IEEE Symposium on Visual Languages, Seattle, Washington, Sept. 2000, pp. 137-144

Time in Grid-Oriented VPLs: Just Another Dimension?

Margaret Burnett, Nanyu Cao, and John Atwood
Department of Computer Science
Oregon State University, Corvallis, Oregon 97331
{burnett, caona, atwoodj}@cs.orst.edu

Abstract

Specifying varying speeds and temporal relationships is necessary when programming graphical animations, but support for temporal programming has usually been done by adding new language features to a VPL, and these features must be mastered over and above the other aspects of the VPL. However, some researchers have believed that time should be able to be treated like just another dimension. In this paper, we explore whether temporal programming can indeed be done using exactly *the same devices as in spatial programming in grid-oriented VPLs. Toward this end, we provide a continuum of models aimed at this goal, with their advantages and disadvantages. Also, we identify core issues that help illuminate the essence of the problem.*

1. Introduction

Many programming languages do not provide explicit control to the programmer over the temporal aspects of a program. However, there are important applications in which temporal behavior is critical to a program's correctness, especially in applications involving visual data. One example is animated graphics, a popular domain for visual programming languages (VPLs).

In earlier work in Forms/3 [2-4], we showed how adding a time dimension to the spreadsheet paradigm facilitates programming of animated graphics. Our interest was in the realm of software animation, and we showed a straightforward way to program an animation such as the one in Figure 1 [4]. However, another goal of algorithm animation is to program such an animation without modifying the original (sorting) algorithm. Although we had initially believed taking this step would be straightforward, we later discovered this not to be the case. In particular, as Figure 2 demonstrates, two problems arose in programming the temporal relationships between the sort and its animation:

Problem 1: How to program temporal behavior in a notation closer to how they evolve over time than the one-dimensional syntax of `earlier` and `fby`.

Problem 2: How to program temporal interrelationships without artificially inserting code such as `counter`, whose purpose is to slow down the algorithm.

In this paper, we explore whether visually programming temporal behavior can be done via exactly the same techniques used in visually programming spatially. We will focus on ways to programmatically control speed and temporal interrelationships among elements of the program, such as are needed in programming animated graphics, but will ignore the issue of programming temporal deadlines. (The latter is important in some domains, but is not required for most animated graphics programming.)

We will consider this issue in grid-oriented VPLs. These include members of the spreadsheet paradigm, as well as other grid-oriented VPLs. The issue is particularly relevant to VPLs of this group supporting grid-based programming of graphics, such as AgentSheets [14], Cocoa [12], Forms/3, NoPump [25], Spreadsheet for Information Visualization [5], and Spreadsheet for Images [16].

The contributions of this paper are two. First, several models that address variants of this issue are presented. Second, differences between temporal and spatial programming are identified that can cause difficulties in supporting grid-based programming of temporal behavior, because

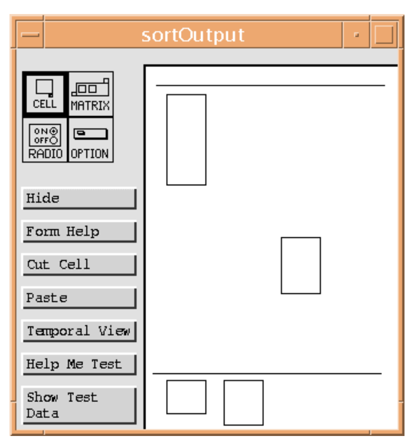


Figure 1: A moving box drops down gradually (10 steps) into place when its corresponding element is inserted into a sorted group.

they can lead to language design problems that are known to interfere with humans' programming abilities.

2. Background and related work

2.1 Cognitive Dimensions

It is possible to bring research into cognitive issues of programming to bear upon VPL design decisions by considering Green's and Petre's *Cognitive Dimensions*, a distillation of psychology of programming knowledge into a form usable by non-psychologists [11]. This work has motivated some of our design goals. For example, solving Problem 1 would improve *closeness of mapping*, the similarity of the programming notation to the problem being solved.

Consistency is another of the Cognitive Dimensions particularly pertinent to this work: "When some of the language has been learnt, how much of the rest can be inferred?" [11]. Since (spatial) grid-based programming has been shown to be easy enough for end users to master, a level of consistency allowing those users to reapply the same techniques to time suggests that users would be able to master temporal programming as well, and this is what motivated our interest in this strategy. In this paper, we will use the Cognitive Dimensions as an early evaluation mechanism for design alternatives, primarily to raise warnings when a design decision is potentially at odds with research from cognitive principles of programming.

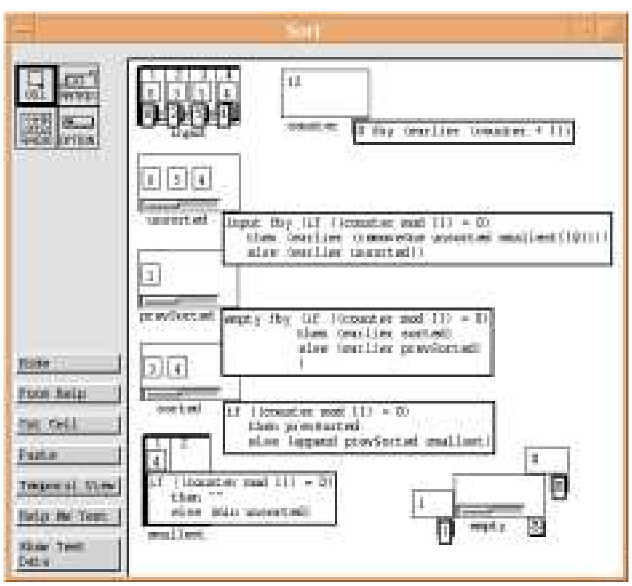


Figure 2: This selection sort, written using a previous version of Forms/3, required cell `counter` to slow down the algorithm so that the accompanying animation of Figure 1 had time to go through 10 graphical movements after each element's insertion.

2.2 VPLs and temporal behavior

Although there are several VPLs that support *visualization* of temporal behavior (e.g., [7,10]), many VPLs have not explicitly supported *programming* of temporal behavior.

Some early VL work regarding temporal aspects of software was done to support temporal logic in program specifications. The earliest such VL approach is probably GIL (Graphical Interval Logic) [15]. GIL is a visual temporal logic for reasoning about the temporal properties of programs. It is a high-level, proof-oriented approach aimed primarily at visual specification and subsequent comparison to a (separate) program's properties. While there is also a tool supporting generation of a model from GIL temporal specifications, it does not support programming other than of temporal specifications.

There have been numerous examples of VPL research about coding temporal information via devices specific to temporal programming. Duisberg's work with temporal constraints [8] and with use of music notation devices [9] are perhaps the earliest VPL work aimed at temporal characteristics of animated graphics. Later examples include programmatic use of diagrams oriented toward time, such as space-time diagrams and concurrency maps as in [21] and time lines as in [13].

Treating time as temporal events has been another approach. For example, in Pavlov [26], there is an integer global time. Programmers explicitly specify what will happen at time n or every n times. However, it is not possible to refer to other objects' previous states. Pavlov's temporal programming is unlike space programming because unlike the other aspects of Pavlov, temporal information is not inferred from demonstrations.

There has also been significant work on temporal aspects of programming multimedia applications. Like the work described above, this body of work also provides constructs specific to temporal programming. One of the earliest examples of such work in VPLs was TYRO [17], a demonstrational, constraint-oriented multimedia authoring language. TYRO is intended as a "designer's apprentice," and infers the correct temporal layouts from a combination of constraints and examples programmed using temporal-specific mechanisms. More recently, Song et al. developed an elastic time model, in which the deadline constraints are specified as the endpoints of a "spring", and the system uses these as constraints to schedule the computations that need to fit in this spring [20]. Their time-box representation provides a visual way of specifying these spring endpoints and their relationships to other springs.

Approaches such as the above support temporal programming by introducing specific models and subsystems that relate only to time. The other possibility is that time is just another axis in a coordinate system, i.e., a t-axis in a coordinate system already containing an x-, y-, and perhaps a z-axis. This approach was used in an early spreadsheet language based on Lucid [23] known as Plane Lucid [6], which gave 2-D operators such as hsby and vsby for "horizontally/vertically spatially by (to the right/below)" to reference elements along the x- and y-axis of a spreadsheet grid and analogous operators such as `fby` meaning "followed by (temporally next)" to reference elements along the t-axis. The earlier Forms/3 approach discussed in the introduction was influenced by this work.

Gamut [18] is a programming-by-demonstration VPL that also supports some temporal programming without introducing new models. The form of temporal programming it supports is referencing the previous state of an object. This is done by inclusion of faded "temporal ghosts" in the demonstration area, so that they can be included in the demonstrated logic. The approach corresponds to our design goal of allowing a space-oriented programming technique to be applied to time, but does not allow time-oriented logic such as relative speed to be included, and allows reference only to the just-previous state, not to any other moment in time.

To summarize, previous VPL research that relates to temporal behavior can be divided into two categories. One is to provide entirely new language devices designed explicitly to support for programming in time. The other is that time is just another dimension, able to be handled exactly like other dimensions. Our work falls in the latter category.

3. Grid-based space and time

"Programming" (defining formulas for) spreadsheet grids has been mastered by thousands of end users. This demonstrated ease of use motivates us to reapply this grid-based way of programming to time.

Here is how grid-based programming works in space. In the spreadsheet system Excel, the user puts a formula in one cell, and then replicates it (copies with the references automatically changed) across as much of the grid as desired. In the spreadsheet system Lotus, the user instead can group several cells in a grid and instruct the system that these cells all share the same formula. Forms/3 supports the latter strategy. The cells' formulas can refer to other cells by row and column number indices.

3.1 The animated sort example

Software visualization is an important application of animated graphics, and thus we have chosen the animated sort as a motivating example for this paper.

Since Forms/3 follows the spreadsheet paradigm, the group of elements being sorted is represented by a spreadsheet grid (also called a matrix in some papers). In Figure 2 from the introduction, there are 6 grids plus 1 standalone cell (`counter`). The overall logic is that `unsorted` starts as `input`, ("`input fby`...", where `fby` means "followed by"), and then is either the result of removing the minimum element from its own earlier value or else is just a reference to its own earlier value. `Sorted`'s logic is the mirror image of `unsorted`'s: it is either its own earlier value or else is the result of appending the minimum `unsorted` element to its own earlier value.

A user can subdivide a Forms/3 grid into regions by dragging a thick border line at the left of a grid rightwards or from the top downwards. All cells in a region share the same cell-level formula. (This is a simplified version of an approach pioneered by Viehstaedt and Ambler [22] and extended in Formulate [1,24].) Grid `input` was subdivided in this way, one cell per region so that each cell can have its own formula. See also Figure 3 for more examples of regions. Alternatively, a grid can have a formula that defines the entire grid as a whole, instead of the cells within it, as in `unsorted`, `prevSorted`, and `sorted`. (Grid `prevSorted` is present to break up what would otherwise have been a long, nested formula for grid `sorted`.)

Forms/3 grids have been empirically shown to be simpler for humans to use for some tasks than are more traditional methods [19], but they do not seem particularly simple in Figure 2: much logic in this program is required for "slowing" down the program via cell `counter` so that the accompanying animation can perform smooth graphical transitions between element insertions into grid `sorted`. To accomplish this slow-down, most formulas contain `if`s that make decisions depending upon the status of `counter`. This is the kind of workaround necessary in languages that do not directly support the ability to program temporal behaviors.

What we would like to do instead is to allow the specification of this temporal constraint without the use of such workarounds. Toward this end, we next consider several models of time to directly support temporal programming.

3.2 A continuum of time models

Several grid-based models of time are presented here in increasing order of power (functionality) but also of expected programming expertise. Given some model's placement along this continuum, we shall evaluate it by the number of problematic VPL design issues it introduces. The issues we consider are cognitive issues raised by Green/Petre's Cognitive Dimensions, and programming language design problems such as limitations and ambiguities. The models are:

0. Base Model: All grid cells have the same speed.
1. SNF Model: Slow, Normal and Fast. A user can choose one of a fixed number of speeds for a cell (e.g., `cellA` goes Fast). In our instantiation, the fixed number is 3: Slow, Normal, and Fast.
2. k*N Model: A user can specify a cell's speed as being an arbitrary integer constant k times faster/slower than Normal (e.g., `cellA` goes 2 times faster than Normal).
3. k*x Model: A user can specify a cell's speed as an arbitrary integer constant k times faster/slower than the speed of another cell x (e.g., `cellA` goes 2 times faster than `cellB`).

4. Programming along the t-axis

4.1 The base model

Figure 3 shows a (spatial) grid-based approach to programming the Fibonacci sequence, using the grid-oriented spatial relationships among the columns. The user separated this grid into the 3 regions shown by dragging the thick border line from the left of the grid rightwards. In Figure 4, the same technique was used to program a Fibonacci sequence over time instead of space, i.e., dragging the region separator border along the t-axis positions instead of the x-axis positions. As with grid-based programming, a formula can reference another cell's value using indices along the axis. In grid-based programming in Forms/3, x- and y-axis referencing is done via format `name[row@col]`; hence temporal referencing is done via format `name<time>`. For example, in Figure 4, `fib<t-1>` references `fib`'s value at the temporal position just before (to the left on the t-axis).

In principle, this is the same notion as that pioneered by Plane Lucid [6], but the programming devices of our approach feature greater closeness of mapping to the domain of temporal programming. The relationships between formulas, ensuing values, and the t-axis are explicit in the way the user enters the formula (by directly moving borders among the values on the t-axis and clicking on other values to reference them) and in the fact that both the formula's temporal aspects and values are explicit in their placement along the time line.

Because of the similarity to the spatial grid-based approach, users can reapply what they already know about programming in space to programming in time. In fact, the *only* thing this model does is add a new axis to the existing grid-based coordinate system. Every cell has exactly the same "logical" speed, i.e. is indexed along the t-axis in the same way as would be done on an x- or y-axis. Because every cell has the same speed, this model does not have the functionality to solve Problem 2, but it provides a basis upon which the later models can build.

Still, even this simple model brings out an intriguing issue about the difference between programming time and space. In space, it is usual for any grid item to be able to reference any other grid item, provided that no cycles are induced. But in time, the ability for the "past" to reference the "future" runs counter to the real world. We believe this flexibility could actually be used to advantage by an imaginative programmer, and would not be encountered by others. However, this is only our own opinion, and empirical work would be needed to solidly determine the extent to which it actually caused problems for users.

4.2 Model SNF: Slow, Normal, and Fast

The base model does not solve Problem 2, because all the cells have the same temporal speed. The goal of Model SNF is to solve Problem 2, for at least some cases, in a way usable by users who know only grid-based programming.

In keeping with this goal, Model SNF adds to the base model by simply allowing the user to specify one of k speeds for a cell, where k is fixed in the language implementation. In our prototype, k=3, which allows speeds of Slow, Normal, and Fast. Also fixed in the language implementation are the relationships between Slow, Normal, and Fast. (We are considering allowing the user

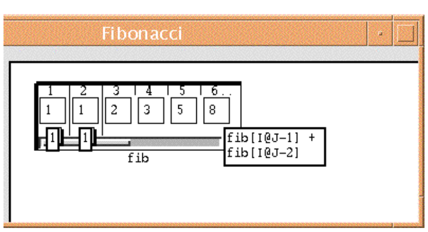
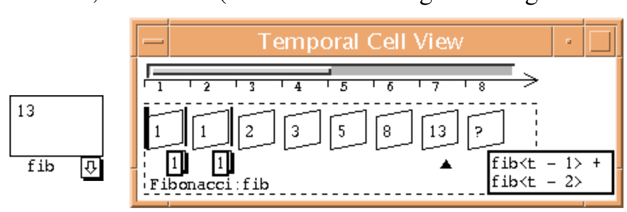


Figure 3: This Forms/3 spreadsheet contains a (spatial) grid of Fibonacci numbers. The grid contains 3 regions, and the cell values in each region are defined by the formula for that region.



(a) (b)

Figure 4: (a) This single Forms/3 cell can compute the Fibonacci sequence temporally under the base model; the computations are specified over time rather than space. (b) The temporal view window of the same cell shows its 3 regions over time. Note the parallels to the spatial approach of Figure 3.

to modify this via a system preferences dialog.) In this discussion, we will set the speed difference factor to 2. Using this model, in the sorting example, the sorting algorithm's cells could have Normal speed (the default) and the user could set the speed of the animation cells to Fast, using the mechanism in Figure 5.

Allowing these multiple rates of speed raises the following question: what do temporal references such as Figure 4's <t-1> mean under this model? This question can be answered via the following three semantic rules:

<u>*Rule 1. The t-axis:*</u> As in the base model, in Model SNF there is one and only one time axis, it is known globally, and its indices are 1,2,… (These characteristics are the traditional characteristics of x- and y-axes.) We term each position along the t-axis a *tick*.

Ticks are not tied to "real" time units such as seconds or minutes; they are simply progressive positions on the t-axis. Speed Fast allows values at every tick, provided that dependencies in the formulas deliver enough data to populate every tick, speed Normal allows values at every 2 ticks, (1,3,...), and speed Slow allows values at every 4 ticks (1,5,...). See Figure 6.

In this and all the models in this paper, if there is a need to compute with or display a value at a time not explicitly populated for that cell, the most recent value before that time is still considered to be current. For example, if a slow cell's value at time 3 is needed, its value at time 1 is considered still current and is used. A more conventional programming language view might be that this value should be considered to be undefined at times 2-4, but that is not the way time works in the real world. In the real world, slower moving objects don't have "undefined moments"; each intermediate state just lasts longer. This is visually depicted by the width of each value of slow cells, as in Figure 6.

<u>*Rule 2. What t means:*</u> For time indices in formulas (e.g., `B<t-2>`), t is an index on the global t-axis.

<u>*Rule 3. What constants mean:*</u> For time indices in formulas (e.g., `B<t-2>`), constants follow the units of the global t-axis.

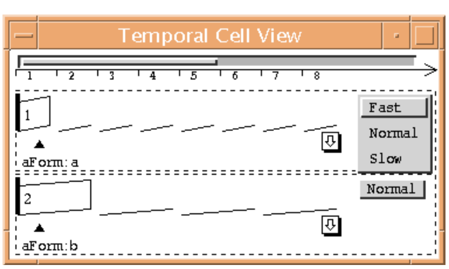


Figure 5: In Model SNF, a user specifies each cell's speed, supported here with a pop-up menu.

Intuitively, t means "now". For example, if `C` refers to `B<t-2>`, then `C`'s value at time 5 will be the same as `B`'s value at time 3, as in Figure 6.

Rule 3 may seem puzzling: why is it necessary in a programming language to say what a constant means? The answer is that the multiple speeds introduce multiple unit possibilities. We will return to these possibilities later. An advantage to Rule 3 as stated here is that it explicitly relates to the t-axis indices visible in the VPL.

Under this set of rules, Model SNF adds the functionality needed to solve Problem 2, provided that only a fixed number of speeds are needed. (The noticeable improvement as a result will later be illustrated by Figure 7.) In this respect, Model SNF takes an important step forward from the base model.

However, the fixed number of speeds, which at first glance may seem only mildly limiting, is quite problematic from the standpoint of two of the Cognitive Dimensions, premature commitment and viscosity. *Premature commitment* means users must make programming decisions before they have enough information to do so. In Model SNF, users must decide upon the speed of cells (or accept the default decision of Normal) even if they do not yet know if they will eventually want to add faster or slower cells related to it. Further, changing their minds later about a single cell's speed requires changing the speeds of, in the worst case, every cell in the program, in order to retain the desired temporal interrelationships. Thus *viscosity*, or resistance to change, is also a problem. Fortunately, removing the limitation on number of speeds can help with these problems.

4.3 Model k*N: k times faster/slower than normal

Model k*N extends Model SNF such that any cell's speed can be specified as an arbitrary constant times faster or slower than Normal; hence it removes the limitation from Model SNF.

Unfortunately, this straightforward extension leads to a

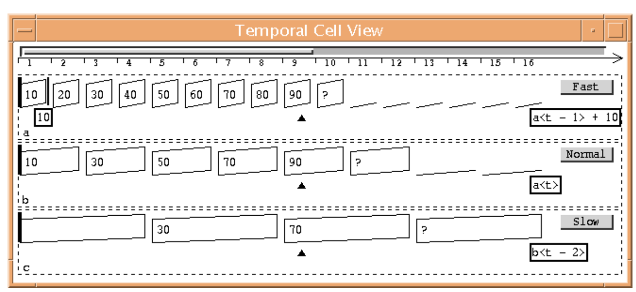


Figure 6: In Model SNF, cell values are located along a global t-axis as shown. The triangle represents the current time, and ?s mark the next values to be computed.

new problem: what to do about the integer numbering specified in Rule 1. If Normal means 1 value every c ticks, then whenever the user selects a k > c, non-integer indices arise. For example, if Normal means 1 value per tick and a cell `A` were specified to be 3 times faster than Normal, it would need 3 values per tick, leading to t-axis numbering like (1, 1.333, 1.666, 2, 2.333, ...). Here the *consistency* Cognitive Dimension resurfaces as an issue, since this numbering system is clearly not consistent with the usual numbering of x- and y-axes.

One potential solution might seem to be to hide the unconventional t-axis numbering. However, hiding the indices would prevent users from knowing how to reference a previous element (e.g., <t-0.333>), so hiding the indices does not solve the problem.

A second potential solution might be to eliminate "faster than" as an option, allowing `B` for example only two speed choices: Normal or "k * slower than Normal". This retains integer numbering, but unfortunately also retains the premature commitment and viscosity problems of Model SNF, since now a commitment is needed that, if cell `A` is at Normal speed, no other cell will ever need to go faster than `A`.

A third potential solution is to retain integer numbering by automatically renumbering the t-axis indices when necessary. The problem here is that all the formulas relying upon a previous numbering would also have to be automatically changed. For example, consider cell `C`'s formula of `B<t-2>` in Figure 6. If the user then added a new cell `X` 4 times faster than Normal—not referencing or being referenced by `A`, `B`, or `C`—she would probably be surprised if the system automatically changed `C`'s formula to `B<t-8>`. Yet, making this change is necessary: if the system did not make this change, `C`'s formula would produce different answers than it had before. This is a classic case of *hidden dependencies*, the Cognitive Dimension that identifies dependencies that are not explicit in the program.

4.4 Model k*N revised via Rule 3

It is possible to leave Rule 1 unchanged, thereby avoiding these difficulties, by instead changing Rule 3, what constants mean. Until now, constants have been in t-axis ticks; e.g., `<2>` meant the 2nd tick, and `<t-2>` meant 2 ticks before t. However, another look at Figure 6 shows that there are multiple granularities along this axis. This presence of multiple granularities is significantly different from the way values populate x- and y- axes of grid-like systems. For example, in the figure, `C`'s reference to `B<t-2>` at position 9 on the t-axis is `B`'s value 2 *ticks* back, rather than `B`'s value 2 *values* back. Changing Rule 3 allows the user to work directly in these granularities:

<u>Rule 3 (revised). What constants mean:</u> For time indices in formulas (e.g., `B<t-2>`), constants are mapped onto the units of the global t-axis by multiplying by the granularity of the *referenced* cell.

For example, if `B`'s granularity is 2 times slower than the t-axis, populating the t-axis at t=1,3,5... as in Figure 6, then `C`'s reference to `B<t-2>` would, instead of as in the figure, refer to `B` at t=1-(2*2), 5-(2*2), 9-(2*2), … (The first t-index in this sequence would be negative, resulting in no value for `C` until t=5.)

With this revision the t-axis can be renumbered whenever doing so is necessary to retain integer numbering of the t-axis. The renumbering will not require any formulas to be changed, and the user will not have to perform granularity conversions manually.

There are still disadvantages, but they are not nearly as troublesome as the problems encountered prior to this revision. One disadvantage is that this variant of Rule 3 is less explicitly tied to the visible t-axis numbering. The loss of explicitness is because the subtraction of 2 from t in `<t-2>` is actually in different units than before: number of values rather than number of ticks. (An alternative would be to remove all ties to the t-axis, which we briefly discuss in Section 5.) Premature commitment and viscosity are also still issues in this model, although to a lesser extent than in Model SNF, because if two cells' speeds exist at adjacent integer factors (e.g., 2 times faster than Normal and 3 times faster than Normal), it is not possible to insert a new speed between those two.

This model has a compelling advantage: it avoids or reduces many of the difficulties of previous potential solutions and solves both Problems 1 and 2. Figure 7 demonstrates this with the new version of the sort[1]. Compare this figure to Figure 2: all the former `if`'s devoted to slowing down the timing are gone, and now only 3 formulas, all of which are now non-nested, remain in the sorting program: one to select the `smallest` unsorted element, one to remove it from the `unsorted` grid, and one to append it to the `sorted` grid. These grids' speeds are specified as Normal, and the animation cells' speeds are all specified as 10 times faster than Normal.

4.5 Model k*x: k times faster/slower than x

Model k*x is such a straightforward extension of Model k*N, few words are required to describe it; yet, it brings highly useful properties. The only difference from Model k*N is that, instead of specifying a cell's speed to

---

[1] With different time indexing (e.g., `t-10` instead of `t-1`), this solution is also possible in Model SNF, but only if the speed differential between speeds were permanently set to 10 in the language implementation.

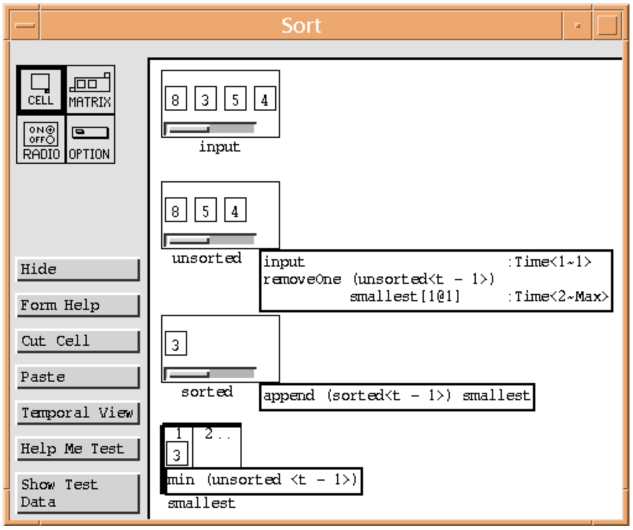


Figure 7: The selection sort under Model k*N. Compare this figure to Figure 2. See Figure 8 for a temporal view.

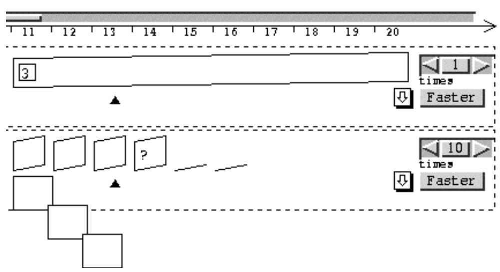


Figure 8: Temporal view of the sorted grid (top row) and the cell containing the rectangular box value moving downward through different positions 10 times faster than normal (bottom row).

be k times faster than Normal as in Model k*N, the user specifies the cell's speed to be k times faster than some other cell x. Thus, the spreadsheet of Figure 7 is exactly the same as that needed under this model. The temporal view of Figure 8 would be almost the same under this model, with the small change that, in the speed setting for the animation cell, the user would specify that it was 10 times faster *than* `sorted`.

Model k*x features better closeness of mapping than the previous model, because the user can specify the temporal relationships among cells directly, instead of translating them all into temporal relationships with Normal speed. Further, this model under the same rules as in Section 4.4 removes the problems of premature commitment and viscosity while retaining the advantages of solving Problems 1 and 2.

5. Future work

Note that the models presented are all static in the sense that the cell's temporal speed can be statically determined, without the need for run-time data. We are also considering experimenting with an inherently dynamic model, a y*x model, in which a user can specify a cell's speed as an arbitrary computation y faster/slower than the speed of another cell x (e.g., `cellA` goes `cellB` times faster than `cellC`).

A more fundamental addition we are working on is to bring some of the elements of computer graphics to bear on temporal programming, such as by employing local coordinate systems and operations such as "scale". However, these changes rest upon a change in Rule 2 (what t means) to a local notion of t, and this in essence completely separates the global t-axis from any use of t in a formula. An implication of this separation is that it makes the numbering of the t-axis useless, which shows how far removed this approach is from grid-oriented x- and y-axis programming. Pursuing this approach could yield an approach in which temporal programming can be done as a simple extension to spatial programming, but the model of spatial programming it extends is graphics programming (normally done by professional programmers) as opposed to grid-based programming (often done by end users).

6. Conclusion

Although several prior researchers have made the point that temporal programming should be achievable using the same mechanisms as spatial programming, this possibility has not been thoroughly investigated. In this paper, we have added to what is known about this possibility for grid-based VPLs by attempting to solve two specific problems in this way. The motivation is this: end users succeed at grid-based programming in space, which suggests that an approach consistent with this for programming temporal behaviors would allow this audience to reapply what they already know to temporal programming applications such as animated graphics.

The models we have developed reveal several core differences between the ways grid-based programming have been supported in space and the extension of such ways to temporal programming:

- The ability to access *any* position on the x- or y-axis loses its consistency with the real world on the t-axis: what does it mean for a past position to reference the future?
- In the real world, in space, if a value is not present at some specific position then it is missing, whereas in time, a situation at time t persists until something different occurs. Upholding these real-world conventions in a VPL seems necessary and useful but introduces inconsistency between space and time.
- Whereas spatial grid element sizes are constant in the

grids found in the VPLs of which we are aware, multiple granularities on the t-axis are needed to support varying speeds.

These differences did not lead to significant problems in the first model (Model Base), but because that model does not accommodate the third difference, it does not solve Problem 2. Accommodating the differences, especially the third one, in the next two models introduced several problems such as premature commitment, viscosity, lack of consistency, and hidden dependencies, all of which have been shown to cause difficulties for humans. However, the last model (Model k*x) overcomes these problems to a large degree, and supports direct, grid-based specification of an unlimited number of speeds and temporal relationships, solving both Problems 1 and 2. This model thus provides a step forward in functionality for grid-based languages.

Acknowledgments

We thank the members of the Visual Programming Research Group at Oregon State University for their feedback and help with the implementation. This work was supported in part by NSF under CCR-9806821 and NYI Award CCR-9457473.

References

[1] A. Ambler and A. Broman, Formulate solution to the Visual Programming Challenge, *J. Visual Lang. Computing* 9(2), 171-209, Apr. 1998.

[2] J. Atwood, M. Burnett, R. Walpole, E. Wilcox, and S. Yang, Steering programs via time travel, *IEEE Symp. Visual Lang.*, Boulder, CO, 4-11, Sept. 1996.

[3] M. Burnett, J. Atwood, Z. Welch, Implementing level-4 liveness in declarative visual programming languages, *IEEE Symp. Visual Lang.*, Halifax, Canada, Sept. 1998.

[4] P. Carlson, M. Burnett, and J. Cadiz, A seamless integration of algorithm animation into a visual programming language, *ACM Proc. Workshop Advanced Visual Interfaces*, Gubbio, Italy, 194-202, May 1996.

[5] E. Chi, J. Riedl, P. Barry, and J. Konstan, Principles for information visualization spreadsheets, *IEEE Computer Graphics and Applications*, July/Aug. 1998.

[6] W. Du and W. Wadge, A 3d spreadsheet based on intensional logic, *IEEE Software*, 78-89, May 1990.

[7] M. Duecker, C. Geiger, R. Hunstock, G. Lehrenfeld, W. Mueller, Visual-textual prototyping of 4d scenes, *IEEE Symp. Visual Lang.*, Capri, Italy, 328-335, Sept. 1997.

[8] R. Duisberg, Animated graphical interfaces using temporal constraints, *ACM Conf. Human Factors in Computer Systems (CHI'86)*, Boston, MA, 131-136, Apr. 1986.

[9] R. Duisberg, Visual programming of program visualizations: a gestural interface for animating algorithms, in *Visual Languages and Applications* (T. Ichikawa, E. Jungert, R. Korfhage, eds.), Plenum Publishing, NY, 1990.

[10] E. Freeman, D. Gelernter, and S. Jagannathan, Uniformity of environment and computation in MAP, *IEEE Symp. Visual Lang.*, Boulder, CO, 130-137, Sept. 1996.

[11] T. Green, M. Petre, Usability analysis of visual programming environments: a 'cognitive dimensions' framework, *J. Visual Lang. Computing*, 131-174, June 1996.

[12] N. Heger, A. Cypher, and D. Smith, Cocoa at the Visual Programming Challenge 1997, *J. Visual Lang. Computing* 9(2), 151-169, Apr. 1998.

[13] S. Hibino and E. Rundersteiner, User interface evaluation of a direct manipulation temporal visual query language, *ACM Int'l. Multimedia Conf.*, Seattle, WA, 99-107, Nov. 1997.

[14] A. Ioannidou and A. Repenning, End-user programmable simulations, *Dr. Dobb's Journal*, 40-48, Aug. 1999.

[15] G. Kutty, L. Dillon, L. Moser, P. Melliar-Smith, and Y. Ramakrishna, Visual tools for temporal reasoning, *IEEE Symp. Vis. Lang.*, Bergen, Norway, 152-159, Aug 1993.

[16] M. Levoy, Spreadsheet for images, *ACM Siggraph 94*, 139-146, 1994.

[17] R. MacNeil, Generating multimedia presentations automatically using TYRO, the constraint, case-based designer's apprentice, *IEEE Workshop Visual Lang.*, Kobe, Japan, 74-79, Oct. 1991.

[18] R. McDaniel and B. Myers, Getting more out of programming-by-demonstration, *ACM Conf. Human Factors in Computing Systems*, Pittsburgh, 442-449, May 1999.

[19] R. Pandey and M. Burnett, Is it easier to write matrix manipulation programs visually or textually? An empirical study, *IEEE Symp. Visual Lang.*, Bergen, Norway, 344-351, Aug. 1993.

[20] J. Song, M. Kim, and G. Ramalingam, Interactive authoring of multimedia documents, *IEEE Symp. Visual Lang.*, Boulder, CO, Sept. 1996, 276-283.

[21] N. Stankovic and K. Zhang, Towards visual development of message-passing programs, *IEEE Symp. Visual Lang.*, Capri, Italy, 144-151, Sept. 1997.

[22] G. Viehstaedt and A. Ambler, Visual representation and manipulation of matrices, *J. Visual Lang. Computing* 3(3), 273-298, Sept. 1992.

[23] W. Wadge and E. Ashcroft, *Lucid, the Dataflow Programming Language*, Academic Press, London, 1985.

[24] G. Wang and A. Ambler, Solving display-based problems, *IEEE Symp. Visual Lang.*, Boulder, CO, 122-129, Sept. 1996.

[25] N. Wilde and C. Lewis, Spreadsheet-based interactive graphics: from prototype to tool, *ACM Conf. Human Factors in Computing Systems*, 153-159, Apr. 1990.

[26] D. Wolber, Pavlov: an interface builder for designing animated interfaces, *ACM Trans. Computer-Human Interaction* 4(4), 347-386, Dec. 1997.