The terms "visual programming" and "program visualization" only partially capture the spirit of a completely new way for programmers to interact with software and the programs they construct. These terms indicate a new dimension added to programs and software systems, namely the ability to gain different insights and new ways to deal with software through visual and graphical means. Although software can be very complex, visual programming can give us the means to cut through that complexity by providing ways to represent software clearly and concisely in both static and dynamic modes and in two or three dimensions, with color and highlighting.

We proposed this special issue on the potential of visual programming to advance the state of software engineering. We are convinced that visual programming offers exciting opportunity for major changes in the way programmers create software and the ways that users interact with software systems. Moreover, as Georg Raeder...
says in the first article of this issue, by representing programs as semantically suggestive graphical images, we can narrow the gap between mind and medium. The result will be improvements in software productivity and in the ability for users to access and develop software systems.

Why the potential?
People tend to relate easily to graphical representations, and they use them often. They draw pictures of linked lists and trees. Graphical representations are used to represent data flows in highly parallel architectures. Such visual methods offer a more natural way to express algorithms, designs, data structures, and control flows than do linear language-style statements. Indeed, the authors of the articles in this issue uniformly agree that people communicate visually about software and algorithms in both formal and informal ways; they readily use formal charts (of which there are many varieties) and informal sketches and diagrams. The experience at Brown University1 shows that graphical techniques for teaching algorithm analysis to undergraduates are superior to conventional methods. Also, Glinert and Tanimoto2 studied the effectiveness of programming using PICT, their iconic programming system. About 95 percent of the programmers definitely preferred it to conventional methods.

We see three distinct, but related, research areas that comprise visual programming. These are graphics techniques that provide both static and dynamic multidimensional views of software, graphics-based, very high-level programming languages, and animation of algorithms and software to deepen understanding of their behavior.

Software views

It is important to isolate and store the essence of a program or software system. They provide the basis for tailoring different views of the software to the needs of different users. Certainly the views desired by a user, a manager, a novice maintainer, and a system analyst will be very different.

---

**Graphics techniques should have a very high payoff in a software environment that supports the whole software life cycle.**

And they will not be simple black and white pictures. Rather they will consist of sets of pictures, icons, and associated graphics carefully structured and colored so that a user can navigate through them for desired information about the software. Thus, new capabilities made possible by graphics and engineering workstation technology—for example, color and dynamic representations—will be used to clarify a complex mass of information. An example of strictly graphics work that has this aim is the graphics editing system developed by Feiner, Nagy, and vanDam at Brown University.3 Their system uses color, very high resolution graphics, and menu selection to give great flexibility in perusing a document.

Graphics techniques should have a very high payoff in a software environment that supports the whole software life cycle. Requirements, specifications, design decisions, and the finished product would all be captured in graphical form for people who have to use or maintain the system or who have to find out about the system. The PV system reported by Brown, Carling, Herot, Kramlich, and Souza provides the programmer’s chosen methodology. The programmer or user interacts through a menu-oriented interface that displays diagrams and text in multiple windows on the screen. An important feature of the system is the provision for subroutine libraries. Beyond the pioneering research of PV, one can envision a system in which designs and design decisions would be contained in a kind of library, so that people with differing goals, such as maintainers and people doing designs for a similar subject, could have an easy reference to past efforts.

There are a number of projects, not reported on in this issue, that propose visual/graphical techniques to aid us in getting new, comprehensive, and easy-to-use views of software. Some of these are reported on in the survey article by Raeder.

---

**Graphical programming**

Within computer science research, there is a strong movement toward very high level programming languages. The potential of our current programming languages (Fortran, Algol, Ada, etc.) for increasing programmer productivity has nearly been exhausted. A new, more productive approach to software creation must involve languages and means of problem specification that dramatically increase our ability to express requirements to the computer. One approach is to create a language that is much closer to the natural language of the user. A different approach is the creation of graphical languages.

The goal of graphical languages is to give a programmer, or even a user, the ability to perceive an apparently real physical universe and manipulate it naturally through the use of icons and symbols. The general notion of graphical programming is that an arrangement of graphical symbols and icons in two (or possibly more) dimensions represents the program. Since the picture or pictures thus created must be translated into instructions to the computer, there would have to be a precise mapping of the meanings of the symbols into a syntactically correct formal
structure, then into efficient computer code. On the other hand, it may be possible to endow a compiler for such a language with inferential capabilities so that it can “reason” about the symbols, then map the result into machine code. This difficult research area is addressed by two articles in this issue.

The system entitled FORMAL is described by Nan Shu of the IBM Los Angeles Scientific Center. She starts from the observation that forms are a natural interface between a user, data, and a program; however, they are limited. For example, the QBE system devised by Zloof works well within a rather restricted domain of relational databases. Shu extends the forms notion beyond the representation of data objects to a representation of program structure, which resembles the user’s point of view more than that of the computer. She describes a forms-oriented programming language and its implementation. In using FORMAL, one starts with a visual representation of an output form and states requirements within the form’s outline. In this way, a user can program a wide range of data processing applications in a forms-oriented manner.

Rob Jacob considers the difficult problem of providing visual programming languages for abstract objects without a graphical image, such as time sequence, hierarchy, conditional statements, or frame-based knowledge. The powerful principle of “what you see is what you get” is not much help because the objects are abstract. Representing abstract ideas graphically is a powerful form of communication, but there is no theory to guide us. Thus, results require adapting successful methods of related areas and verifying them by experimentation. Jacob has taken the notion of state transition diagrams, which have been widely used by computer scientists (with paper and pencil) to describe algorithms, and applied it to the graphical representation of abstract objects. He has extended the state transition diagram idea to form the basis for a visual programming language, which uses conventional graphical diagrams of the form used to describe finite state automata. He describes an implementation of the language and its use in a programming methodology for designing user interfaces. This novel language has a number of new features, for example, the ability of one diagram to call another.

Seeing dataflow and control structures of algorithms and software as they execute allows software engineers to “feel” software or algorithm action.

Work on this subject includes that of Giacalone, Rinard, and Doeppner at Brown University, where they are preparing an ideographic and interactive program description system. They have invented an ideographic language as the primary means for user-computer communication. It uses an ideographic syntax and supports proofs of equivalence properties. The formalism is based on Milner’s calculus of communicating systems, which was found to be convenient for describing programs. Another visual programming system is that of Glinert and Tanimoto at the University of Washington. Their PICT system is truly graphical: it contains no alphanumeric symbols. PICT programs have a resemblance to flowcharts, but go beyond them through the use of graphics, color, sound and animation, which make programs appear both concrete and multidimensional.

Animation

Certainly the ability to see dataflow and control structures of algorithms and software as they execute will give software engineers and computer scientists the ability to understand and “feel” the action of software or algorithms, as was shown by Brown and Sedgewick of Brown University. Their algorithm simulator and animator, Balsa, has been used for many examples and has led to greater understanding of the algorithms.

A key to this understanding is how the data is represented. For example, in the animation of sorting algorithms, an unordered list of numbers can be represented as a cloud of dots in the Cartesian plane. Each dot is an ordered pair (x,y), where x is a number on the list, and y is the number’s rank in the list. An element in its place has representation (x,y). The viewer can observe how the algorithm visits and changes the cloud of dots on the screen, thus obtaining a new perspective on how the algorithm operates. Moreover, the computer can capture statistics on the performance and present these to the viewer as graphs and charts. Data representations are possible for more complex data structures, such as those found in graph algorithms.

Ralph London and Robert Duisberg report on an animation capability they have created within the Smalltalk programming environment. This environment provides powerful programming and graphics tools upon which to build an animation system. Their approach isolates a graphical viewing structure from the code of the algorithm being animated and separates the user from implementation details. Multiple views of a single object convey an understanding of what an algorithm does, how it works, and why it works. Although their system is experimental, they have run several samples including sorting, an abstract queue or ring buffer, and the Dutch national flag problem.

One should be able to use graphics techniques to test, debug, and main-
tain software. Animation can be particularly effective in guiding the user to bugs and logical inconsistencies in the software. The animation can be full or limited and ought to be coupled with as much automation as possible. There must, however, be an exact, formal correspondence between the icons depicting the software and the software itself. Then, as the software is run, logical inconsistencies within the program can be highlighted. This approach is discussed by Moriconi and Hare, who describe PegaSys, a system that uses graphical images as formal, machine processable documentation. It automatically reasons about logical relationships among pictures representing programs and the programs themselves, thus permitting detection of logical errors in the program.

Visual methods in simulation modeling is the topic of the article by B. Melamed and R. J. T. Morris. In their approach to simulation, the user has command of extensive graphics tools used to draw a model on the screen and observe its behavior through animation and dynamically evolving statistics. The tools are embedded in a performance analysis workstation that uses visual input and output to make the simulation programming process more convenient for system designers than traditional, procedural simulation languages. This graphical simulation tool has been used by about 25 groups within AT&T for a variety of applications, including teaching simulation.

These articles describe the infancy of a new technology that is the creative union of the disciplines of graphics, computer languages, and software engineering. Its maturity will provide us with a very powerful way to produce software systems, control their complexity, and, most important, to understand them.

References


Robert Grafton is a scientific officer in the Computer Science Division of Naval Research, where he manages an international research program in computer science and software engineering. His research interests range from functional differential equations to software engineering. He is the author of several papers in functional differential equations, analysis, and software. He also teaches in the Computer Science Department at George Mason University.

Grafton received a PhD in applied mathematics and a bachelor in engineering, both from Brown University. He subsequently taught applied mathematics at the University of Missouri and the University of Leicester, England. He is a member of the Program Committee for the IEEE International Computer Software and Applications Conference.

Questions about this special issue can be addressed to Grafton at the Office of Naval Research, Code 433, Arlington, VA 22217.

Tadao Ichikawa is a professor of engineering at Hiroshima University. His research interests include database systems, reliable software development environment, and visual programming languages. He co-chaired an IEEE workshop on visual languages held in Hiroshima in December 1984 and is currently serving as general chairman for the second workshop in Dallas, Texas. He is currently an Asian coordinator for the IEEE International Conference on Data Engineering and a member of the Program Committee for the IEEE International Computer Software and Applications Conference. He coedited a book entitled *Special Computer Architectures for Pattern Processing* with King-sun Fu, and published several other books in Japanese. He received a Bachelor of Engineering degree in 1962 and a Doctor of Engineering degree in 1971, both from Waseda University in Tokyo. He is a senior member of the IEEE, and is Associate Editor of the IEEE Transactions on Software Engineering.