In Search of a Simple Visual Vocabulary

Elisabeth Freeman and David Gelernter

YALEU/DCS/TR-1073

May, 1995

YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

In Search of a Simple Visual Vocabulary

Elisabeth Freeman, David Gelernter
Yale University

Abstract— *Visual languages are more complex than we would like. We introduce a small but powerful visual vocabulary for a visual programming environment that is simple, yet expressive enough to represent the structure of programs and program executions. This vocabulary is not based on any existing textual language. It was designed for the purpose of visually representing and understanding programs and their executions.*

I. Introduction

VISUAL languages are designed to make programming simpler by representing programming concepts visually. However, many researchers in the field concede that visual languages are still more complex than we would like. When the large collection of constructs that we use in textual programming is translated to a large collection of visual constructs for a visual language, textual complexity is merely replaced with visual complexity. Additional complexity arises when there is no clear relationship between visual symbols and the concepts they represent.

Many visual languages have been inspired by the pictures programmers draw when they are sketching outlines of their programs, flowcharts, or data dependencies. Such languages tend accordingly to visualize programs in 2D. 3D is a more convenient drawing mode for software than programmers, and by extending the visual space to 3D, we can reduce some of the complexity that results from the limitations of 2D space; we can take advantage of the extra dimension in representing concepts and program structure.

Few programming environments have addressed the problem of how to represent a program and its execution in an integrated way. Often, there is no relationship between the representation of a program and its execution: either the program execution is not represented at all and we are only shown the results, or the system uses two different visual vocabularies for source and execution. The result in the latter case is unneeded complexity in debugging. A visual vocabulary that can represent both programs and executions can make it simpler to understand program behavior and allow debugging in the same environment in which the program was created.

The question of what constitutes a good visual vocabulary for a visual programming environment is still open. We propose a visual vocabulary that is simple and small, but expressive enough to represent the structure of programs and program executions. This visual vocabulary is not based on any existing textual language; it was designed from scratch for the purpose of visually representing and understanding programs and their executions. The task of constructing a visual program in this environment is understood, not in terms of the specification of expressions to be evaluated, but in terms of arranging program elements spatially. The programmer specifies which elements are adjacent, which are grouped together and which are distinct, which share one lifetime and which occupy adjacent lifetimes. By executing a program we take a structure in space and turn it into a structure in space-time. The visual constructs we propose provide a way of building and understanding both the program and its execution in an integrated, visual environment.

In the remainder of this paper, we introduce **MAP**, a visual programming environment based on the ISM model[3], and discuss its visual constructs for building programs.

II. MAP

MAP programs are built using *regions*. Regions are containers for basic values (like integers and strings) or expressions, or they can be empty. Regions can be assembled in two ways; into a *space-map* or a *time-map*. By assembling regions into a space-map we specify that they are spatially distinct and share a lifetime. By assembling regions into a time-map we specify that they are distinct in time, but share one living space. The way in which we assemble regions determines their spatial arrangement, their execution semantics and the lifetime during which they are active.

Figure 1 shows the three dimensional structure of maps. Maps are defined spatially on the horizontal (x) axis, temporally on the vertical (y) axis, and with depth on the z axis for nested structures. Space-maps are collections of regions juxtaposed on the horizontal axis; time-maps are collections of regions juxtaposed on the vertical axis (see figure 2). All the regions in one map are considered to be on the same *level.* Maps can be nested: a region can recursively contain an entire map. If a region contains another map, the regions of the nested map are one level down from the regions of the enclosing map (see figure 3). Regions can be accessed by name or position; for example in figure 4, the third region of the space-map *values* can be referred to as *values*[2] or *values.c*.

The MAP visual vocabulary consists only of space-maps and time-maps. These two constructs serve as the basic building blocks for all program structures, from data structures such as arrays and records through local naming environments such as blocks and functions to entire programs.

In figure 5, a record named *myplan* with three elements is created with space-maps. The C code for a record that corresponds to this structure is also shown. The first element is an array, named *spysubjects*, and the other two are basic values, named *starttime* and *endtime*. The record, *myplan*, and the array, *spysubjects*, are both constructed using space-maps with several regions. The only difference between these constructs is that the record has a heterogeneous structure, while the array is homogeneous. In traditional languages a distinction is made between these two kinds of data structures, but in MAP they are two facets of one structure. By simplifying the basic building blocks and their visual representations, we simplify the resulting program.

Figure 6 shows how a compound statement corresponds to a time-map. The sequence of expressions that are to be evaluated are defined with a sequence of time-map regions.

Local naming environments such as blocks and functions are also built using maps. A function is a *template* space-map that can be invoked more than once. Figure 7 shows the function *executeplan* defined in C and in MAP. The function takes one argument, *spyplan*, defines local variables *audio* and *video* and executes a sequence of expressions in the time-map *runplan*.

The next example, in figure 8, shows an entire MAP program. The program *spy* consists of the data structure and function we defined above, and has a top-level time-map with several regions that contain the sequence of expressions to be evaluated. Note that, like the other data structures we have discussed, a MAP program is simply a space-map. We identify this map as the ***program space-map*** in the rest of the paper. As a space-map, a program is a first-class structure and can be manipulated just like any other data structure in the language. We can pass programs, like records, arrays, and functions, to other functions or programs.

III. Evaluation of MAP programs

Recall from the definition of space-maps that the regions in a space-map, while spatially distinct, share one lifetime. This means that when we evaluate a space-map, all of its regions (and any evaluatable expressions they contain) are evaluated concurrently. The evaluation of that space-map yields another space-map with the same shape: a new space-map that looks just like the old one, except each region in the result space-map contains the value yielded by the corresponding region in the source space-map.

In practice, the semantics of evaluation could be realized either synchronously or asynchronously. In the current implementation, each region is evaluated for one *step* in a synchronous sweep from left to right over the program space-map. A step is defined as the evaluation of one expression, but could also be defined in other ways, such as a time quantum, for example. After each region has evaluated for one step, a new program space-map is created so that each region of the program space-map is updated with the results of the evaluation of the previous step. The evaluation continues with successive synchronous sweeps until no more evaluation can take place in any region.

A space-map is evaluated using the following rules: a basic value yields itself; a simple expression yields its value; an empty region remains empty until it is a assigned a value, and any region referring to an empty region *blocks* until it is no longer empty. MAP uses static scoping, so a template space-map evaluates to itself, except that all free variables have been replaced with references to the named regions. (As a result, the behavior of closures in MAP is different from the behavior of closures in statically scoped Lisp; MAP provides other ways to achieve the same results.) A time-map is evaluated sequentially; as each expression is evaluated (with possible side effects to other regions in the program) it disappears, leaving behind only a value (since all program statements in MAP are expressions), then the next time-map region is evaluated.

An example in figure 9 shows the concurrent evaluation of a MAP program with several regions containing expressions and a time-map. Note that the region $z[1]$ is intially empty and is filled in with a value as a result of evaluating the first time-map region.

The sequence of program space-maps that is created as a MAP program is evaluated represents the execution of the program, and is called the ***program history***. The next section describes in more detail how the program history is created and visualized, and how we can use the program history for debugging and understanding of program execution.

IV. Program Histories

Traditionally, the syntax of a source program bears no relationship to the runtime process that is the execution of that program. This is true for textual languages, and for most visual languages as well (a notable exception is Pictorial Janus[7]).

In contrast, the underlying model of MAP allows the program history to be represented using the same visual vocabulary as the source program. As discussed earlier, the process state is captured by forcing the evaluation of a program to yield a sequence of intermediate space-maps en route to the final result space-map. Any space-map in the program history describes the program state at that step and contains enough information to restart the computation at that point.

So, we can visualize program executions as a series of space-maps. But a series of space-maps is itself just a space-map — a space-map being any arrangement of regions in a space of arbitrary dimension. We can manipulate this history using the same construction, deconstruction and evaluation rules that we use for any other data object. A program history can be used as data for visualizing program execution, debugging and communication. It becomes simple to write another MAP program that reads a program's execution as it evolves, or a program that looks at a particular data region or named time-map region for errors, or one that displays some statistics about the program's behavior. Manipulating the program history is no different from manipulating any other data structure.

Viewing a program's execution is valuable when programmers need to correct bugs, improve performance, understand algorithms, and make updates. In the MAP programming environment, viewing and understanding the program history is no different from understanding the source program. Having built the source, the programmer knows what the shape of the program history is. The nested structure of programs allows the programmer to visualize action at the highest abstraction level ("make all second-level regions opaque") or any other level. The fact that the program history is a graphical object makes it possible to rotate it for a better view or zoom in for a closer look at the action. The graphical object together with color will simplify certain important kinds of debugging — if we color some column blue as soon as its value is $> x$, we can tell at a glance where things went wrong. Programmers can watch concurrency taking place, watch access patterns to data structures, etc. The uniform visual vocabulary simplifies the process of analyzing a program's behavior and eliminates the necessity of learning a completely new set of commands to examine and debug a program execution.

Figure 10 shows an example of a program history. The three dimensional source program has been projected onto a plane, in effect squishing the time-maps; time will now be represented in terms of the steps in the program history. In this example, we show how color can be used to highlight a particular value; in this case, *video* is colored grey when its value is greater than 100. This example also shows how maps are used as an abstraction mechanism for visualization; here we are not interested in seeing the entire array *spysubjects*, so the array is only visible as an opaque region.

Any step of the program history can be selected and edited. For example, the user might want to select step 12, change the value of *video* to 8, and restart the computation from that point. Regions of the program history can be selected by name, so that, for example, the region *audio* refers to a vector; namely the vector of values over the entire course of the computation. These values can then be passed as a space-map to another program for analysis.

The visual representation of the program history, using the same visual vocabulary as source programs use, illustrates the idea of the *shape* of a computation. The shape of MAP programs and their executions is determined by the locality and contiguity of regions. The source program structure retains its shape while it executes; thus the shape of the source program also imposes shape on the program history. The shape of the program history in turn determines the shape of the result yielded by the program. (This "retention of shape" is one property that distinguishes the model we describe from dataflow, graph reduction and other models in which the source program may also be represented as a structure in space.)

Saving intermediate state for visualization and debugging has long posed a problem in the development of debugging systems. A program history is potentially gigantic. From a practical standpoint, it is important that the user be allowed to vary the granularity of the program history depending on how much data he needs, and is willing to spend the time and storage space to save. In the current implementation, the synchronous sweep method of evaluation allows recreation of steps, so intermediate steps that are not initially saved can be regenerated. The step size can be adjusted so that, instead of only one expression evaluation occurring at each step, 10 or 100 expressions can be evaluated.

It is important to note, however, that while we want a system that we can run on today's systems with our current storage capabilities, we also know that in principle, storage is cheap and easily expandable. Finding innovative uses for large amounts of storage is part of the exploration and development of new systems.

V. Implementation

The graphical interface for building MAP programs has been developed with the three dimensional structure of maps in mind. MAP programs are constructed using a 3D editor built with *Open Inventor*^{TM}. Open Inventor is an object-oriented tool kit for developing interactive, three dimensional graphics applications. The main window, shown in figure 11, is the work space in which programs are constructed. The icons on the top left side of the window are used for adding space-maps, time-maps and text, naming regions and executing programs. The other icons are used for fine tuning navigation and views, and for selecting the construction mode in which programs are created.

Programs are constructed by adding regions to the work space. Regions are represented by cubes, so maps with nested regions are visualized by nesting cubes within cubes. A basic value, expression, or collection of expressions is entered into a region as text, and names of regions are displayed as text in the top left corner of the region. The name and text of any region can be edited using a 2D text window that pops up when a region is selected and the name or text icon is chosen. Figure 12 shows the *spy* program with MAP interface.

When we evaluate a program, the program history is displayed in the work space, so that the user can access each step to view the intermediate values and results in any region of the program.

MAP represents a compromise between the advantages of having visual representations for programming concepts and the chore of having to use visual representations when it's easier just to type. The visual vocabulary is not designed to represent expressions or control structures, but rather to represent program structure. The goal is not to eliminate text but to embed it in an integrated visual framework.

Although teaching programming is one focus of this research, the problem of scaling-up visual programming environments to accommodate "real programs" is one that many researchers are investigating([1]) and is addressed in the development of MAP as well.

MAP includes two techniques to aid in the visualization and understanding of large programs. First, as we discussed earlier, it is possible to suppress the detail of structures whenever we want, so that only the outermost structure of a particular module or array is visible. Second, we use the technique of *folding*: large horizontal structures such as arrays (or whole programs) are folded by arranging groups of regions on the z axis until the width of the structure is manageable. Folding allows the programmer to see the size of the structure and access its elements without being overwhelmed by too many elements on the horizontal axis.

VI. 3D Space and Navigation

3D environments are beginning to be used more often in the visualization of data([10], [6]), and even for a few visual programming languages([8]). A benefit of using 3D interfaces is that screen space can be used more effectively. (Glinert has argued that it is often unnatural to program, view data structures, and perform other programming activities in less than three dimensions[4].) Key challenges in creating 3D interfaces include developing navigation techniques, overcoming the speed/quality tradeoff, and understanding human-computer interaction in such environments[2].

With three dimensions, all three aspects of program structure can be visualized: nested structure, spatial arrangement of program elements, and temporal arrangement of program elements. The user has more control in viewing context and detail by selecting which regions and levels are visible. He can see at a glance the high level structure of a program: a region containing a large, homogeneous array looks distinctly different from a region containing a record or a basic value.

Navigation in this environment is currently restricted to hierarchical movement in the program or program history space-map. The user can select and move into any region on the current level. Moving into a region changes the focus so that all other regions on the previous level *fade* (less information is provided about those regions) and more detail about the current region appears. This functionality is a result of balancing the speed/quality tradeoff and the detail and context in the display. Levels of the program that are far away from the user, and regions that are no longer in focus, provide less or no detail about their contents, while levels and regions currently in focus display their contents (either nested space-maps or text).

This technique of *zooming* is used to provide the capability to see a far view for an overall picture of a program or section, and a more detailed view, when the user needs to see the structure of a region or edit the text in a region. Zooming is an effective technique for visual browsing and has been used in other environments ([9],[5], etc).

VII. Conclusions and Future Work

Development of MAP is ongoing. Future work will include extending the functionality of the 3D editor to allow full visualization and functionality of program histories, to include more debugging and analysis features, to improve and extend the textual portion of the language, and to improve the navigation facilities. MAP program histories can be used for high-level communication between programs as well as for visualization of program execution. Future work includes further development of this functionality, both in the language and in the graphical user interface. We will also build a utility to take a program in a conventional language and turn it into a MAP structure.

References

[1] Margaret M. Burnett, Maria J. Baker, Carisa Bohus, Paul Carlson, Sherry Yang, and Pieter van Zee. Scaling up visual programming languages. In *IEEE Computer*. IEEE Computer Society, March 1995.

[2] Elisabeth Freeman and Susanne Hupfer. A model for 3D interaction with hierarchical information spaces. Position Paper for CHI '95 Research Symposium, May 6-7 1995. Denver, Colorado.

[3] David Gelernter and Suresh Jagannathan. *Programming Linguistics*. MIT, 1990.

[4] Ephraim P. Glinert. Out of flatland: Towards 3-d visual programming. In *Visual Programming Environments: Applications and Issues*, pages 547–554. IEEE Computer Society Press, 1990.

[5] Michael Gorlick and Alex Quilici. Visual programming-in-the-large versus visual programming-in-the-small. In *IEEE Symposium on Visual Languages*. IEEE Computer Society, October 4-7 1994. St. Louis, Missouri.

[6] S.K. Card J.D. Mackinlay, G.G. Robertson. The perspective wall: Detail and context smoothly integrated. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, pages 173–179. ACM, 1991.

[7] Kenneth Kahn and Vijay Saraswat. Complete visualizations of concurrent programs and their executions. In *IEEE Workshop on Visual Languages*. IEEE Computer Society, October 1990.

[8] Marc-Alexander Najork. *Programming in 3 dimensions*. PhD thesis, University of Illinois at Urbana-Champaign, 1994.

[9] Ken Perlin and David Fox. Pad: An alternative aproach to the computer interface. In *SIGGRAPH 93 Conference Proceedings*. ACM SIGGRAPH, August 1993. Anaheim, California.

[10] George G. Robertson, Stuart K. Card, and Jock D. Mackinlay. Information visualization using 3d interative animation. In *Communications of the ACM*, volume 36, pages 57–71, April 1993.

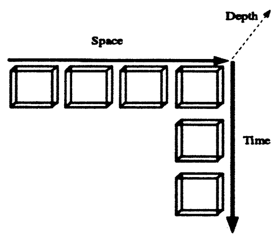

Fig. 1. Maps.

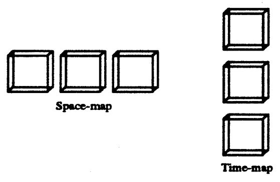

Fig. 2. Space-map and Time-map.

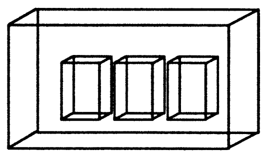

Fig. 3. Nesting.

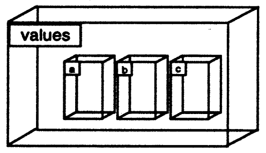

Fig. 4. Accessing maps by name or value.

```
struct myplan {
    string spysubjects[5];
    int    starttime;
    int    endtime;
}
```

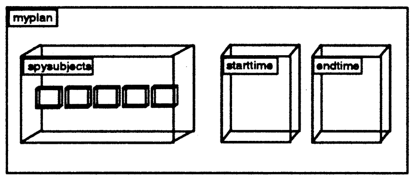

Fig. 5. Record.

```
{
   a; b; c; d;
}
```

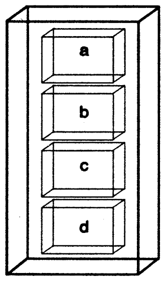

Fig. 6. Compound statement.

```
int executeplan(int spyplan)
{
   int audio;
   int video;
   runplan:
     ....
}
```

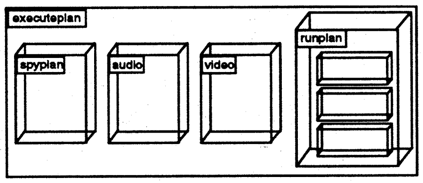

Fig. 7. Function.

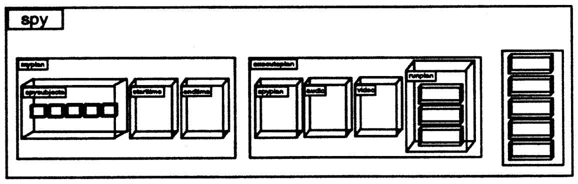

Fig. 8. MAP program.

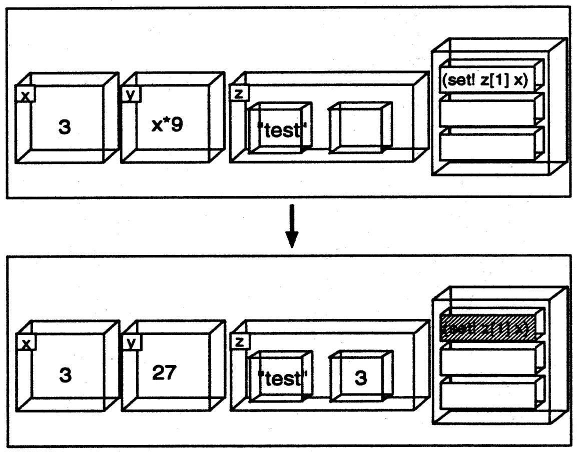

Fig. 9. Concurrent evaluation.

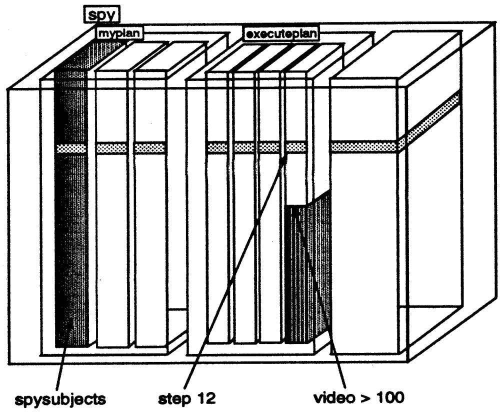

Fig. 10. Program history.

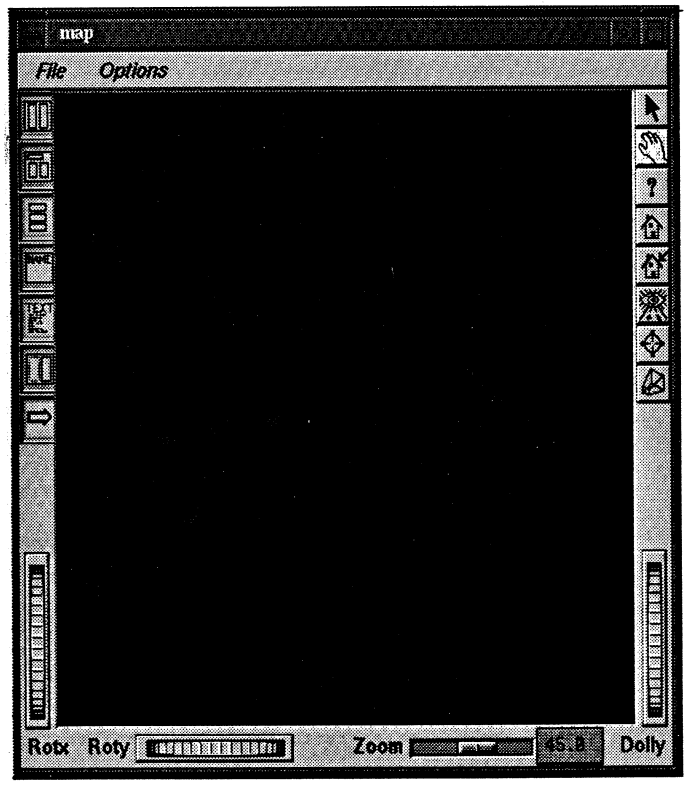

Fig. 11. MAP interface.

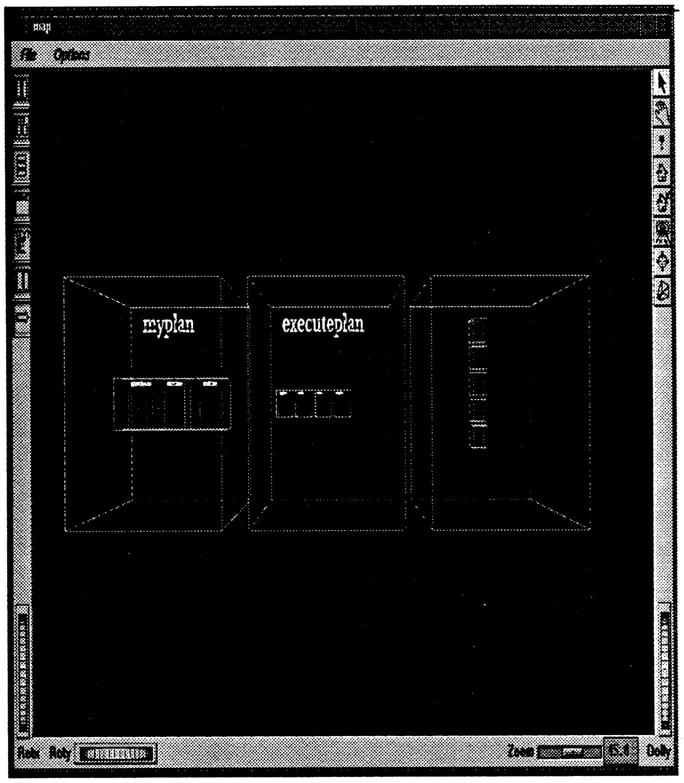

Fig. 12. MAP program.