The Visual Programming Environment

Elisabeth Freeman
Department of Computer Science
Yale University

April 27, 1993

1 Introduction

This paper presents **Visual**, a programming environment based on the Ideal Software Machine model (ISM)[1]. In Visual, programs and data structures are described graphically. Visual has several advantages over traditional programming languages. Program structures that are traditionally considered as different objects are unified. Procedures, arrays, records and other aggregate data objects have the same graphical structure. This simplifies the language as the programmer is no longer required to know more than one way of building a program structure—every structure is built the same way.

Another advantage is that we introduce a third dimension to the conventional two-dimensional structure of programs. This third dimension is the nesting level of programs. When nested objects occur in a program, the program is graphically structured so the programmer must go *inside* the objects to get to the inner structure. This three dimensional view of programs provides a new method of abstraction and modularity.

The inherent parallelism in the ISM is also an advantage. No special mechanism is needed to build parallel programs in the Visual environment. As we will see, parallelism comes naturally from the way programs are defined.

Program execution is visualized using the same graphical structures used to build programs. This visualization lets the programmer view each step of the program execution, and acts as a high level debugger with access to the history of execution. This is an improvement over current debugging technology, such as **dbx**, because the programmer does not have to learn new syntax to debug and understand the program execution.

In the following sections we present the Visual programming environment. First we briefly introduce the ISM, and then describe the syntax and semantics of Visual programs. Examples of how to build Visual programs are then given, and the implementation of the system is described, including a description of the parallel execution environment and the program visualization system. Finally, the shortcomings of the current system and goals for the future are presented, along with a discussion of related work.

2 The ISM

ISM programs are created in a space-time coordinate system (see figure 1). Programs are built with two constructs: *space-maps* and *time-maps*. Space-maps are collections of regions that are arranged in space (horizontally), while time-maps are collections of regions arranged in time (vertically).

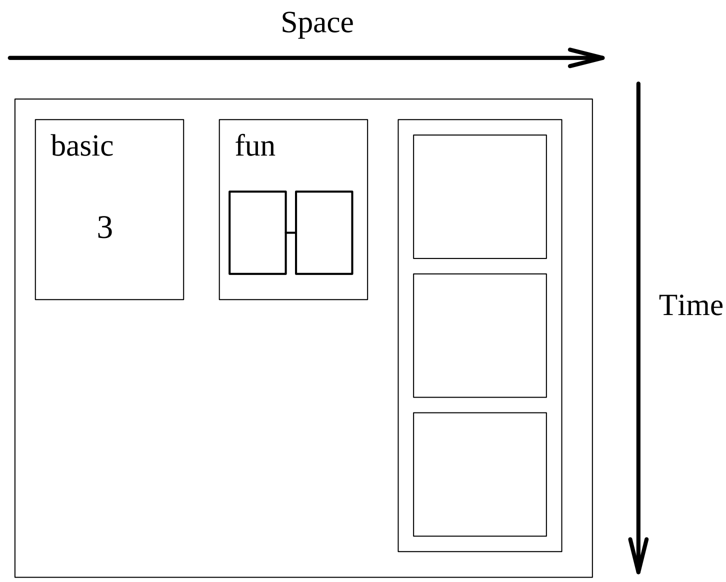

Figure 1.

Space-map regions can contain *basic values* (such as integers, booleans and simple expressions), other space-maps or time-maps. Space-maps are used to represent all program structures—the program itself, data structures such as records and arrays, and functions. They are first class objects and can be passed as data to other functions or programs. Space-maps are lexically scoped. Time-map regions contain program statements with one program statement in each region. Programs are inherently parallel—a space-map with regions containing basic values, space-maps or time-maps is executed by evaluating all regions concurrently. Adding more than one time-map to a space-map creates more than one thread of control.

For example, figure 1 is an ISM program having one thread of control. The illustrated space-map has three regions. The leftmost region contains a basic value, 3, and is named `basic`. The middle region contains a nested space-map indicated by the double box icon, and is named `fun`. The rightmost region contains a time-map, also containing three regions. Adding another time-map to the space-map would add another thread of control to this program.

ISM programs are executed by evaluating every region in a space-map concurrently. Time-maps are executed sequentially by evaluating one program statement at each time step. The last statement in a time-map returns the value of the right hand side (all program statements are expressions). At every time step a new space-map is created. These space-maps have regions corresponding to the source program. Each region is updated to reflect any side effects the previous program statements may have had on the value or data structure in that region. The final space-map in the execution of the program (if it terminates) is the result space-map. The source, intermediate and result space-maps make up the *space-time volume*, and represent the *process*. The space-time volume is itself a space-map and thus is a first class object just like every other space-map.

3 The Visual programming environment

The ISM lends itself well to a visual programming environment. Visual implements the fundamental concepts of the ISM, representing space-maps and time-maps as graphical structures. Programmers build programs with space-maps and time-maps using the graphical user interface. A program is defined by the outermost space-map, and all other space-maps and time-maps are added to the program space-map. Space-map windows may be iconified to named buttons which provides a useful abstraction mechanism for managing the complexity of large or deeply nested programs. Nested program structures are represented with nested space-maps, which helps to organize a program and keep it modular. The nested structure of a program can also be viewed in the three-dimensional visualization system (described in section 4).

The graphical interface for constructing Visual programs is shown in figure 2. The interface consists of a menu of icons representing program constructs and utilities, and a programming window in which to construct programs.

3.1 Space-maps

Primitive data values and simple expressions are represented with the *basic value space-map*, shown in figure 3. Basic value space-maps bind a name to a value or simple expression.

An *aggregate value space-map* is a space-map that corresponds to an aggregate data structure, such as a record, block or array. Functions are also represented with aggregate value space-maps in Visual. These structures are all built in the same way, by adding space-maps and time-maps to the aggregate value space-map representing the structure.

Figure 4 shows an aggregate space-map that corresponds to a C or Pascal block. This space-map contains several space-maps and a time-map, corresponding to local bindings and the body of the block, respectively. The lexical scoping of Visual programs is easy to follow: a space-map S defines a local scope for all space-maps and time-maps within S, and the nested structure clearly shows if a new binding shadows an older binding.

A function space-map is similar to a block space-map, except that it creates a function which is bound to the name of the space-map. For example, in figure 5, the space-map $f()$ is bound to a function with local variables $counter$ and $result$, and one time-map.

An array space-map is constructed like a block or function space-map. Regions in the array space-map are elements of the array. The size of an array space-map can be specified in the name field, and if the number of elements in the array space-map is less than this size, whatever is in the last specified region is copied into each of the remaining regions. Figure 6 shows an example of an array with three basic value space-map elements.

3.2 Time-maps

A time-map contains program statements. These statements are similar to statements in the C programming language. In addition, these statements have been extended to include space-maps. Four types of time-map program statements are available; simple expressions, assignment, conditional and iteration. An expression is *simple* if it does not contain a space-map definition, so simple expressions contain only text. The other three time-map expressions contain space-map regions that are used to define a space-map within the expression.

The assignment statement is shown in figure 7. The programmer enters a variable name on the left hand side of the expression (the variable must be defined in the current scope) and constructs a space-map on the right hand side. The assignment statement in figure 7 assigns a function $f()$ to a variable x.

The conditional statement uses an `if` expression. The programmer enters a boolean expression for the test, and then defines two space-maps, the first executed if the conditional succeeds and the second if it fails. A conditional statement with block space-maps is shown in figure 8.

The iteration statement consists of a control expression and a space-map for the body of the loop. The control expression is a limited version of the C `for` statement and has the following syntax:

$$\texttt{for}(expression_1; expression_2; expression_3)$$

where $expression_1$ is an initialization (eg, $i = 0$), $expression_2$ is a boolean expression (eg, $i < 10$), and $expression_3$ is an assignment statement (eg, $i = i + 1$). The example in figure 9 shows a loop with a block space-map as the body of the loop.

3.3 Windows and Buttons

The three primary menu items for constructing programs are the two space-map buttons and the time-map button. The basic value space-map button adds a basic value space-map to the programming window, and the programmer is prompted for a name and value. When the aggregate space-map menu item is selected, the programmer is again prompted for the name of the space-map, and when the name is entered, a space-map button appears in the programming window. To add nested space-maps to this space-map, the programmer clicks on the button, which opens the space-map window, and this space-map becomes the new programming window. Any space-maps or time-maps are added to this window until the programmer closes the window with the close button in the left top corner of the window.

Similarly, when the time-map menu item is selected, a time-map button appears in the programming window which opens the time-map window. Any program statements that are selected are added to this time-map window until the window is closed.

The following section illustrates how to build a Visual program with some example programs.

3.4 Example programs

The first example, shown in figure 10, is a simple program that defines a value for a variable x and returns this value. A basic value space-map is used to define x, and one time-map containing a simple expression returns the value of the variable. To add the statement to the time-map, the programmer selects the `expr` statement button and enters the text to return the value of x.

The second example shows how to build a slightly more complicated program using a function. This example builds the *factorial()* program and is shown in figure 11. The programmer adds a function to the program by selecting the aggregate data space-map menu item. When the space-map button is clicked, a window opens up in which the programmer can add more space-maps and time-maps. For the factorial function, only a time-map is needed, so the programmer adds a time-map and then chooses a conditional statement. This adds a template for building the `if` expression. After entering the text for the test, the programmer defines the two space-maps each containing one time-map used like a traditional C or Pascal block. The main program also has one time-map which calls the *fact()* function and returns the value.

Since each region of a program is evaluated concurrently, adding more than one flow of control is just a matter of adding more than one time-map to the program. Figure 12 demonstrates a simple

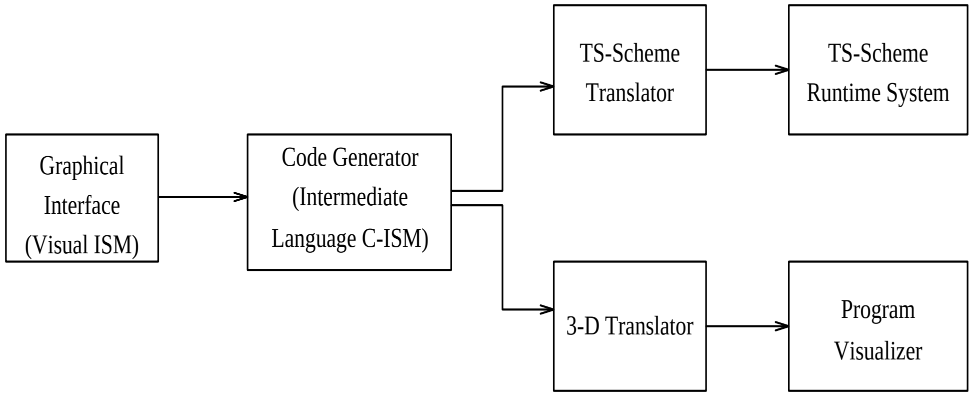

Figure 16.

parallel program using the *factorial()* program from above. The programmer adds a second time-map to the top level program which also calls the *fact()* function. In this program two time-maps are thus executed concurrently.

Figure 13 shows a program that uses an array. The array *a[]* is constructed in the same way the function *fact()* was created—using an aggregate data space-map button. Array elements are accessed by indexing the space-map. The data regions within the array are given names, although in the current implementation, these names cannot be used by program statements.

All program structures in the Visual programming environment are first-class objects. Any space-map may be passed as data to a function or returned from a function. These example programs use space-maps to represent all program structures—the program itself, functions, blocks and arrays. The uniformity of program structure clearly demonstrates that the natures of these constructs are similar, a fact hidden in many languages with complex and widely varying syntax.

These are all simple examples and in reality a programmer will seldom want to use nested space-maps for every block or simple expression. Instead, programmers use the graphical structure of the Visual environment as an organizing tool, using space-maps and time-maps for program modularity. Figure 14 shows an example of a compiler in which the program is organized at a very high-level using space-maps to represent the most general modules of the program. This program can easily be changed to a parallel program by using multiple time maps, as shown in figure 15. Using the Visual environment to organize a program gives the programmer a high-level view of his program and the ability to re-structure a program easily as we did in changing the compiler from a sequential to parallel program.

4 Implementation

Visual is implemented by several modules. Figure 16 displays an overview of the implementation. The front end to the system is the Visual graphical interface. A program built in Visual is translated to an intermediate language, C-ISM, by a code generator. This intermediate language is then used by the backends of the system. Currently, there are two backends: a runtime system that first translates intermediate code to TS-Scheme and then executes it, and a three-dimensional program visualization system.

4.1 Graphical Interface

The graphical interface is implemented in Motif. The source program is stored in a tree structure that is built as the programmer creates the program. Each node in the tree is either a space-map or a time-map, and contains the textual information that the programmer enters in the base language (a subset of C-ISM). Once the source program is built and ready to execute, the programmer selects the run button (shown in figure 2), and the program is translated into the intermediate language by the code generator.

4.2 Code Generator

A Visual program is first translated to C-ISM (so called because the language is similar to C). Figure 17 shows the C-ISM code for the parallel $factorial()$ program in figure 12. Using C-ISM as an intermediate language allows the backend systems to be changed without changing the implementation of the graphical interface.

The text the programmer has used in the program remains the same, since it is already written in a subset of C-ISM, while the graphical structure of the program gives structure to the intermediate code.

A C-ISM intermediate program is created from a graphical Visual program by traversing the tree in which the graphical program is stored. The root of the tree is the top-level space-map, and the nodes are space-maps and time-maps.

The top-level space-map is translated differently from node space-maps, and defines a program:

$$\texttt{program} < \mathit{name} > < \mathit{body} >$$

Node space-maps are bindings within a block (defined by a program or space-map):

$$< \mathit{name} > : < \mathit{body} >$$

where $< \mathit{name} >$ is the name of the space-map and $< \mathit{body} >$ is the space-map definition.
A basic value space-map is translated to a binding of the name to the expression:

$$< \mathit{name} > : < \mathit{expression} >$$

A block space-map is translated to a block binding, delimited with $\{\}$:

$$< \mathit{name} > : \{ < \mathit{body} > \}$$

A function space-map is translated to a template using the `template` keyword:

$$< \mathit{name} > : \texttt{template} < \mathit{parameters} > < \mathit{body} >$$

An array space-map is translated to an array definition and is designated with `[]`. Parentheses are used to delimit the array:

$$< \mathit{name} > \texttt{[}< \mathit{size} >\texttt{]} : \texttt{(} < \mathit{element}_1 > < \mathit{element}_2 > ...)$$

Each time-map is translated to a `begin end` block,

$$\texttt{begin} < \mathit{body} > \texttt{end}$$

where $< \mathit{body} >$ consists of all program statements and space-maps in the time-map. If there is more than one time-map, the blocks appear consecutively in the C-ISM code.

```
program factorial()
fact: template(n)
        begin
        if (n==0) { begin
                       return 1;
                    end };
                  { begin
                       return (n*fact(n-1));
                    end };
        end;
begin
   return fact(4);
end
begin
   return fact(5);
end
```

Figure 17: C-ISM code for parallel $factorial()$

```
(define factorial (lambda ()
   (labels ((fact (lambda (n)
                    (labels ()
                      (block (if (eq?  n  0 )
                               (labels ()
                                    (block  1 ))
                               (labels ()
                                    (block (*  n (fact (-  n  1 ))))))))))))
       (timemaps (block (fact  4 ))
                 (block (fact  5 ))))))
```

Figure 18: TS-Scheme code for parallel $factorial()$

4.3 TS-Scheme translation

We execute Visual programs by translating C-ISM code into TS-Scheme[5, 6] and evaluating the program using the TS-Scheme interpreter. C-ISM programs are translated to TS-Scheme using the following transformation rules.

A C-ISM program definition is translated to the special form `define`. All space-maps within a program become bindings in a mutually recursive `let` special form (`labels`) within the program definition. The nested structure within a block space-map is translated to a nested `labels` binding:

$$\begin{array}{lcl} < name_1 > : < body_1 > & & \texttt{labels}\ ((< name_1 > < body_1 >) \\ \vdots & \Rightarrow & \quad\vdots \\ < name_i > : < body_i > & & \quad(< name_i > < body_i >)...) \end{array}$$

A function space-map is translated to a `lambda` expression:

$$\begin{array}{lcl} < name > : & & \texttt{labels}\ ((< name > \\ \quad\texttt{template}\ (< parameters >) & \Rightarrow & \quad(\texttt{lambda}\ (< parameters >) \\ \quad\quad< body > & & \qquad\qquad< body >))\ ...) \end{array}$$

An array space-map is translated to a `vector`. There are two approaches to translating an array. The first creates a vector with the nested bindings as the initial values of the elements of the vector, while the second creates the vector with no initial values, and then sets each element of the vector to the nested binding. The current implementation uses the second method for defining an array, with the consequence that the array elements are evaluated *before* they are added to the array. The translation requires the use of a temporary name for the vector, so that the elements of the vector can be set within the same binding definition:

$$\begin{array}{lcl} < name >[< size >]: & & \texttt{labels}\ ((< name > \\ \quad(< element_1 >, & \Rightarrow & \quad(\texttt{let}\ ((\texttt{\_tmp}\ (\texttt{make-vector}\ < size >))) \\ \quad\;< element_2 >, & & \qquad(\texttt{vset \_tmp 0}\ < element_1 >) \\ \quad\;...) & & \qquad(\texttt{vset \_tmp 1}\ < element_2 >)\ ...)) \end{array}$$

This approach has the advantage that TS-Scheme functions are evaluated to procedure objects before being added to the array and can thus be accessed as procedure objects for evaluation and application. One disadvantage of using vectors to represent arrays is that names given to space-map definitions in array space-maps are lost in the translation.

C-ISM `begin end` blocks, corresponding to Visual time-maps are translated by binding each `begin end` block to a `future`:

$$\begin{array}{lcl} \texttt{begin} & & \\ < body_1 > & & (\ \texttt{labels} \\ \texttt{end} & & \quad((\texttt{TM}_1\ (\texttt{future}\ < body_1 >)) \\ \vdots & \Rightarrow & \quad\vdots \\ \texttt{begin} & & \quad((\texttt{TM}_\texttt{i}\ (\texttt{future}\ < body_i >)) \\ < body_i > & & ...) \\ \texttt{end} & & \end{array}$$

The TS-Scheme code that is actually generated uses a macro, `timemaps`, which converts to the above syntax when the code is loaded into the TS-Scheme interpreter. Using the macro, the translation from a time-map to TS-Scheme is:

```
begin
< body_1 >
end
⋮                ⇒  ( timemaps < body_1 > ...< body_i >)
begin
< body_i >
end
```

Figure 18 shows the TS-Scheme code for the parallel $factorial()$ program.

4.4 Visualizing programs

One of the goals of a visual programming environment is to give a program the feel of a physical object. The programmer should be able to manipulate the object, see it executing and view the results. A first step in this direction is the visualization of programs and results as three dimensional objects. The nested structure is represented with nested cubes, each cube representing a region in the program. So far, only the static program and result can be viewed; however, we would like to add functionality to visualize the process as it is created by the execution of the program. As program statements are executed, each new space-map in the space-time volume is added to the visualization, showing the programmer what is happening at each step in the program and creating a graphical representation of the process. The program, the process and the result are all built using the same uniform language, are all visualized using the same visual vocabulary, and are all first-class objects.

Figure 19 shows a visualization of the factorial program. The top level program, *factorial()* is the outermost cube, while the nested program structure shown in figure 20 can be found by zooming into the cube. Zooming into the *fact()* function reveals the inner structure of this nested function as shown in figure 21. Time-maps are represented with *TM* cubes and do not have any inner structure in the current implementation.

5 Future work

The current implementation of Visual gives a flavor of programming in a graphical environment, however there are still many issues to be resolved. The following sections address some problems and describe future work that is planned for the system.

5.1 Parameters

In the current implementation, parameters can only be passed by typing the name of the parameter in the text field of the function name and function call. Function parameters are actually a space-map with one or more regions corresponding to the number of parameters. The future implementation will allow space-maps as parameters to functions.

5.2 Uniform access to space-maps

All Visual program structures are built and represented with space-maps. Although they may be used in different ways, the fact that they are all space-maps implies that each object can be manipulated in the same manner. In the current implementation, each structure is mapped directly to the corresponding TS-Scheme construct, imposing limits on the ways that each can be manipulated. For example, an array space-map can be referenced with the expression $a[i]$ which represents the i'th region in the array space-map. It should also be possible to access the i'th region of a block space-map with indexing, but the current translation does not allow this since blocks are mapped to **let** expressions.

Every region of a space-map can be named. A reasonable goal for a future implementation is to allow space-maps to be accessed by name or by index. A record space-map, similar to a C structure or Pascal record, would allow access by name to the elements in the space-map. For example, in a space-map f with two elements, *name* and *age*, these elements could be accessed by name using $f.name$ and $f.age$. To maintain the uniformity of all space-map structures the programmer should also be able to access these elements using the syntax $f[0]$ and $f[1]$. This uniformity should apply to all space-maps, so that any structure can be accessed by name or by index.

Indexing space-maps provides a way to manipulate other Visual structures such as the space-time volume and the result space-map. The space-time volume is a space-map that can be used just like any other space-map. Visual programs can use indexing to access any element of the space-time volume, or any element of the result space-map.

5.3 Functions should be visible objects

In the current implementation, function space-maps are translated to TS-Scheme `lambda` expressions. A disadvantage to this approach is that a procedure object in TS-Scheme is not "visible"; the programmer can't look at the object after it has been evaluated. Since functions are space-maps in the Visual environment, it should be possible for the programmer to look at a function at any time. Alternative approaches for implementing functions will be explored in future work.

5.4 Parallelism

Programs with more than one time map are concurrent programs since all time-maps are executed in parallel. A program definition is also a concurrent definition, since all regions—even passive data structures that don't actually compute anything—are evaluated at the same step. Although time-maps are executed concurrently using the future mechanism, space-map definitions are mapped to **labels** bindings in the current implementation, and the concurrency of the data definitions is lost. An alternative that maintains the concurrency of all space-map regions is to evaluate each space-map region concurrently using the TS-Scheme **spawn** function (like the C-Linda **eval** function) and to evaluate the time-map regions relative to the tuple-space in which the data definitions are stored. Using tuple-space to store space-map regions also provides a way to add synchronization mechanisms to the language. All regions in a space-map are evaluated synchronously, but there is currently no way for separate time-maps to communicate, except through variables. A shared variable could be implemented as a tuple, thus allowing synchronization of, and communication between, time-maps.

Data parallelism is also inherent in Visual programs—an operation can be made data parallel by distributing the operation over all regions of a space-map and then using the normal space-map evaluation rule of evaluating the regions concurrently. Data parallelism has not yet been implemented, but could be added easily with distributing operators, or recursive application of operators to each region.

An important programming concept that has not yet been implemented is that of dynamic memory allocation. Dynamic allocation is used both for allocating space-map regions and time-map regions; for example, a parallel program may need to allocate n time-maps for n operations, where n is determined at run-time. Nested parallelism, however, can be obtained in the current implementation by recursively calling a function with more than one time-map. An example of a program with nested parallelism is shown in figure 22.

5.5 Visual Complexity

An implementation of the Visual programming environment must deal with many problems similar to those of visual programming systems. Many elegant visual programming languages are not usable for large programming projects because they become unwieldy as the size of the application increases [9].

High consumption of screen space is one of the biggest problems facing any graphical programming environment. Programs not only become semantically complex, but also visually complex as they become large and require large amounts of screen space. Expert programmers become frustrated with these environments because many require lengthy steps to set up and design programs. The graphical representation of programming concepts to help novice programmers can be a hinderence to experts.

Techniques for window management such as zooming and compression can help with some of the problems facing graphical programming systems. Abstraction is a vital part of the system—once a programmer has finished building one portion of the program, this portion can be made abstract by zooming out and making it smaller, or by iconifying it. The Visual programming environment uses abstraction of program elements to help keep programs modular, manage screen space, and reduce the visual complexity. The three-dimensional nested representation of programs is one way to help ease the above problems and a natural way to represent the nested structure of programs. However, there is minimal navigation capability and the problem of programmer frustration still needs to be addressed.

5.6 Process Visualization

Process visualization is an important part of the Visual programming language. It gives the programmer a way to view the program as it executes and to debug the program once complete. There are several problems that need to be addressed by the implementation, including navigation within the space-time volume so that the navigator does not get lost, and elegant representation of complex programming constructs such as iteration, recursion, parallelism, multi-dimensional data structures and procedure objects.

As a first attempt, the three dimensional view of programs currently implemented allows the programmer to zoom into a program to see the inner structure of nested space-maps. However, the issues raised above need to be addressed in a future implementation.

5.7 A true three-dimensional programming environment

We plan to extend the current graphical interface to a three-dimensional interface that takes advantage of the three-dimensional, nested structure of programs built in the Visual environment. As mentioned above, using a three-dimensional environment requires a robust navigation system. A "you are here" mechanism is needed to keep the programmer from getting lost. This mechanism shows the programmer where the current focus in the program is in relation to the rest of the program. For example, this could be implemented as a map in a separate navigation window. The map could be a tree-structured view of the program, with nodes representing space-maps and time-maps. As the programmer moves more deeply into the program, new nodes appear and the map expands, highlighting the node representing the current position in the program. The nested graphical program structure is used to keep programs modular and well organized, and also as an abstraction mechanism.

6 Related work

Visual differs from most visual programming systems in that the graphical nature of Visual programs is based on the view of a program as data—space-maps are used to represent traditional data structures such as records and arrays, as well as blocks, functions and the program itself. Many visual programming languages use data flow and program control flow as the graphical element of a program. For example, some visual languages use icons to represent operations, iteration, conditionals and procedures, and connect these icons with arrows to show data flowing from one element to the next[7, 2, 10]. DataVis[3], a language for scientific visualization, does have graphical representation for data objects, which consist of scientific data such as vectors, data fields and bitmaps. DataVis still uses the data flow model to connect these data objects with program elements and operations represented by icons. These languages are one-dimensional for the most part, the arrows between icons representing the flow of data in time. One programming evironment which does use a three-dimensional view of a program as it executes is Lieberman's three-dimensional representation of program execution[8]. Here again, the focus is on program control flow with successive program expressions appearing as nested cubes. The three-dimensional view of Visual programs in space, time, and nesting level simplifies the graphical representation of programs considerably and provides natural organization, modularity and abstraction to the static program. When the program is executing, the space-time volume clearly shows program control flow, and values are updated showing data flow.

Another way in which Visual differs from many visual programming systems is that few incorporate a general purpose programming language. Many previous systems have been implemented for diverse domains such as music, constructing user interfaces, image processing, graphics and science[4]. Since Visual is designed as a general purpose programming system, it can be used both to teach programming concepts and to build real applications.

References

[1] David Gelernter and Suresh Jagannathan. "Programming Linguistics", The MIT Press, 1990.

[2] E.P. Glinert and Steven L. Tanimoto, "Pict: An Interactive Graphical Programming Environment", *IEEE Computer*, pp 7-25, 1984.

[3] Daniel D. Hils. "DataVis: A Visual Programming Language For Scientific Visualization", *ACM Computer Science Conference*. 1991.

[4] Daniel D. Hils. "Data Flow Visual Programming Languages", *Department of Computer Science, University of Illinois*. 1990.

[5] Suresh Jagannathan. "A Generalized Framework for First-Class Tuple-Space Systems (Preliminary Report)", NEC Research Institute, 1992.

[6] Suresh Jagannathan and Jim Philbin. "A Foundation for an Efficient Multi-Threaded Scheme System", NEC Research Institute, 1992.

[7] Takayuki Dan Kimura et al. "Show and Tell: A Visual Programming Language", *Visual Computing Environments*, E.P. Glinert (editor), IEEE Computer Society Press, 1989.

[8] Henry Lieberman. "A Three-dimensional Representation For Program Execution", *1989 IEEE Workshop on Visual Languages*, October 4-6, 1989.

[9] Nan C. Shu. "Visual Programming", Van Nostrand Reinhold Company, Inc., 1988.

[10] Franklyn Turbak. "GRASP: A Visible and Manipulable Model for Procedural Programs", unpublished, MIT MS Thesis.