corresponding element's value.

The programmer animated this algorithm by programming a copy of the Animation form for *one* element in the matrix in a manner similar to that seen in the wagon wheel example. The programmer's only other tasks were to add the line (to serve as a visual orientation cue), and to define the formula for the output matrix's cells shown in Figure 6 by simply clicking on the animated object in the Animation form copy. From these actions, the system knows the relationship between the first animated object and the sort's first input element, and automatically generalizes so that the other elements of the output matrix in Figure 6 display animated objects for the remaining sort input elements. (This generalization is done using an extended version of the generalization method described in [Yang and Burnett 1994].) The automatic generalization combined with the fact that matrices in Forms/3 are flexible—they grow and shrink like lists—allow the sort program and its animation to work for any arbitrary number of elements.

Figure 8 shows the Animation form for the first position—which holds the 8 element—that is the last to be moved into the sorted group. The bar is a box whose height depends on the value of the element being moved, and whose width is 35. (If the programmer had wished to limit the screen real estate of the entire animation to be, say, 600 pixels wide, the box's width could have been defined to be the maximum of 35 or the result of dividing 600 by the total number of input elements.) The path's endpoint determines whether the bar moves to the left or the right relative to where it started. It will move left if its destination, the current size of the sorted group (available by referencing sorted[numcols]), is less than the position at which it started (whichPosition); otherwise it will move right. continueEvent is true when this element is being moved, and resetEvent is true only at the beginning of the program's execution history (reported by cell Initial? on Forms/3's built-in System form). When the move into the new group is completed, the continueEvent will become false.

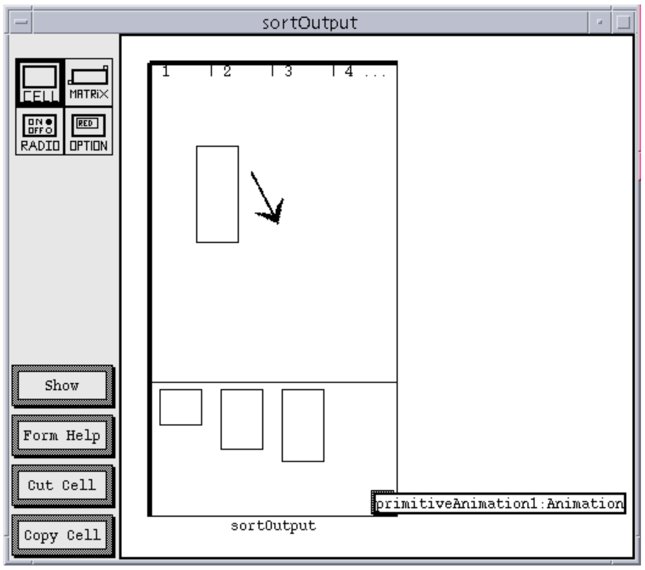


Figure 6: Animation of a selection sort. The arrow is superimposed on the screen shot to show the direction of motion. The formula shown is automatically generalized so that it can be used by every entry in the matrix.

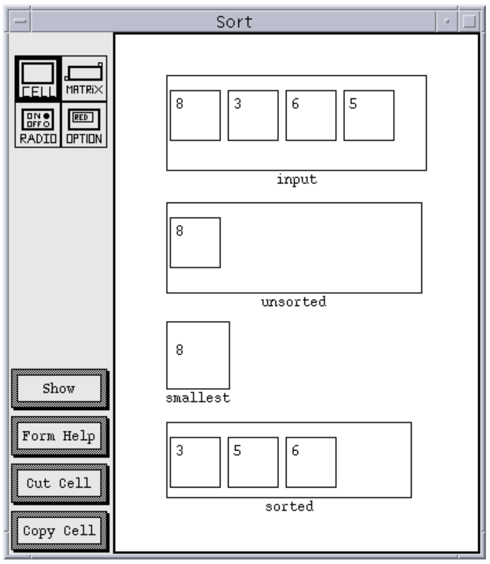
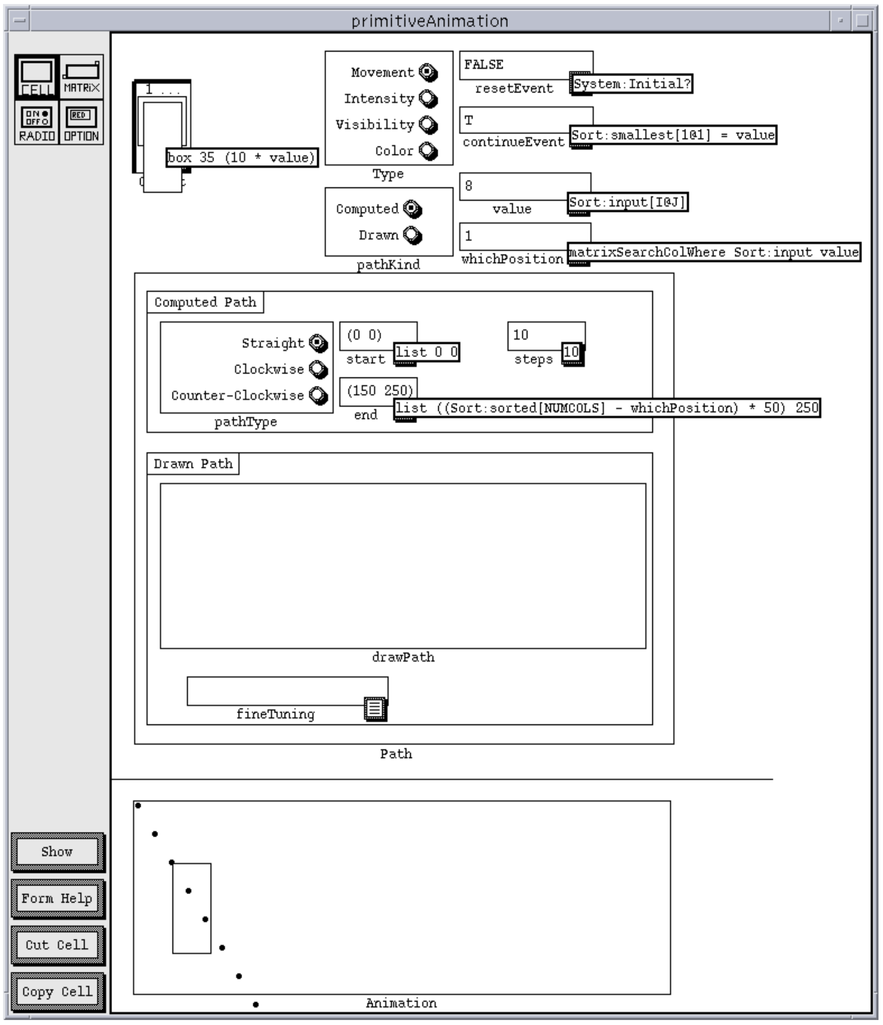


Figure 7: A user's view of the selection sort during execution. The sort constrains each new element of the sorted group to be the smallest of the unsorted group.



box 35 (10 * value)

Movement [✓ Movement] Intensity Visibility Color

[✓ Computed] Drawn

resetEvent: FALSE — System:Initial?

continueEvent: T — Sort:smallest[1@1] = value

value: 8 — Sort:input[I@J]

whichPosition: 1 — matrixSearchColWhere Sort:input value

[✓ Straight] Clockwise Counter-Clockwise

start: (0 0) — list 0 0

steps: 10 — 10

end: (150 250) — list ((Sort:sorted[NUMCOLS] - whichPosition) * 50) 250

Figure 8: A representative Animation form from the selection sort animation. Formulas impose the constraints so that this form can correctly animate any element in the unsorted group, no matter what its value, starting and ending positions, or direction of movement.

4: ISSUES RAISED BY DECLARATIVENESS AND RESPONSIVENESS

As the preceding examples show, Forms/3's declarativeness and responsiveness, which follow from its constraint-oriented evaluation model in a visual setting, are important from the standpoint of the small amount of actual programming needed to produce animations. In this section, we will explore other unusual features and issues that declarativeness and responsiveness bring to algorithm animation.

4.1: The Path/Transition Paradigm in a Declarative, Visual Setting

As mentioned earlier, the path/transition paradigm appealed to us as a starting point for devising and implementing animation in Forms/3 because it is a sound conceptual model with precise semantics. 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*.

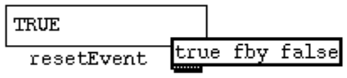
In Stasko's imperative textual implementation of the path/transition paradigm, a transition takes three parameters: a transition type, an image, and a path. The transition is then explicitly executed with a 'perform' command. In our approach, a transition also depends on a transition type, an image, and a path, but further depends on a resetEvent and a continueEvent. In effect, we replace Stasko's perform command with resetEvent and continueEvent. This is because in the responsive visual domain of Forms/3, the transition (represented by the Animation cell on the Animation form) is in principle continuously evaluated, meaning it does not need to be explicitly commanded to execute. Rather, it is told the constraints under which it may begin (because resetEvent's constraint formula must evaluate to true) and told the constraints under which it may advance (because continueEvent's constraint formula must evaluate to true). For example, Figure 9 shows the resetEvent cell from the wagon wheel program. Its formula of 'true followed by false' allows the transition to begin at the beginning of time (i.e., the first point in the time dimension), but not to be reset at any other time.

Stasko's path/transition paradigm includes a location abstract data type which, to maintain referential transparency, is not used in our paradigm. 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. If a Forms/3 object could report its absolute location, moving the cell that contains that object would produce different answers. This would violate referential transparency since in Forms/3, a cell's location on a form has no semantic meaning; moving the cell to a different location on a form may affect the program's appearance, but it does not change the program's semantics. To omit the use of absolute locations, in our paradigm the location of the path as well as the steps within the path are relative to the object's position in the program. In other words, each cell defines a local coordinate system for the objects within it, which are rendered on the screen in the context of the cell's position on the window.

4.2: Generalizing the Approach to Initiating and Advancing Animation

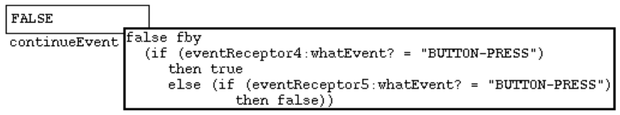
Non-declarative algorithm animation systems base the initiation of an animation sequence on 'interesting events.' These events are places in the algorithm program at which significant algorithm operations occur, and are identified by a call to initiate the desired animation sequence. Once an animation operation has been initiated, most of these systems advance them from frame to frame solely through the passage of time. Our declarative approach generalizes upon the traditional method in two ways: it expands the notion of what an interesting event is, and it allows this expanded notion of events to control not only the initiation of an animation sequence, but also its frame-by-frame advancement.

We have shown that in Forms/3, the equivalent of an 'interesting event' to activate an animation sequence is expressed as a constraining formula for the resetEvent cell on the Animation form. Since the detection of this event is expressed as a constraint, it is continuously evaluated. This provides more expressive power than imperative approaches because it allows events to be defined not only as the algorithm having arrived at a particular operation, but as *any* possible condition in the program. For example, in Forms/3, resetting the animation can depend on a cell having a particular value no matter when or how it occurred. This approach allows completion of an operation (signaled by some change in values) to be an interesting event as in



```
TRUE
resetEvent  true fby false
```

Figure 9: resetEvent cell from the wagon wheel program.



```
FALSE
continueEvent false fby
                (if (eventReceptor4:whatEvent? = "BUTTON-PRESS")
                   then true
                   else (if (eventReceptor5:whatEvent? = "BUTTON-PRESS")
                           then false))
```

Figure 10: continueEvent cell from the wagon wheel program. The formula specifies when the constraints change; lack of a final 'else' simply means that for any case not listed, there is no change in the constraints.

imperative systems, but it also allows interesting events to be defined as the satisfaction of other kinds of constraints, such as the presence of a certain combination of values *whenever* such a circumstance arises. Under this declarative approach, the programmer needs only to specify the condition, and does not need to worry about placing the constraint multiple places in the program to 'trap' all the possible places in the program that might cause the event.

In Forms/3, this generalized approach is also used in advancing an animation frame-by-frame. After all, the traditional time delay between animation frames is really just the satisfaction of a particular constraint—the passage of a time interval—which can be replaced by some other constraint in our system. The Forms/3 programmer defines this constraint, which can be time-based if desired, in the formula of the continueEvent cell on the Animation form.

As an example, Figure 10 shows the continueEvent cell from the wagon wheel program. The cell's formula constrains continuation to depend upon a mouse button press on the Start button and not the Stop button. (Another way to think about this is to say that 'continuing' the animation is a high-level event generated from a combination of low-level mouse events.) When the Start button is pressed, continueEvent's 'eventReceptor4:whichEvent?=BUTTON-PRESS' constraint is satisfied, allowing the animation to continue rendering with the next animation frame generated by the path. The constraint remains satisfied as the animation progresses through the frames until the Stop button is pressed, at which time continueEvent's (negative) constraint 'eventReceptor5:whichEvent?=BUTTON-PRESS' causes the continuation constraint to no longer be satisfied and the animation to halt.

4.3: On-The-Fly Exploratory Programming

Since Forms/3 is responsive (the programmer is given visual feedback immediately about the effects of a program change), algorithm animation programming in Forms/3 does not have the edit-compile-restart loop of traditional algorithm animation programming. Instead, a programmer can modify the constraints that govern an algorithm animation during its execution (a refinement characteristic of growing importance in scientific computations known as 'steering'), and will receive immediate feedback showing the effects of those modifications. For example, suppose that something affecting the animation such as an input to 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 declarativeness and responsiveness of Forms/3

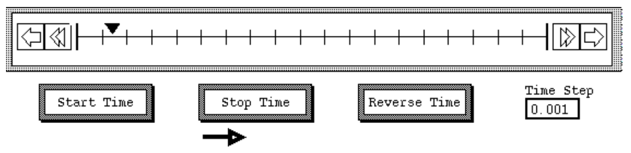


Figure 11: The time-related buttons in the Forms/3 environment allow the programmer to control the direction of execution. The programmer may also manipulate the slider bar or arrows to travel flexibly forward or backward through time.

means that the complete history of the new animation resulting from this change will be inherently redefined, and the screen display can be automatically updated to reflect the same point in time that the animation was at before the modification was made.

4.4: Traveling through Time

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. In fact, the programmer can travel to any moment in time desired using the time slider bar and, when ready, resume execution in a forward or backward direction—one step at a time or at normal speed—as many times as desired. See Figure 11.

While some algorithm animation systems allow the *animation* to run in both directions, it is usually impossible for the *algorithm* to execute in reverse. A few visual debugging environments do feature 'video recordings' of execution that can be traversed in either direction, but these recordings do not adapt to changes in the program. In Forms/3, the animation and algorithm can actually execute synchronously in either direction, even when programming changes are entered, because the cell's changed formula is a complete specification of its history of values.[1]

4.5: Independence of the Algorithm From Its 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, and our system is not the first. We mention it because this feature is considered important in algorithm animation research, and the fact that it is inherently available in declarative approaches has not been brought out before. 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 compiled[2] program must be kept up-to-date—one with the unaltered algorithm and one with the animation-annotated algorithm. In contrast to this, a declarative mapping approach allows the same algorithm program to be used both with and without its animation without the need for separate compiled (or source) versions of the program.

5: CURRENT STATUS AND FUTURE WORK

The algorithm animation features described in this paper have been implemented in our research prototype, which is implemented for Unix workstations in Lisp and Garnet, a one-way constraint system that handles the user interface [Myers et al. 1990]. A more detailed description of the approach and of an earlier implementation can be found in [Carlson and Burnett 1995]. However, this work continues to evolve, and we have several improvements planned.

One improvement we are considering is a more flexible way to specify paths so that the path of one object may depend on the paths of others. Such a change would be a fairly minor matter of changing the user interface regarding how the path specifications are entered. We are also currently working on several algorithm animation examples, especially tree-based algorithms, to experiment more with the power of the approach. Although the approach is not computationally limited by the use of one-way constraints (our approach is computationally as powerful as standard imperative languages), it is limited in terms of expressive power by the need for further development of Forms/3's explicit time dimension. Ideally, an entire animation transition should occur within one logical time unit from the algorithm's perspective. In the current version, it is necessary to simulate this when an animation is nested inside an algorithm step by inserting 'slow down' constraints into the algorithm. In the future, we plan to expand the automatic synchronization to handle these cases by adding new temporal constraints that instead speed up the animations by breaking the atomic time units into smaller nested units.

6: CONCLUSION

By seamlessly integrating algorithm animation into Forms/3, we have shown that algorithm animation need not compromise the characteristics of a declarative VPL with a constraint-based evaluation model, nor need it add complexity to the language. Rather, we have exploited such a language's

[1]The system tracks the placement of values in time using an implementation technique we term "lazy marking", which is described in detail in [Burnett and Atwood 1994].

[2]In fact, two versions of the *source* code are also needed if compile-time directives and all the animation annotations detract too much from the readability of the 'production' version of the source code.

declarative and responsive characteristics to produce the following unusual features of algorithm animation:

- a more general way to initiate and advance animations;
- on-the-fly exploratory algorithm animation programming (steering);
- the ability to change the direction of execution and flexibly travel through time, in both the animation and the algorithm synchronously;
- removal of the animation calls from the algorithm code; and
- a unified, visual setting for programming and experimenting with *both* the algorithm and its animation.

These features show ways in which algorithm animation, when integrated with a constraint-based visual setting, can surpass the capabilities of algorithm animation for imperative, textual languages. Key to these results is integration with the same constraint-based VPL used for the algorithm programming. These advances form a natural and powerful mechanism by which certain VPLs—those that are responsive and that follow a constraint-based evaluation model—can provide highly interactive support for combined visual programming of algorithms and visual programming of their animations.

ACKNOWLEDGMENTS

Sherry Yang's work on generalization greatly simplified the way programmers specify algorithm animations. We would also like to thank John Atwood, Herkimer Gottfried, Rebecca Walpole, and the rest of the Oregon State University Visual Programming Language Research Group for their contributions to this research.

REFERENCES

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

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

[Burnett and Atwood 1994] M. Burnett and J. Atwood, "Lazy Marking: A Lazier Implementation of Functional I/O for Graphical User Interfaces", TR 94-60-9, Oregon State University, Computer Science Department, Dec. 1994.

[Carlson and Burnett 1995] P. Carlson and M. Burnett, "Algorithm Animation in a Declarative Visual Programming Language," TR 95-60-2, Oregon State University, Computer Science Department, Apr. 1995.

[Duisberg 1987] R. A. Duisberg, "Visual Programming of Program Visualizations: A Gestural Interface for Animating Algorithms," *1987 Workshop on Visual Languages*, Linkoping, Sweden, 55-66, Aug. 1987.

[Duisberg 1987/88] R. A. Duisberg, "Animation Using Temporal Constraints: An Overview of the ANIMUS System," *Human-Computer Interaction*, Volume 3, Number 3, 275-307, 1987/1988.

[Hansen 1994] W. J. Hansen, "The 1994 Visual Languages Comparison," *1994 IEEE Symposium on Visual Languages*, St. Louis, MO, 90-97, Oct. 4-7, 1994.

[Helttula et al. 1989] E. Helttula, A.Hyrskykari, and K. Raiha, "Graphical Specifications of Algorithm Animations with ALADDIN," *22nd Hawaii International Conference on System Sciences*, Kailua-Kona, HI, 892-901, Jan. 1989.

[Lieberman and Fry 1995] H. Lieberman and C. Fry, "Bridging the Gap Between Code and Behavior in Programming," *ACM Conference on Computers and Human Interface (CHI'95)*, Denver, Colorado, Apr. 1995.

[Moher 1988] T. G. Moher, "PROVIDE: A Process Visualization and Debugging Environment", *IEEE Transactions on Software Engineering*, Volume 14, Number 6, 849-857, June 1988.

[Mukherjea and Stasko 1993] S. Mukherjea and J. T. Stasko, "Applying Algorithm Animation Techniques for Program Tracing, Debugging, and Understanding," *15th International Conference on Software Engineering*, Baltimore, MD, 456-465, May 17-21, 1993.

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

[Roman et al. 1992] 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, 161-193, June 1992.

[Stasko 1990a] J. T. Stasko, "Simplifying Algorithm Animation with TANGO," *1990 IEEE Workshop on Visual Languages*, Skokie, IL, 1-6, Oct. 4-6, 1990.

[Stasko 1990b] J. 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, 213-236, Sept. 1990.

[Stasko 1991] J. T. Stasko, "Using Direct Manipulation to Build Algorithm Animations By Demonstration," *CHI'91 Conf. Proceedings*, New Orleans, LA, 307-314, April 27-May 2, 1991.

[Takahashi et al. 1994] S. Takahashi, K. Miyashita, S. Matsuoka, and A. Yonezawa, "A Framework for Constructing Animations via Declarative Mapping Rules," *1994 IEEE Symposium on Visual Languages*, St. Louis, MO, 314-322, Oct. 4-7, 1994.

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