Toward Visual Programming Languages for Steering Scientific Computations

Margaret Burnett
Oregon State University

Richard Hossli, Tim Pulliam, Brian VanVoorst, and Xiaoyang Yang
Michigan Technological University

Imagine running a computationally intensive model and being able not only to visualize the data as soon as results start emerging, but also to experiment with and change the underlying calculations in midstream, all with a consistent visual interface.

THE GOAL OF RESEARCH IN SCIENTIFIC VISUALIZATION is to give scientists and engineers productive ways to work with data. The benefit of visual representations is that they are generally easier to comprehend than their textual equivalents.

Two orthogonal areas of scientific visualization research have been emerging in recent years. One group strives to increase the *power* of the science and engineering researcher (hereafter abbreviated "researcher") through work on *steering*. Steering means the researcher can give feedback to the computer, affecting both the visualization and the computation as it progresses. The other group focuses on the *process* by which researchers work with computations and visualizations by providing *visual programming languages* for them to use. These languages differ from traditional programming languages because they are at least partially graphical in nature. This article focuses on research leading toward a marriage of these two areas—visual programming languages for steering:

> The most exciting potential of visualization tools is not the still color photographs or movies a scientist can produce to portray the results of his simulation or computation, but the ability to test and develop code interactively.

1070-9924/94/$4.00 © 1994 IEEE

The scientist should be able to spot visual anomalies while computing and immediately steer, or modify the calculations, to test out new theories. Interactivity puts the scientist back in the computing loop; the scientist gets to drive the scientific discovery process.[1]

What Is a Visual Programming Language?

The syntax of a visual programming language consists of combinations of text, pictures, and other geometric figures, spatial relationships between these entities, and/or a sequence of direct graphical manipulations. The programmer is largely freed from using one-dimensional sequences of commands, pointers, and abstract symbols to express relationships that can easily be expressed spatially. Some visual programming languages concretely represent data or logic or both, and the data and logic respond immediately and visually to any change. This immediate visual feedback supports an exploratory style of programming, one of the most important strengths of visual programming languages for science and engineering research.

A scenario

The following scenario, based on a current scientific application at the US National Aeronautics and Space Administration, shows how this kind of capability might be used. It represents a goal, not the current state of the art.

Picture yourself studying a particular airplane design during descent. The application runs on a supercomputer and generates 400 gigabytes of raw data. The supercomputer maintains a queue of researchers' programs, and executes them as time and priorities permit. Your application has been running in this way for many weeks and has not yet finished.

Whenever the data for a new time step are calculated, they are immediately made available, along with the data accumulated from previous time steps, for viewing on a graphics workstation. Graphical representations are continuously produced by a program you wrote—not by writing graphics code in C, but rather by using a visual programming language.

As the results start to appear, you can also change these representations by directly manipulating parts of the program as well as the data. Because the visual programming language is interpreted and interactive, it supports both ad hoc and permanent changes to the graphics programming, allowing you to experiment with the graphics and to view multiple representations of the data simultaneously with complete flexibility. Not only can you view data from the current or previous time frames; you can also view an animation of data spanning any subset of this time range. All this can occur without using traditional programming languages or turning to a professional programmer to produce or modify the graphics.

As you interact with the emerging results, you decide that the calculations themselves need to be changed. With the same visual programming language, you make the changes and add additional computations within the scientific application (changing the wind speed parameter value, for example). By moving the application back in time a few steps, you can compare the results of the calculations under this new parameter value with those previously obtained.

Steering can involve more than simply changing parameter values. If these experiments lead to the conclusion that there is incorrect logic in the application, in our scenario you can replace the faulty section with new code, programmed with the same language. Similarly, it is possible to redesign portions of the model itself. For example, after performing several experiments, you might decide to improve the aircraft's wing design. By changing the relevant section of the model using the visual programming language, you can enter this redesign without exiting the program to see if the airplane's simulated behavior improves as a result of the change. There is no restriction on the kinds of experiments and changes you can make: the model's equations, the algorithm used, the values of variables, and the data structures can all be modified without having to start over.

Could this scenario really work? Some would say no, that it is not possible to add or replace code while a program is executing, and that even if it were possible it would be unwise. Yet conventional debuggers, perhaps the most useful of all programming tools, have long supported exactly this kind of capability on a simpler scale in the textual programming world.

Still, the current state of technology is far more cumbersome than that depicted by our scenario. Although much progress has been made in many areas of scientific visualization, the potential to provide a flexible, easy-to-use experimentation environment has yet to be realized. Barriers remain, artificially separating simulation from analysis, algorithm from result, action from reaction.

In this article we investigate the research that is moving current technology toward the goal of providing the flexible, visual experimentation environment we've just described. Although this is not a complete survey, a few representative systems will illustrate important points.

Steering with visual languages: A taxonomy

We classified the two areas of research—computational steering and visual programming languages—in six dimensions to provide insights and to suggest future research directions. The six dimensions draw in part on earlier work.[2] The first four measure how thoroughly a system fulfills significant attributes of the requirements of our scenario (see Figure 1). The fifth and sixth dimensions help clarify the advantages and disadvantages of different approaches by looking at the ways they use various programming paradigms. The dimensions are

(1) the system's steering capabilities,
(2) the power and visual extent of the interface,
(3) the level of support for preexisting scientific application programs,
(4) system generality,
(5) the programming paradigm used for scientific programming, and
(6) the programming paradigm used for visualization and steering.

The first two dimensions together, shown along the x and z axes in Figure 1, measure the extent to which steering through visual programming languages has been achieved. The x axis indicates the extent to which a scientific visualization system has achieved steering capabilities. If the visual representation of the data is available only after the application has completed (known as *postprocessing*), the researcher must wait until the application finishes, and then can only analyze the data actually produced. That analysis cannot be used to change parameters or otherwise affect the scientific application. If the pictorial representation is produced and updated continuously as the application executes (termed *tracking*), the researcher's waiting time for feedback is greatly reduced. Still, the decisions that can be made from the data can be used only to abort the application, starting over with new parameters if desired. The flow of information under tracking is still strictly in one direction.

Interactive visualization and steering involve bidirectional exchanges of information. Unlike postprocessing and tracking, these two terms have not been used consistently in the literature. In this article, we use the term *interactive visualization* to describe tracking with feedback, allowing dynamic changes in the way visual representations appear. Following the report by the NSF Panel on Graphics, Image Processing, and Workstations,[1] we use the term *steering* to mean that the user may, upon seeing the graphical representation of the data, use it to make decisions and interact with the executing application, adjusting parameters, backing up to a certain point before continuing, and even changing the model or data structures. Key to the notion of steering is the ability to make *unanticipated* changes to the *data and logic* of an application, not just changes to input parameters. Steering then consists of three features:

(1) continuous display of data (tracking);
(2) the ability to modify a visualization interactively at any time (interactive visualization); and
(3) the ability to modify any aspect of the application (not just input parameters) at any time.

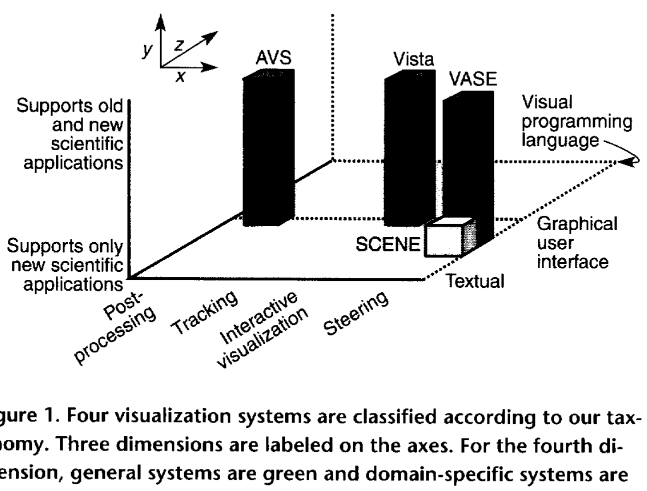


Figure 1. Four visualization systems are classified according to our taxonomy. Three dimensions are labeled on the axes. For the fourth dimension, general systems are green and domain-specific systems are white.

The z axis measures not *what* a user can do with a system, but rather *how* that user does it—textually or visually. If the user communicates with the system via a textual command language or a textual programming language, we classify the system as textual. At the opposite end of this axis, *visual programming languages* provide as much expressive power as we would expect from traditional programming languages, including conditional execution and repetition. However, unlike textual languages, their syntax incorporates graphics or spatial relationships. For example, referencing may be done by pointing with a mouse instead of by naming, and data may be passed using arcs connecting two functions. Examples of visual programming language approaches include

- ♦ programming languages whose syntax is dataflow diagrams,
- ♦ programming-by-demonstration languages in which program logic is demonstrated by manipulating data on screen, and
- ♦ form-based languages in which the spatial relationships of elements on a form help determine program semantics.

Between textual interfaces at one end and visual programming languages at the other lies the *graphical user interface* approach. The user manipulates visual representations to communicate with the system, but the interface lacks the power of a programming language. This category includes everything from pop-up menus and simple icons to graphical toolkit approaches.

The third dimension, shown on the *y* axis, classifies each system as to whether it applies only to new scientific applications written using that particular system, or whether it functions equally as well with existing scientific applications written in traditional textual languages such as Fortran or C. Systems that support existing applications can exploit the fact that huge libraries of such software exist.

The fourth dimension, indicated by color, represents the generality of each system. To achieve ease of use some researchers restrict the problem domain, building in special-purpose tools for common needs in that domain, while others choose not to add such restrictions.

From postprocessing to steering using visual tools

Although much scientific visualization work has been oriented toward textual languages and tools, many recent approaches have started using visual approaches such as visual toolkits and environments. Although these visual approaches are not visual programming languages, they form the roots of a trend to provide more power and flexibility to the researcher using increasingly accessible and easy-to-use means.

AVS

The commercial system AVS (Application Visualization System)[3] represents a class of systems that use graphical tools for scientific visualization, primarily in the area of postprocessing. IBM's Data Explorer,[4] FAST (the Flow Analysis Software Toolkit),[5] and apE[6] are other examples. This type of system provides an environment in which visualizations can be created by combining previously written scientific applications and built-in software modules.

To date, the dataflow paradigm has been the most widespread visual approach to scientific visualization. In this paradigm, streams of data flow like fluids through a network of nodes, each of which performs a computation consuming the data flowing into the node, and producing new data that flow out of the node. The researcher specifies only the flow of data; the order of evaluation is implied by the routing of data through nodes, and thus can be automatically scheduled by the system.

AVS uses the dataflow paradigm to combine software components (see the sidebar on the next page). These components are either built in, or written by the user in C or Fortran with calls to the appropriate AVS routines inserted. AVS handles the scientific visualization process in two distinct cycles: the computational cycle produces a set of raw data, and the analysis cycle produces the visualization. AVS provides a graphical, interactive approach to aid in the analysis phase. This separation of computation and visualization leads to AVS's strong support for postprocessing.

It is also possible to use AVS for tracking, interactive visualization, and even a form of steering. However, doing so often requires traditional programming: using a traditional text editor to modify or create new C or Fortran modules, compiling them, and then adding instances of them to an executing dataflow graph. This is because AVS is not itself a programming language, but rather an environment for combining modules written in other programming languages. AVS has an extensive library of predefined modules, and new ones can be written in C or Fortran and then added to the library. Predefined and user-defined modules are represented by icons that can be selected from a menu. Instances of these modules can be connected as nodes in a dataflow graph; the arcs between the nodes represent the flow of data. Each module can also have variable settings that allow the user to control certain aspects of its operation, such as color and scale.

It is possible to break the dataflow graph during execution, insert instances of different modules, and then resume program execution. However, this is strictly at the module level. To change the algorithms within the modules, the traditional style of textual programming in C or Fortran is required.

Vista

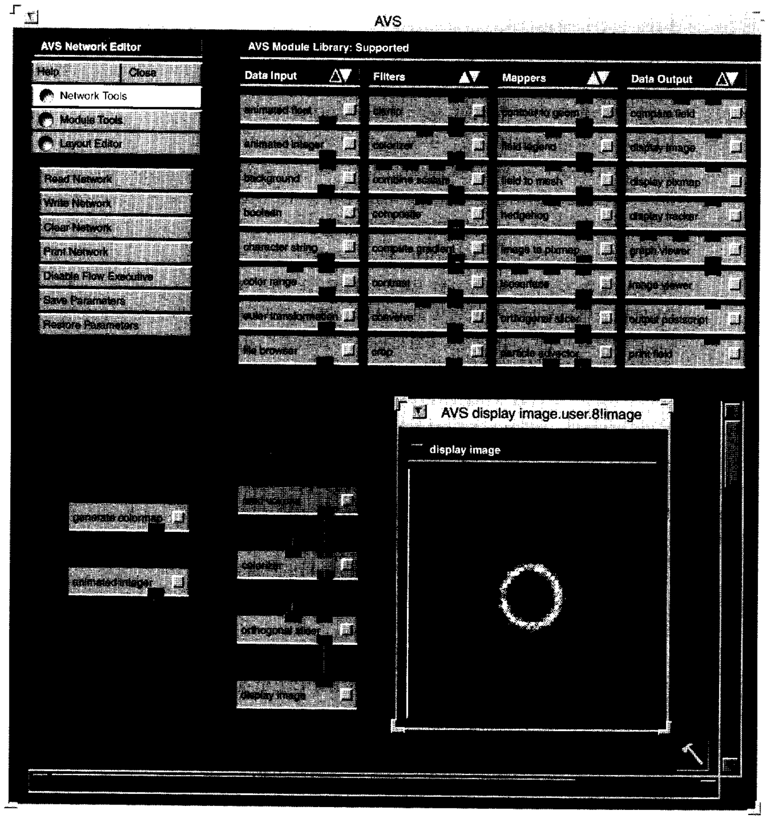
Vista[7] is a system for tracking and interactive visualization, developed at the Center for Supercomputing Research and Development of the University of Illinois at Urbana-Champaign. Displayed data are dynamically updated on screen, and their appearance can be interactively modified via a graphical user interface during execution. This interactive, visual control of the visualization during execution and its easy access to graphics capabilities are Vista's main contributions. Like AVS, Vista can be

Using AVS

AVS's main contribution is the interactive, graphical way in which it enables visualizations to be created and modified. The figure below shows the AVS network editor and graphical user interface, as well as a typical dataflow graph placed in the editor's workspace. At the top of the editor is the AVS module library. To construct a visualization, you simply drag the desired module icons from the library menus to the workspace. Each icon contains several small, colored blocks on its upper and lower edges representing the module's input and output ports, respectively. The ports are color-coded to show the types of data associated with them. You connect nodes by first clicking on the first node's output port and then moving the mouse to the desired input port of the second node. The system will not allow an output port of one type to be connected to an input port of another. Each icon in the library has a button that opens the control setting interface for that module, so you can control the module's operation with built-in graphical tools such as dials and file browsers.

Execution begins as soon as the dataflow graph is complete, that is, when it contains both a data source (a module that reads data from some external source) and a data destination (a module that generates output to the screen or a file). Whenever a module executes, its icon is highlighted, as is each arc whenever data pass through it. This continuously orients you as to exactly where execution is occurring. At any time during execution, you can remove or add modules to the dataflow graph or change any module's control settings.

This example shows AVS going beyond postprocessing to tracking, with the help of prior programming in C or Fortran. The built-in Animated Integer module has been programmed to provide continuous output; its control panel lets you click on desired options, such as stepping through this output or running continuously. In the continuous mode, data move continuously out of the Animated Integer module, and are automatically propagated through the entire graph, since any module can act on data as soon as the data arrive, resulting in a continuously updated view of the graphical representation. However, if the application generating data has not been previously programmed to provide continuous output, only postprocessing is possible.



AVS's Read Volume module reads in a set of volumetric data. The Generate Colormap module then creates a user-adjustable color map for the Colorizer module, which allows the volumetric data to be converted into color values. The Orthogonal Slicer module takes a volumetric slice of the data and sends it to the Display Image module, which shows the image in a window. Finally, the Animated Integer module can provide a set of integers, one at a time, to the Orthogonal Slicer to determine which slice of the volumetric data should be shown. Each module can also have additional user-adjustable control settings.

used to visualize existing scientific applications, since little modification is needed.

Vista is divided into three logical parts: the *application executive* controls the scientific application, the *visualization manager* controls the visualization of data, and the *data manager* handles data synchronization and exchange via asynchronous message passing. The application executive executes the application under the constraints and requests of the visualization manager. The application executive keeps a symbol table of all the variables, which gives it access to the internals of the application. The application can be executed on a remote scientific computer or on the same machine as the visualization manager.

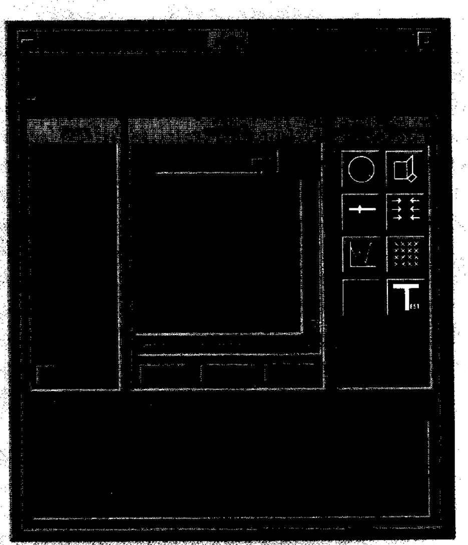
The researcher annotates the application with *visualization breakpoints*, points at which the data are fully consistent and valid, using calls to application executive routines. This is necessary because during execution related data are often unsynchronized or even temporarily corrupt. For example, when a list of numbers is being sorted, one value sometimes overwrites another, and the second has not yet been replaced. If the list were displayed at this point, it would be wrong; one number would be duplicated and another would be missing.

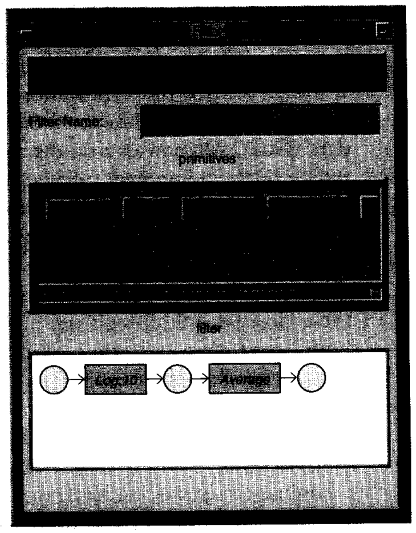
Using Vista

First you textually edit the scientific application to insert visualization breakpoints, and then you compile and link it with the Vista library. To start a visualization session, you type "vista," which launches the visualization manager (see the figure on the left below). When you begin executing the scientific application in another window, which may or may not be on a remote machine, the application executive starts up and the two processes automatically establish communications. This causes automatic update of the displayed data whenever the application encounters a visualization breakpoint.

Across the top of the window are the push-button controls available. In the figure on the left, Halt at Breakpoint has been depressed, causing the application executive to pause at every visualization point in the application. Pressing the Proceed button resumes execution. The bottom window can show messages regarding the status of the application executive and its communications with the visualization manager.

The middle sections of the window contain the heart of the graphical visualization process. The center rectangle contains the names of symbols (variables) that are visible to the visualization manager. You make a variable visible by pressing the Add button below the symbols list, or by optional annotations previously placed in the application. You may display these variables at any time with an appropriate built-in display method (eight methods, including dials, line plots, and tables, are represented by icons on the right side of the window) by clicking on the variable and then on a method. Or, you can first transform the data either by using the provided filters (listed on the left) or by creating new filters via simple dataflow graphs chaining built-in primitives together, as shown in the figure on the right. If you associate the variable with the filter (by pointing), the display method associated with that variable uses the filtered data rather than the original data.


The main window of Vista's Visualization Manager.



Constructing a data filter in Vista.

For distributed scientific visualization systems like Vista that are interactive while still supporting existing scientific applications, approaches to communications become an issue: the greater the degree of interactivity supported, the finer the granularity requiring communications support. Yet this fine granularity needs to be achieved with minimal modification to the existing application. Physically, the data manager in Vista is actually two modules—one on the supercomputer, and another handling communications on the graphics workstation. The data manager on the workstation accepts connections from the host(s), sends messages as needed to the application, and receives data from the application. The data manager on the supercomputer keeps a list of things to do at each visualization breakpoint, and stores prioritized new messages that have arrived since the last breakpoint, interrupting the application executive as needed to deal with these messages.

The visualization manager consists of a graphical user interface as well as various prepackaged data conversion routines, widgets, and user-selected display methods. The researcher's manipulations cause the visualization manager and the application executive to exchange such internal messages as "quit," "send value (for some particular variable)," "send value continuously (for some particular variable)," "stop for awhile," and so on.

VASE

The Visualization and Application Steering Environment,[8] or VASE, also from CSRD, is a collection of tools and software to support interactive visualization and steering in a distributed environment (see the sidebar on the next page). In VASE, not only can you as the researcher control the appearance of data during execution, you can also interactively control data values, add new data to the application, and interactively define new calculations. There are graphical methods for parts of this task, and strictly textual methods for others.

The user model in VASE assumes there are three classes of people who might work with these processes (although one individual might fulfill more than one role), and different kinds of tools are needed by each:

- The *application developer* writes the scientific application using a standard programming language such as C or Fortran.
- The *configurer* configures the different processes over the network and specifies the communications between them. The configurer is not familiar with the source code at the lowest level of detail, but has a high-level understanding of all the functions performed and computational resource requirements of all the processes.
- The *end user* (a researcher) uses and steers these processes.

VASE's *steerable locations* are a generalization of the visualization breakpoints introduced in Vista. After writing and debugging the program, the application developer adds textual annotations (nonexecutable Start and Stop directives) to the sections of the program(s) that might interest the researcher, specifying which data objects may be accessed and modified at runtime at each breakpoint. A graph builder tool uses this annotated source code to produce a collection of *control flow graphs* (such as the two shown in the figure on the right). Each node represents a logical group of statements defined by the application developer; the arcs represent the places where steering can occur. The resulting hierarchical graph serves as an abstraction of the program with which configurers and researchers can work, eliminating the need for them to deal directly with source code. The system then produces a modified version of the source code, ready for compilation into a steerable executable file. The modified source code's structure allows full use of optimizing compilers.

The configurer then assigns the application and visualization processes to computers and establishes communications among them using a graphical user interface. (Most distributed scientific visualization systems use one computer for the application and a second for visualization. VASE is more general, allowing any configuration of computers.) The configurer models the assignment of processes to computers, and the asynchronous communications among them, in a dataflow graph superimposed over the control graph structures (again, see the figure). Each node of the dataflow graph is the entire control graph of a process, and the arcs show the flow of communications. Although neither the control graph nor the location of the breakpoints can be changed at runtime, it is possible to modify the configuration interactively at any time before or during execution. Examples include adding or removing processes to or from the configuration, and adding, changing, or deleting communications between processes.

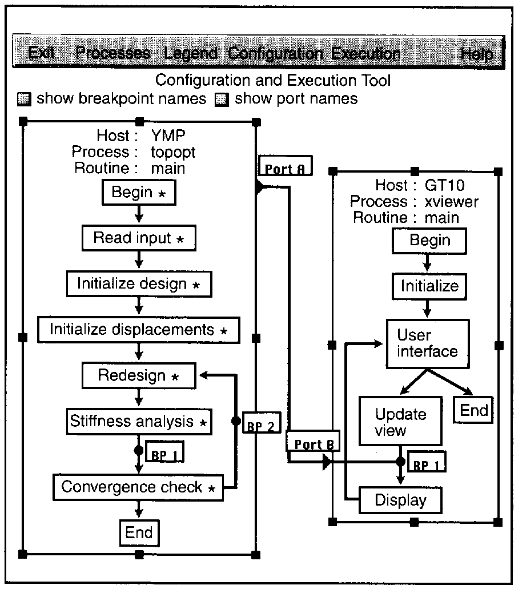
Consider the control-flow abstraction versus the dataflow abstraction in this system. The control-flow abstraction is used to model the functions of the underlying application program

Using VASE for Interactive Visualization and Steering

In this example, the scientific application (called Topopt) is supposed to find a density function describing the distribution of a fixed amount of material that generates the stiffest possible structure under the conditions specified by the parameters. The figure below shows the control graphs for Topopt and Xviewer, the visualization program. Xviewer uses a series of polygon data sets to produce animated displays, and provides interactive rotation and zooming capabilities through its user interface. The application developer has already inserted the breakpoints shown. The configurer has assigned these processes to two computers in the network, and has set up communications such that whenever Topopt writes anything to output Port A, Xviewer can read that data at BP1 via input Port B.

Your tasks here are to cause the desired data to be written to Port A so that it can be tracked, and to perform any steering operations desired as the experiment progresses. To perform the first function, you create a textual script to be executed whenever breakpoint BP1 is encountered in Topopt. The script is written in a subset of C, and maps density to color, producing a two-dimensional display of the structure via calls to a library of visualization functions. You can modify the script at any point during execution to guide the visualization or change the mapping of data to a graphical representation.

To steer the scientific calculations themselves, you incorporate scripts or ad hoc programming statements at any time during execution, thereby guiding the computations in new directions as the intermediate results begin to appear. For example, at BP2 in Topopt, you can at any time change the values of variables and parameters in calculations, enter new calls to existing routines, or enter new code to be executed.



Here two VASE control flow graphs describe the logic of the two processes involved. A dataflow abstraction of how the processes communicate is also shown: each process is a node with input and/or output ports, and an arc between the nodes describes the flow of communications. (Adapted from Haber et al.[8])

(written in C or Fortran, control-flow–oriented languages). In contrast, the dataflow abstraction is not used to model programs written in another language, but rather to support specification of communications. The use of control-flow graphs is necessary for consistency; it would be unreasonable to ask users to mentally "translate" back and forth between dataflow and control-flow ways of describing a single program. But when no underlying program exists, dataflow graphs support straightforward specification of desired logic (for communications in this case).

At this point you, the researcher, take over. When the system encounters a breakpoint, you may choose a course of action via menu selection. One option is to disable a particular breakpoint, causing it to be completely ignored until it is later reenabled. Or, you might specify that whenever a particular breakpoint is executed the system should pause, allowing imperative textual program statements to be entered ad hoc and executed in immediate mode using the command interpreter, until you tell the system to resume execution. Programming statements do not, however, always have to be entered at the keyboard on an ad hoc basis: you can enter a script of programming statements, specifying automatic execution whenever a particular breakpoint is encountered.

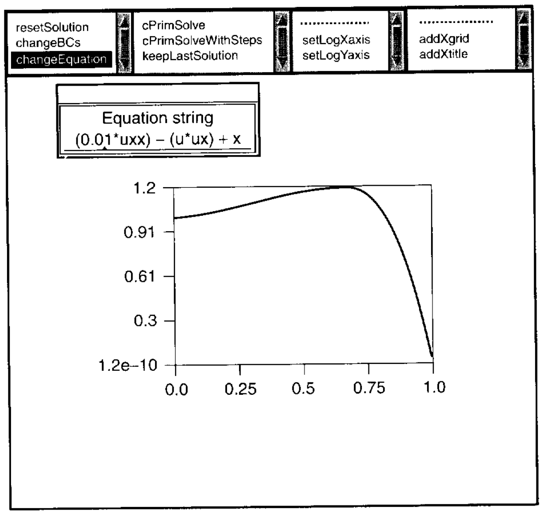
A wide variety of steering actions is possible. For example, in the scientific application the researcher can modify the values of variables, add new variables, call existing routines, or even write and execute new routines. The researcher can also add or change scripts at any point. The only limitations are that the structure or type of an existing variable cannot be redefined, and existing calculations in the original application cannot be modified.

SCENE

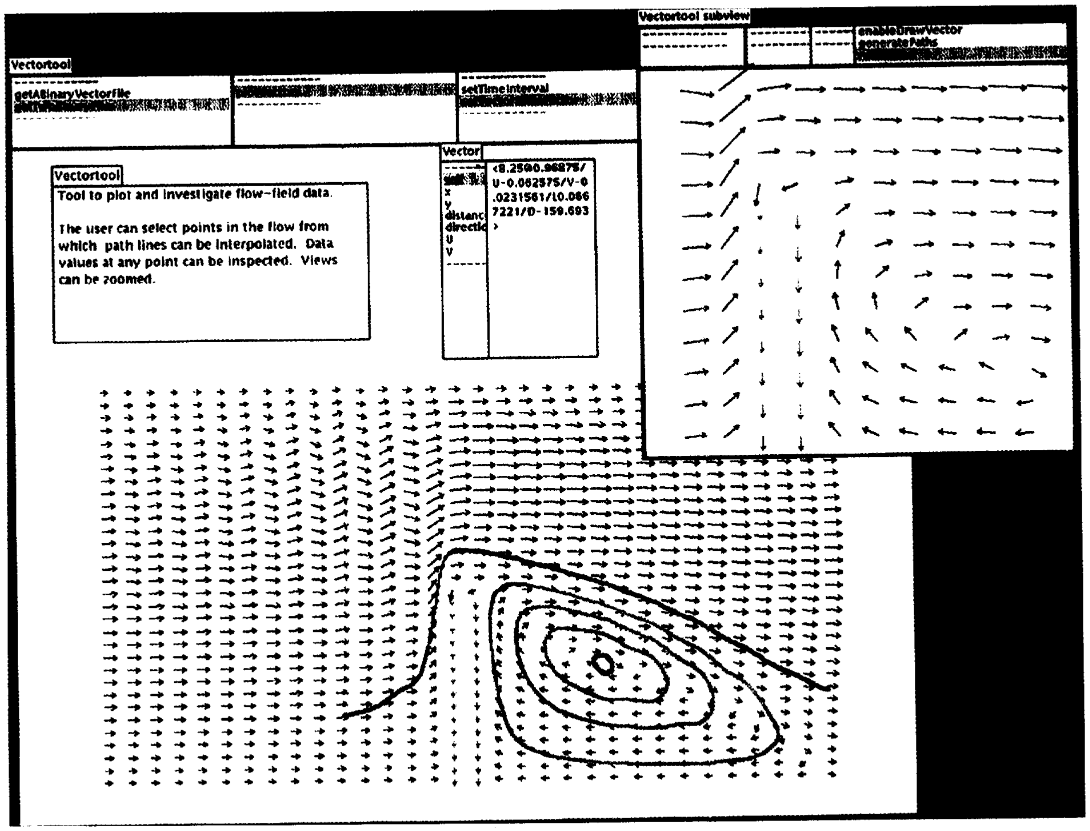
The Scientific Computation Environment for Numerical Experimentation,[9] or SCENE, is an evolving scientific visualization environment for Smalltalk, an object-oriented language and

Steering with SCENE Tools

The SCENE environment includes a number of visual tools. For example, if you enter an equation in the differential equation solver tool, the output is parsed and encoded, and then delegated to a remote computer with algebraic software. You may intervene with the computation at any point where Smalltalk is in control. If any part of the problem is altered (including the equation itself), the result is recomputed and reflected in a solution window.



This figure depicts the output of the equation solver tool. The solutions to the problem were returned from the back end to the Smalltalk interface for graphical display. Here the user has selected "changeEquation" from the menu to experiment further with the equation. (Adapted from Peskin, Walther, and Boubez.[10])



Vectortool

Tool to plot and investigate flow-field data.

The user can select points in the flow from which path lines can be interpolated. Data values at any point can be inspected. Views can be zoomed.

<8.2590.96875/U-0.062575/V-0.0231561/10.0667221/D-159.693>

Here SCENE's vector tool is illustrating vector flow with path interpolation. The user can zoom in for an expanded view of any region or subregion as many times as desired, as shown in the upper right. In the box in the middle of the screen, the original data value corresponding to one of the vectors on the screen has been retrieved.[9] (Copyright 1991 by International Business Machines Corp.; reprinted with permission.)

environment. The advantages of a Smalltalk-based approach are significant, because Smalltalk's features inherently provide a foundation on which complex steering capabilities can be developed.

In Smalltalk, a class defines an abstract data type, and objects are instances of that type. Classes are arranged in a hierarchy, and each class can inherit or override code (methods) defined for the classes above it. Such code is invoked by dynamic message-passing, so termed because it is done entirely at runtime. When an object is sent a message during execution, its class definition is checked to see if a corresponding method definition exists. If not, its ancestors in the hierarchy are searched until an appropriate method is found (or until there are no more to check).

The applicability of dynamic message-passing to steering lies in its ability to support runtime access to the symbol table. Smalltalk capitalizes on this. Since the environment contains an integrated editor, incremental compiler, and runtime system, it is possible to incrementally develop and test code even without the SCENE enhancements. You can define a method and then immediately test it by sending a message to an applicable object. Since code can be changed interactively, breakpoints can be inserted or deleted at will. You can run a simulation for a period of time, interrupt it via the keyboard or a breakpoint to send an object ad hoc messages, make changes and additions to the code, and then continue the simulation without losing the previous state information. This support for many of the characteristics needed for steering is available without the scaffolding needed in procedural approaches.

SCENE is a collection of Smalltalk classes for scientific visualization in fluid dynamics. Some of the classes are implementations of visual tools for steering Smalltalk programs (a few are described in the sidebar on the left). SCENE was designed (1) to demonstrate the applicability of Smalltalk's integrated object-oriented environment to steering scientific simulations, and (2) to add more powerful capabilities for steering computationally intensive applications. The researcher incorporates instances of these classes into an application by programming in Smalltalk. Since some of these classes are visual tools, a number of steering tasks can be performed using graphical mechanisms; others are done textually.

One of the problems that has been addressed experimentally in SCENE is performance. Smalltalk isn't known for speed, and this is true of most other object-oriented languages as well, although there has been recent progress in this area. In SCENE, Smalltalk classes have been implemented on a front-end graphics workstation, with C versions existing on the back-end support systems (such as a supercomputer) to provide high performance. These modules operate cooperatively and transparently between platforms. In early experiments, identical Smalltalk and C versions had to be created manually, but more recently automatic generation of back-end C code during interactive visualization has been reported.

SCENE uses an object-oriented data management approach to support exploration of an object's computational behavior through time. This approach has been prototyped for applications based on collections of numeric data. Using a hashing scheme, data are stored in a database according to their spatial and temporal positions. To make use of these data, a tool that creates a visual representation from data must also be able to compute the inverse, that is, to determine the corresponding data object(s) from a point on the screen. For example, in the box on the left in the bottom figure on page 52, SCENE's vector tool displays detailed information about an object that has been clicked on.

The use of this spatio-temporal approach to data management not only provides access to any object at any time, but also allows objects to be changed incrementally. This is facilitated by the fact that the components of an object are not stored together. Thus you can add a new numeric field to an object that already has several fields by allocating space for it and entering appropriate keys and pointers into the hash table. For example, suppose a large computational fluid dynamics application producing basic velocity-field vectors has been completed. If, upon examining the output, you decide that vorticity information would also be helpful, you describe the new field to the SCENE tool (by naming it and identifying it as a scalar or vector), and enter a mathematical expression defining its contents. SCENE then computes the new values based on the other information already in the database, and stores them without restarting the application; hence, the other components do not have to be recomputed.

Visual programming languages in scientific visualization

As we have already seen, visual approaches are increasingly used as a way of making advanced capabilities more accessible. But in the systems just described, there is a barrier beyond which visual interaction is no longer possible, forcing the scientist to revert to traditional textual programming methods. Visual programming languages strive to remove this barrier, allowing the scientist to perform the entire process—programming, visualization, and steering—interactively and visually. This area of research has as its primary emphasis improving *how* steering is done. Here we present a few representative systems to suggest how visual programming languages might help us achieve flexible steering in scientific applications. Figure 2 presents similarities and differences among these systems.

Visual dataflow programming: VPL and Khoros

VPL[11] is a visual dataflow programming language and environment. It was designed initially for interactive image processing but is not restricted to this domain. It provides general programming capabilities, including conditionals, loops, recursion, and higher-order functions. Because it is a programming language, you can create a complete scientific application in it without having to use one language or tool to create visualization code and another to write the scientific application. VPL does not follow the conventional programming process of editing, debugging, linking, loading, and executing. Rather, you can add, view, modify, or remove program fragments interactively and visually at any time during execution, with results shown immediately and dynamically. This is perhaps the most important contribution that visual programming languages, and VPL in particular, can make in steering: distinctions among such operations as programming, tracking, and steering can be eliminated.

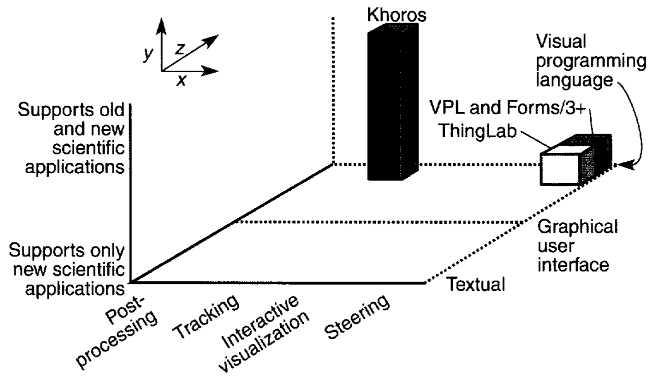

y
z
x
Khoros
Visual programming language
VPL and Forms/3+
ThingLab
Supports old and new scientific applications
Supports only new scientific applications
Graphical user interface
Textual
Post-processing
Tracking
Interactive visualization
Steering

Figure 2. Similarities and differences among four systems. Comparing Figures 1 and 2 shows the chief differences between current scientific visualization and steering research versus visual programming language approaches applicable to visualization and steering. As in Figure 1, the first three dimensions of the taxonomy are labeled on the axes, and for the fourth dimension, general systems are green and domain-specific systems are white.

A VPL dataflow graph is created in much the same way as an AVS dataflow graph, with icons being dragged from a library to a workspace and then connected as nodes in the graph. However, there is a fundamental difference between the two approaches. In AVS, you use the visual environment to manage textually-created program modules; in VPL, manipulating the dataflow graph is the only way to program.

The use of dataflow nodes to perform even the simplest operations is termed *fine-grained dataflow*, an example of which was seen in Vista. In contrast, AVS uses *coarse-grained dataflow*, meaning that each node is defined to be some collection of calculations, which in the case of AVS are defined in C or Fortran. VPL supports both fine-grained and coarse-grained dataflow. The use of fine-grained dataflow provides consistency in the language, since there is only one way to program and it is completely visual. But you would quickly be overwhelmed with low-level details if VPL did not also support coarser granularity. To create a complex node, you connect nodes that have previously been defined (by the system or by you). This collection can then be added as a single node to the toolkit of available nodes and used in the usual way. Because the programming process is the same whether you use the primitive or the complex nodes, you can abstract away any combination of nodes, building abstractions out of other abstractions. This provides procedural abstraction, a necessary feature for any programming language intended to be used in creating sizable programs. The sidebar on the next page shows an example of programming in VPL.

It is possible to program solely by connecting nodes because these nodes are functional operations into which and out of which data flows; there are no hidden states and no possible side-effects in VPL. Execution is automatically sequenced by the system according to the dependencies inherent in the routing of data. To preserve a high degree of interactivity and responsiveness, VPL prioritizes and distributes the computational tasks. The front end, which supports tasks related to user interaction, runs on a color workstation, and the other parts of the system, including the computationally demanding image-processing functions, can run on the workstation or elsewhere on the network.

VPL's use of polymorphism reduces the amount of mechanistic programming needed.

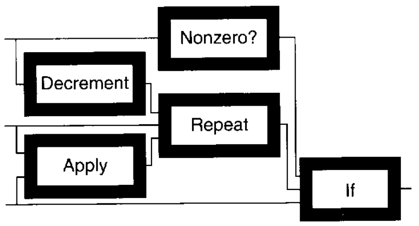
For example, you can program an operation that displays a value with a textual description centered below it without worrying about whether that value is an integer, image, or string. Polymorphic operations like this can be created because a number of the built-in primitives (out of which user-defined operations are defined) are themselves polymorphic. The advantage to this is that you do not have to create numerous similar operations whose only differences are that they work on different types; a potential disadvantage is that the system might not be able to inform you of type-related programming errors. In VPL, however, this problem is largely solved through a type-inference system that automatically tries to infer what types will flow through the dataflow graph, informing you if incompatibilities arise.

Like VPL, the visual programming language used in the Khoros system[12] is based on the dataflow model. They both use nodes to represent routines that have been written in other languages. But unlike VPL, Khoros uses the dataflow metaphor primarily at a coarse-grained level.

Khoros's language-based approach to adding visualization capabilities to externally written applications differs subtly from the toolkit approach represented by AVS, and from the approach represented by VPL, which does not incorporate arbitrary, externally-written code. Using built-in facilities, you convert C or Fortran routines used in other scientific applications into Khoros nodes and then connect them, along with mathematical and image-processing library routines, into a dataflow graph similar to those described earlier. In Khoros, programming is possible at the connection level via programming constructs that can be placed in the dataflow graph like any other node. In this way an existing scientific application, originally written in C or Fortran, can be rebuilt as a Khoros program with the routines brought together with iteration and conditional-execution nodes (see the sidebar on page 56). This ability to combine existing textual code with new code created via a visual programming language distinguishes Khoros from all the other dataflow approaches described in this article.

Programming with VPL

In the figure below, the VPL user is defining a new Repeat operation. (VPL actually has a built-in Repeat operation, but for the purposes of this example, we will pretend it does not exist.) While the user is building the graph using direct manipulation, the workspace is completely active, so the user can try to run sections of the code simply by feeding a source of data into them and triggering execution (by connecting Probe nodes or other display operations). As soon as a name for the new operation has been established it is added to the Components Browser; at this point it can also be incorporated into a program, even if its definition is not yet complete. This is the feature that enables the use of recursion.



The Repeat operation is being defined visually. Note the use of recursion within the definition. (Adapted from Lau-Kee et al.[11])

Visual programming with forms and formulas: Forms/3+

Forms/3+ is a proposal[13] to extend the visual form-based programming language[14] Forms/3 to support interactive visualization and steering in a distributed environment. Forms/3+ supports steering by building on two characteristics of Forms/3. First, you program by entering ordinary mathematical formulas. Second, Forms/3 emphasizes abstraction using visual, interactive mechanisms, thereby helping researchers write complex applications without calling on a professional programmer.

To program in Forms/3, you place cells on forms and provide formulas to define desired calculations. The formulas are entered using a combination of typing and direct manipulation, and are represented by a visual mixture of text and graphics. Forms/3+ identifies some of the formulas as being computationally intensive, automatically designating them for distribution to a remote supercomputer. Forms/3+ extends Forms/3 through this automatic distribution and through other mechanisms aimed at high performance. The system continuously evaluates (solves) formulas as needed to keep the displayed cells up to date. Because the programming process is visual and highly interactive, and because it employs immediate and continuous visual feedback, steering is just the process of temporarily interrupting evaluation in order to add or change formulas.

Formulas in Forms/3 do not have to be strictly numeric; they can also include textual and even graphics operations and primitives. But no matter what type of data is employed in a formula, it is still simply an expression; for example, the formula for a cell C could be A+B, and no side effects are possible. Thus the system can automatically sequence the evaluation of formulas according to their dependencies.

Since every cell has a formula, every cell has a defined value (if no formula has yet been entered, the cell's default is the (constant) formula NoValue). Each formula can produce an answer as soon as it is defined, because the cells to which it refers also have values defined. This continuous evaluation provides an immediate source of error feedback, so you can notice and correct many logic errors as soon as they are entered, rather than discovering and trying to track them down later. Using automatic type inference, an incremental type-checking system immediately checks each new formula entered to decide if it contains a type incompatibility.

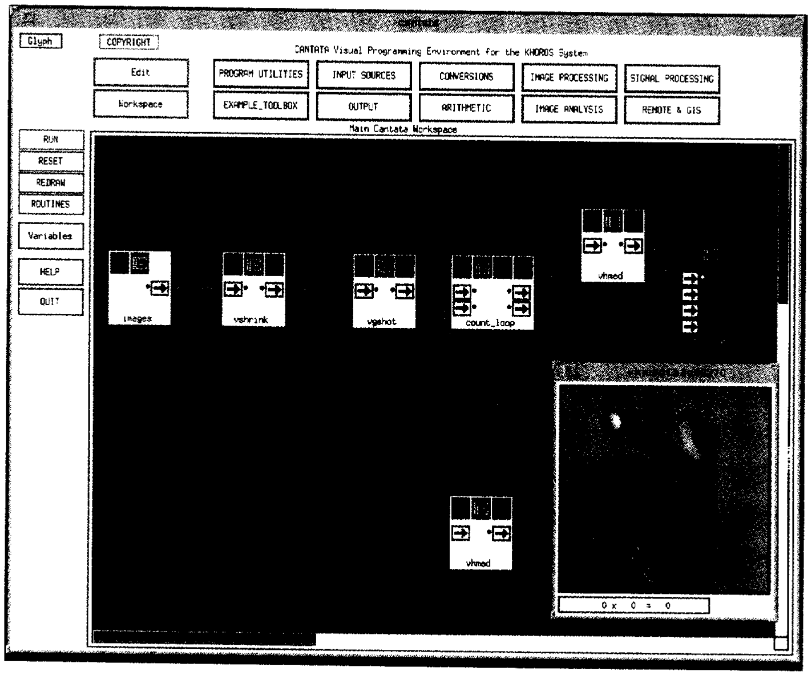
A goal in many visual programming languages is conceptual simplicity. A simple, small language can be more accessible to a researcher than traditional programming languages. The goal is to eliminate concepts without eliminating the expressive power they provide. Forms/3 is an example of the active work in this area.

For example, unlike traditional textual languages, the Forms/3 approach to procedural abstraction does not require a special process of declarations and definitions. Rather, it uses the form as a grouping and encapsulation mechanism. To encapsulate a reusable group of formulas, you place cells with these formulas on a new form and name it something mnemonic such as GroundWaterSimulation (see the sidebar, right). Some of these cells can be hidden through the menu-based formatting mechanisms

Khoros Example

Like other dataflow environments we've described, Khoros uses nodes connected by arcs to represent a program. In Khoros, these nodes represent routines that have either been provided in the Khoros library, or have been previously written in C or Fortran and converted into Khoros nodes.

The application below is averaging 10 noisy images together to form a clean image. The nodes are the routines that act upon the images. Notice the use of count_loop; it is an example of a construct allowing program-controlled repetition. This and the if_then_else (not shown here) are examples of the kinds of programming constructs that make Khoros a visual programming language rather than simply a visual toolkit. Because of such constructs, not all the power of a program has to be located *inside* the modules. This allows textually-programmed modules to be incorporated into a dataflow graph with more flexibility than is true of tool-oriented approaches, and also encourages smaller, reusable modules that can produce the intermediate results necessary for tracking.



The Khoros environment includes a visual programming language known as Cantata, which can be used with library routines or with routines written in other languages. In this example, Cantata is averaging 10 noisy images together to form a clean image.

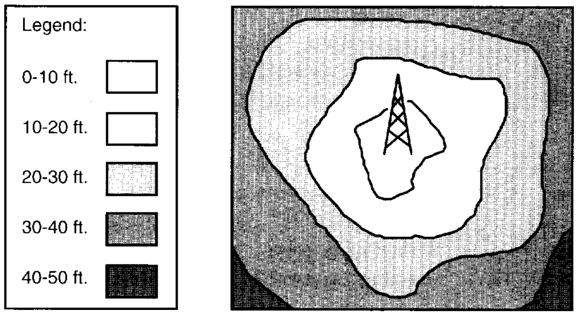
that are used to control a cell's appearance (fonts, colors, and so on). After the cells on the new form have been defined, it is available for use. You can create and modify copies of the form by direct manipulation, or the system can do it automatically. This provides the same functionality as parameterized procedures in traditional programming languages, but does so

Steering with Forms/3+

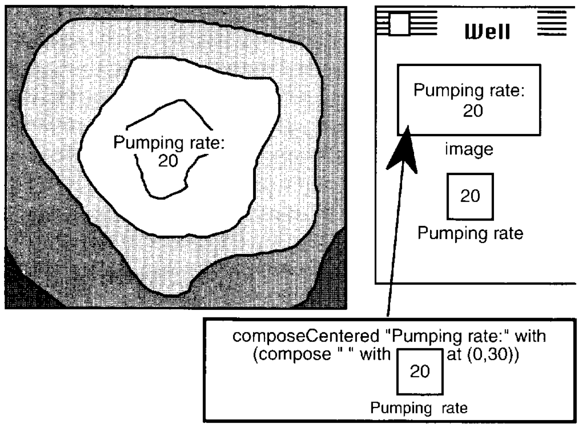
In this example, you (the researcher) are modeling groundwater flow in an unconfined aquifer using numerical methods. Part of this model is executing on a remote supercomputer and part on a local workstation, but you control and steer the simulation as though the program resided entirely on the workstation. The water elevations appear continuously on screen as they are calculated (see the top figure on the right).

If you open up a well form via a menu and click on the displayed well, the form will show details that can be modified at any time during execution (see the middle figure on the right). This is because the well details, like all data, are specified by entering a formula, such as the constant formula "20" for the cell "Pumping rate." The well's graphical appearance on screen is also controlled by a formula that can be changed at any time. Formula changes are immediately reflected in the displayed cells they affect.

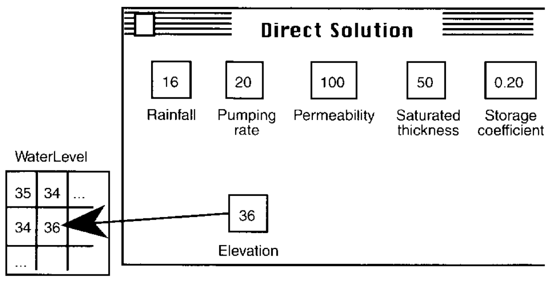
The model itself can also be changed the same way as the visualization—by entering and changing formulas. Such formulas may be numerous and mathematically complex, and for this reason are usually grouped in one or more forms intended to perform a particular group of related calculations. This is an example of the Forms/3 approach to abstraction. In the bottom figure, a form entitled Direct Solution performs the calculations needed to determine the water table elevation, given the parameters shown at the top of the form. The system automatically creates as many copies of the form as are needed to calculate over different sets of parameter values. Many cells on the form are not shown because you have chosen to hide them to avoid unnecessary visual clutter. If desired, you may choose to make the cells visible, thus removing a layer of abstraction to reveal the underlying details.



In this Forms/3+ groundwater simulation, a pumping well affects the water elevation of the surrounding area.


Pumping rate:
20
Well
Pumping rate:
20
image
20
Pumping rate
composeCentered "Pumping rate:" with
(compose " " with 20 at (0,30))
Pumping rate

The researcher can view and change the well's characteristics at any time.



Similarly, the researcher can view the numeric data showing groundwater levels, and steer the model and/or the visualization by entering new or changed formulas or values. The formula for each element of the WaterLevel matrix is a reference to cell Elevation (indicated by the arrow) on the relevant copy of the Direct Solution form, which contains the calculations needed to determine elevation given the parameters shown.

without introducing the concepts of formal parameters, actual parameters, and methods of parameter passing. Data abstraction is supported using the same mechanism: new abstract data types are defined via forms, cells, and formulas.

Forms/3 includes an explicit time dimension. The formula for a cell actually defines a vector of values along a time dimension, rather than an atomic value. The position of the workspace along the time axis determines which value in a cell's time-indexed vector will be displayed, namely the most recent value that occurs on or before the position of the workspace in time. In Forms/3+, this allows you to interactively control the tracking of data values, regardless of which machine they reside on. You can view new values as they appear, and interactively move backward and forward in time to review the evolution of values.

Visual constraint programming: ThingLab

ThingLab[15] is an object-oriented constraint-based visual programming language originally designed for constructing interactive physics simulation environments. (In describing ThingLab, we will also include features contributed by the later ThingLab II,[16] which was designed specifically for constructing user interfaces. While this later system is not intended for simulations, we include it in the description because of its advances in the use of constraints in a visual programming language. For the purposes of this article, we will not distinguish between the two systems.) Using ThingLab, you program and steer simulations by interactively specifying and manipulating constraints on the data. The system's constraint satisfaction mechanism continuously tries to keep all constraints satisfied, which enables changes to be made to a program during execution, with the changes immediately incorporated into new results. ThingLab programs can be run on a single computer or in a distributed environment.

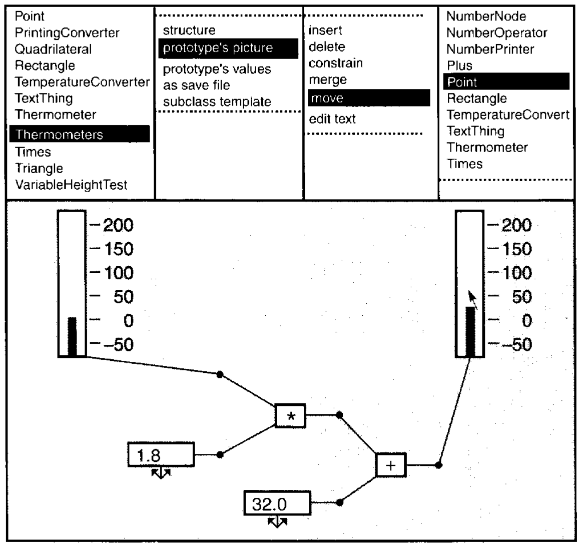
The basic idea behind constraint-based programming is that given a set of constraints (rules) describing the invariant properties and relationships of all values in a particular problem space, the set of solutions is the set of values that simultaneously satisfy all constraints. To illustrate constraint programming, consider the familiar task of unit conversion (see the sidebar to the left). For example, the equation $F = 32 + 9/5 \times C$ converts Celsius temperatures into Fahrenheit units. At first glance, this small constraint program looks the same as a program written in any number of languages. But in most languages, such a program can *only* be used to produce Fahrenheit values from Celsius values. In constraint programming, if C's value is known, it can be used to compute F; likewise, if F's value is known, it can be used to compute C. Thus this program is not a statement of some computation to be performed, but a true equation expressing a relationship between F and C. Furthermore it is entirely equivalent to the program $C = 5/9 \times (F - 32)$.

In constraint languages in general, a program is a collection of constraining equations such as

Using ThingLab

To create a Fahrenheit/Celsius converter, you start the definition by choosing "Times" from the class menu list. A blinking instance of the Times class appears in the display window. You position the instance, as well as the two input and one output connectors attached to the instance. The Plus operator is inserted similarly, and one of its input connectors is attached to the output of the Times instance. The anchors below the constants indicate that the constraint solver cannot change these values in the process of satisfying the constraints. This specification defines a TemperatureConverter class which can be used with a variety of objects representing numeric values.

You continue the session by attaching temperature values (in this example, represented by instances of a Thermometer) to each side of the constraint network. Whenever you modify the value of the Fahrenheit thermometer (using the mouse to move the slider), the constraint solver automatically adjusts the Celsius display to reflect the new value. Also, because of the multi-way nature of the constraints, you can move the Celsius slider up and down and watch the result on the Fahrenheit thermometer.



A temperature converter in ThingLab. (Adapted from Borning.[17])

Table 1. Classification by paradigm.

System	Scientific application paradigm	Visualization or steering paradigm
AVS	Textual imperative (C and Fortran)	Visual dataflow (coarse-grained only)
Vista	Textual imperative (C and Fortran)	Visual dataflow (fine-grained only) Also imperative breakpoints textually inserted into scientific application
VASE	Textual imperative (C and Fortran)	Textual imperative Also visual coarse-grained dataflow to configure communications
SCENE	Textual object-oriented (Smalltalk)	Textual object-oriented
VPL	Visual dataflow	Visual dataflow (fine-grained and coarse-grained)
Khoros	Textual imperative (C and Fortran) Also visual dataflow programming to link imperative code	Visual dataflow (primarily coarse-grained)
Forms/3+	Visual form-based	Visual form-based
ThingLab	Visual constraint-based	Visual constraint-based

the Fahrenheit/Celsius example. It is up to the underlying system to find a solution that satisfies the equations. In general, constraint satisfaction techniques are limited, and can be applied only to specific problem domains. Improving these mechanisms is an active area of research.

In ThingLab, programming is done interactively and visually. The language incorporates many object-oriented features, including inheritance and abstraction. An object consists of its component objects and the constraints among its parts. Objects are defined and instantiated via menu selection and direct manipulation of components and constraints on the screen. Constraints consist of a predicate, which is used to test whether the constraint is satisfied, and methods which tell the system how to alter values so that the constraint might be satisfied. You specify only the predicate, by selecting needed operations and connecting them graphically. From this predicate, the system automatically derives methods that can be used by the constraint satisfier to arrive at the solution. Constraints are used not only for conventional calculations, but also to constrain such matters as the appearance of an object on screen.

For its limited problem domain, in which objects are concrete and can be specified graphically, ThingLab provides interactive, visual support for scientific simulation programming, including steering capabilities. Further, due in part to its object-oriented philosophy, ThingLab offers many features needed for sizable programs. Considerable work has also been done on the speed and flexibility of its constraint satisfaction algorithms, and a compiler provides added speed.

Classification by paradigm

The last two dimensions of our taxonomy classify each system by the programming paradigm it supports for scientific applications, compared with the paradigm it uses for visualization programming (see Table 1). By considering the systems from this view, we can better understand the advantages and disadvantages of the different approaches, because these two dimensions reflect the ways researchers must think in order to specify and interact with scientific computations.

One trend that stands out in the table is that for many of the systems supporting preexisting scientific applications, the researcher is forced to continually switch from an imperative programming paradigm for scientific programming to a declarative paradigm (usually dataflow) for visualizing or steering. Conversely, systems that support a single, consistent paradigm for the entire process do not usually support C or Fortran, and therefore do not support preexisting scientific applications. VASE is the exception to this: it has achieved consistency by working within a single paradigm while still supporting preexisting applications, but so far it has incorporated only primarily textual means to do so.

The most prevalent approach used by visual programming languages for scientific programming as well as for visualization and steering is the dataflow paradigm.[18] Perhaps this is due to its emphasis on computation, and to its naturally visual representation of the relationships among data. This combination seems well suited to the expression of scientific calculations, both ad hoc and permanent, and to the calculations required to convert one group of representations, such as a group of integers, into another, such as a contour map. Approaches based on fine-grained dataflow also carry the advantage that the sequencing of computation is no longer a task of the programmer/researcher; rather, the system can automatically take care of all scheduling based on data dependencies. Automatic scheduling is also possible in coarse-grained dataflow systems that represent modules written in C or other imperative languages, but programmers must exercise more care with this class of systems because imperative languages allow side effects in calculations, which makes the sequence in which each node is executed more of an issue.

Like the dataflow paradigm, the form-based and constraint-based approaches focus on the computations needed to arrive at the desired answers. The form-based paradigm has the advantage that it allows the researcher to write a program by simply providing appropriate mathematical formulas. This feature capitalizes on the fact that experimental scientists and engineers are accustomed to thinking and working in precisely this way. Because changing a formula on screen provides immediate visual feedback, programming and steering resemble experimenting more closely than they resemble traditional programming. This notion of experimenting is also well supported by the constraint-based paradigm when used with physical simulations of multiple objects, all of which may have complex and even circular effects on each other. Like the dataflow paradigm, form-based and constraint-based systems can generally sequence calculations automatically.

Future directions and open problems

Research contributing toward visual steering capabilities is still very young, and many research questions remain.

Scaling up

Despite recent progress,[19] most visual programming languages are still inadequate to the task of large-scale programs such as those needed for scientific applications. To solve this scaling-up problem in the context of steering, it is necessary to devise solutions that do not undermine the characteristics that make visual programming languages conducive to experimentation, such as immediate visual feedback, direct manipulation, and concrete representation. The difficulty is that these characteristics contribute to a proliferation of details and performance penalties that can severely hamper a researcher's productivity in working with a large application.

Metacomputing

In much of the current work in scientific visualization, researchers must be aware of each separate computer's function: which computer is running which program, where each relevant variable is located, and how to converse with each machine involved (such as through Unix commands). Often the researcher must have so much knowledge about programming, operating systems, and networks that the help of a computing professional is required. Smarr and Catlett's notion of *metacomputing*[20] is useful here: Tomorrow's computer users will use only the machine on their desks; if additional resources are needed, the system will automatically reach out across a network to accomplish the task. Most scientific visualization systems are still not addressing this notion of the researcher doing the complete task, without depending on computing professionals for any portion of the task.

Visual programming for steering existing scientific applications

Users are loath to discard existing scientific programs, which can be complex, large, and expensive to recreate. But there are difficult technical problems in devising a visual programming language for steering scientific applications written in another language. Users need ways to easily and clearly communicate with the system using a visual language about changes to logic and data in a program written in another (textual) language. The matter of interprocess communications in general becomes more difficult. Because of this, although several systems support various kinds of visualizations for scientific programs written in C or Fortran, most of these cannot be used for steering.

Yet, despite substantial problems, evidence suggests that doing so is possible. VASE has managed to achieve steering of existing applications, although not through the use of a visual programming language, and visual debuggers already support limited forms of steering textual programs visually. Further, the success of visual systems in postprocessing, tracking, and interactive visualization of existing applications, as well as for steering programs written in visual programming languages, demonstrates their potential in this area. However, extension of these approaches to allow fine-grained additions, changes, and deletions to preexisting applications will be difficult: application code will need to be represented using a visual programming language so that the researcher can not only visualize and steer but also change the program, temporarily or permanently, using that same language. The blurring of distinctions among programming, visualization, and steering—to which visual programming languages have contributed—should be maintained.

Virtual reality

Work on interactive scientific visualization in a virtual reality environment is emerging. In virtual reality environments, a researcher can experience a simulation rather than watch it passively. Klaus Schulten and his colleagues at the Beckman Institute Center for Concurrent Biological Computing have developed a system for simulating molecular dynamics which incorporates virtual reality technology, offering very advanced user interfaces for interacting with the model.[20] Via devices such as head-mounted special glasses and a data glove, the user can interactively control objects in the model, not only seeing the molecule's structure in 3D but also manipulating its molecular structure. Within that system only the objects themselves may be modified; the visualization code, model code, and other supporting objects are not accessible.

Can virtual reality technology be extended to enable full steering of scientific applications? To do this, it is necessary to harness the technology in a way that provides not only a user interface with vivid visualization capabilities, but also true programming capabilities.

Researchers in interactive scientific visualization systems and in visual programming languages have been making contributions toward combining the two areas, but have approached the problem by separate paths. Recent research shows that steering capabilities can eliminate the artificial separation between simulation and data analysis, allowing insights through interactive experimentation that were not possible before. Visual programming languages make this power accessible to the researcher because they do not require traditional programming skills, and because they eliminate the distinctions among programming, visualizing, and steering. By combining emerging research in these two areas, perhaps the prediction of the Panel on Graphics, Image Processing, and Workstations will finally be fulfilled:[1] "Visualization will put the scientist into the computing loop and change the way science is done." ◆

Acknowledgments

We thank Robert Burnett and John Lowther for help with the examples, and Chung-Cheng Luo for contributions to the research behind this article. This material is based on work performed at Michigan Technological University and Oregon State University and is supported in part by NSF under CCR-9215030/CCR-9396134.

References

1. B.H. McCormick, T.A. DeFanti, and M.D. Brown, "Visualization in Scientific Computing," *Computer Graphics*, Vol. 21, No. 6, Nov. 1987, pp. *v–ix*, 1–14, A1–E8.
2. N.C. Shu, "Visual Programming Languages: A Perspective and a Dimensional Analysis," in *Visual Languages*, S.-K. Chang, T. Ichikawa, and P.A. Ligomenides, eds., Plenum, New York, 1986, pp. 11–34.
3. C. Upson et al., "The Application Visualization System: A Computational Environment for Scientific Visualization," *IEEE Computer Graphics & Applications*, Vol. 9, No. 7, July 1989, pp. 30–42.
4. B. Lucas, "An Architecture for a Scientific Visualization System," *Proc. Visualization '92*, IEEE CS Press, Los Alamitos, Calif., 1992, pp. 107–112.
5. P. Walatka et al., "FAST Users' Guide, FAST 1.1a," Tech. Report RND-93-10, NASA Ames Research Center, NAS Division, RND Branch, Moffett Field, Calif., Jan. 1994.
6. D. Dyer, "A Dataflow Toolkit for Visualization," *IEEE Computer Graphics & Applications*, Vol. 10, No. 4, July 1990, pp. 60–69.
7. A. Tuchman, D. Jablonowski, and G. Cybenko, "Runtime Visualization of Program Data," *Proc. Visualization '91*, IEEE CS Press, Los Alamitos, Calif., 1991, pp. 255–261.
8. R. Haber et al., "A Distributed Environment for Run-Time Visualization and Application Steering in Computational Mechanics," CSRD Tech. Report 1235, Center for Supercomputing Research and Development, Univ. of Illinois, Urbana-Champaign, June 1992.
9. R.L. Peskin et al., "Interactive Quantitative Visualization," *IBM J. Research and Development*, Vol. 35, Nos. 1/2, Jan./Mar. 1991, pp. 205–226.
10. R.L. Peskin, S.S. Walther, and T.I. Boubez, "Computational Steering in a Distributed Computer Based User Interface System," in *Artificial Intelligence, Expert Systems and Symbolic Computing*, E.N. Houstis and J.R. Rice, eds., Elsevier Science, North-Holland, Amsterdam, 1992, pp. 104–113.
11. D. Lau-Kee et al., "VPL: An Active, Declarative Visual Programming System," *Proc. 1991 IEEE Workshop on Visual Languages*, IEEE CS Press, Los Alamitos, Calif., 1991, pp. 40–46.
12. J. Rasure and C. Williams, "An Integrated Data Flow Visual Language and Software Development Environment," *J. Visual Languages and Computing*, Vol. 2, No. 3, Sept. 1991, pp. 217–246.
13. A. Ambler, "A Visual-Language-Based Approach to Scientific Visualization," Tech. Report 92-6, Computer Science Dept., Univ. of Kansas, Lawrence, 1992.
14. M. Burnett and A. Ambler, "A Declarative Approach to Event-Handling in Visual Programming Languages," *Proc. 1992 IEEE Workshop on Visual Languages*, IEEE CS Press, Los Alamitos, Calif., 1992, pp. 34–40.
15. A. Borning, "Defining Constraints Graphically," *ACM SIGCHI*, Apr. 1986, pp. 137–143.
16. J. Maloney, A. Borning, and B. Freeman-Benson, "Constraint Technology for User-Interface Construction in ThingLab II," *ACM Sigplan Notices*, Vol. 2, No. 10, Oct. 1989, pp. 381–388.
17. A. Borning, "The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory," *ACM Trans. Programming Languages and Systems*, Vol. 3, No. 4, Oct. 1981, pp. 353–387.
18. C. Williams, J. Rasure, and C. Hansen, "The State of

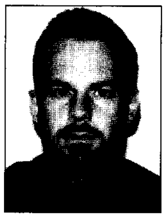
the Art of Visual Languages for Visualization," *Proc. Visualization '92*, IEEE CS Press, Los Alamitos, Calif., 1992, pp. 202–209.
19. M. Burnett et al., "The Scaling-Up Problem for Visual Programming Languages," to be published in *Computer*, Vol. 28, No. 3, Mar. 1995.
20. L. Smarr and C. Catlett, "Metacomputing," *Comm. ACM*, Vol. 35, No. 6, June 1992, pp. 44–52.

Margaret Burnett is an assistant professor at Oregon State University's Department of Computer Science. She has served as guest coeditor of special issues of *Computer* and the *Journal of Visual Languages and Computing*, and is coeditor of *Visual Object-Oriented Programming: Concepts and Environments* (Prentice-Hall, to appear in 1994). Burnett won the National Science Foundation's Young Investigator Award in 1994. Her research focuses on visual programming languages.

She received her MS and PhD in computer science from the University of Kansas and her BA in mathematics from Miami University in Ohio. She is a member of the IEEE Computer Society, IEEE, and ACM.

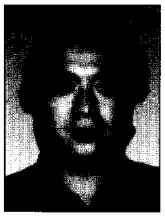
Richard J. Hossli is a computer support specialist in the Civil/Environmental Department of Michigan Technological University. He earned his BS in computer science and his MS in geology at that university.

The authors can be reached in care of Margaret Burnett, Computer Science Dept., Oregon State University, Corvallis, OR 97331; e-mail burnett@cs.orst.edu.



Tim Pulliam is a systems analyst with Ford Motor Company's Powertrain Operations Division. His interests include computer graphics, human-computer interaction, software design methodologies, and graphical user interface design. He received a BS in computer science from Michigan Technological University and is now working toward a MSE in industrial and systems engineering at the University of Michigan.

Brian VanVoorst is a Recom Technologies contractor at the NASA Ames Research Center, where he works for the tools group in the NAS Scientific Computing branch. He contributed to this article during his graduate studies at Michigan Technological University, where he received his BS and MS in computer science. VanVoorst's current interests are in parallel and distributed computing, message-passing algorithms, and performance-tuning tools.



Xiaoyang Yang is a research associate in the Visible Language Workshop at the Media Laboratory, MIT. He received a BS in quantum electronics from Beijing University, and a MS and PhD in computational atomic physics from Michigan Technological University, where he contributed to the research for this article. He is working on the 3D information landscape, a new metaphor that goes beyond the window concept, and dynamic balance in visual communication design.