Friend-21 Conference
Interaction and Technology

Visual Programming: A Vision for the Future

Henry Lieberman

Visible Language Workshop
Media Laboratory
Massachusetts Institute of Technology
Cambridge, Mass. USA

From graphical interfaces to graphical programming

The advent of high-resolution graphical workstations and direct manipulation interfaces has now led users to expect easy-to-use iconic and pictorial interfaces for a wide variety of applications. But what about programming itself? Despite the advances in computer graphics and interfaces, and despite the increasingly graphical nature of the programming activity itself, programmers still use interfaces that are limited to "glass teletype" textual type-in and print-out.

Visual programming is the idea that pictures can be used as a representation for computer programs, and the activity of programming can be performed by manipulating pictures with an interactive graphical editor. Hopefully, a visual programming system could bring the same kind of productivity leap for programming that real-time display-oriented text editors brought over the previous line-oriented editors, or that desktop-metaphor graphical file systems brought over older command-line shells.

But is this expectation realistic? Is there an effective way to map programming concepts like variables, conditionals, and procedures to a visual display, or is programming an inherently verbal activity? I think visual representations hold great potential for the future, and in what follows, I will discuss objections I commonly hear to the idea of visual programming and try to answer them.

Hasn't visual programming been tried before?

> *We must expect [... from graphical programming] permanent mental damage from most students exposed to it.*
>
> *Edsger Dijkstra, SigCSE 1989*

The idea of using pictures as a representation for computer programs is not new. In fact, the earliest example I have found is the flowchart language GRAIL, dating back to 1966. The fact that experiments in visual programming go back so far in the history of computing has resulted in some pessimism regarding the future prospects of the idea. If graphical programming languages have been around so long, why haven't they succesfully displaced textual programming languages by now? Even some researchers in the field respond to criticism by hedging about the propsects for widespread visual programming, limiting their claims for it only as a teaching tool or in limited domains.

I'd like to convince you that the pessimism surrounding graphical programming is misplaced. Some of these early experiments failed for reasons that are no longer valid, such as lack of sufficient graphics computing power. Others were more successful in demonstrating the feasibility of graphical programming, but did not receive wide publicity and distribution. Many are so different that conservative people have difficulty changing their ingrained habits enough to accept them.

The fundamental problem of visual programming:
Mapping properties of a program onto properties of pictures

Programming is the art of constructing static descriptions of dynamic processes to be performed by the computer. Those static descriptions are traditionally, but need not be, in the form of linear source code in a textual programming language. Maybe the reasons are historical. Computers first had hardcopy text-only printers as output devices, so it was necessary to be able to print the descriptions of programs. Perhaps if graphic displays had been hooked to computers first, we would now think of programs as pictures and it would be hard to imagine how they could be represented only by text!

Let's look at some experiments that have been done in trying to construct visual programming environments. I don't have time to consider them all, or even all the ones I consider good, but a representative selection is illustrated here. They all look very different from each other, but some similarities emerge.

All programming languages, textual or visual, are made of the same kinds of things -- numbers, strings, variables, conditionals, function calls, declarations, and so on. In designing a textual language, the problem is what to call them and how to string them together. In visual languages, the problem is how to make pictures that represent them, and how the visual relations between pictures represent the visual relations between program elements.

In the early days of programing, flowcharts were touted as a conceptual design tool. It was only natural, then, that the first visual programming systems were based on a graphical flowchart representation. Early programming languages, like machine languages and Fortran, had tangled control structures which flowcharts helped unravel. With the rise of "structured" and functional programming, the need for visualizing the linear flow of control lessened, so flowcharts became irrelevant as a visual representation. However, some people still confuse the notion of visual programming with the idea of "programming with flowcharts".

Several more recent visual programming efforts still use a node-and-link visual representation, but these should not be confused with early flowcharts. Examples include Levitt's Hook-Up, Mainstay's VIP for the Macintosh, Apple's Fabrik, and some of my own work. The structure that these diagram is not the flow of control, but the network of function/argument relations. The function/argument relations are much closer to the way in which programmers in modern programming systems conceptualize the structure of their programs, so it is much better to represent this structure in a direct visual manner. Some other systems, such as Smith's Pygmalion or Lakin's VennLisp, also use direct visual representation of function/argument relations, but use containment of icons instead of nodes and links.

Benefits from a pictorial representation of programs can come with ease of learning, especially for beginning users. But even experienced users, after they take the effort to "unlearn" the textual representation, might also experience the benefit of being able to directly perceive relations in the program that would be difficult if it were shown as text. Certainly, the node-link or nested element representations make it easier to perceive scoping relations in the program. They might also, with suitable operations in the graphical editor, make it easier to browse large programs or control level of detail. Pictorial representations might also have benefit in international communication for their independence from a particular natural language, as do the standard international iconic road signs.

Visual representations have already been successful in contexts where either conventional textual representations cannot succeed, or where a simpler domain than full procedural programming is involved. It could be argued that the success of spreadsheets for many programming tasks is due to the visual feedback of representing both programs and data on a square grid, and immediate display of changes. Recent additions of charting functions to spreadsheet programs, and extension of spreadsheet languages move them closer to visual programming, although they stop well short of a true programming environment. Graphical programming environments have also been succesful in teaching programming to young children or beginning users, or in various systems controlled by handicapped users, for whom conventional textual languages would be difficult. The challenge for the future is to extend these successes into more demanding programming domains.

There is now a strong movement to use computer graphics to display simulations of dynamic processes in scientific and engineering fields, under the slogan of *scientific visualization*. The benefits from visual representation of complex data, even when the mapping of data to graphics is very abstract, as it is in most false-color imagery, are quick visual apprehension of complex relations in the results. To the extent that computer science is also a science, visual programming should do for computer scientists what scientific visualization already achieves in other fields of science.

Animated representations of program execution go beyond just graphically representing static programs

What happens when a visual program is executed? Now we broaden our concerns from consideration of just the programming language to the whole programming environment. Some environments merely provide a correspondence between the graphics and a command to execute the program. But the real power of a visual representation should come from making visible the actual steps of execution of a program. Animation should be used to make changing graphics reflect the changing state of the computer. The payoff for this should come when the need arises to debug the program, since debugging is mainly a process of visualizing the operation of a program and matching it to the programmer's expectations.

Again, the issue here in designing a dynamic visual representation for execution of programs is mapping the properties of the execution of programs onto visual properties of pictures. I will show some of my work where I use the idea of graphical substitution of values for the expressions that give rise to them. This is a natural, intuitive model of the process of evaluation. Other systems have used moving cursors to show control flow and moving values to show data flow.

Do graphical program representations take up too much screen space?

A charge that is frequently levelled against graphical representations is that they are too wasteful of screen space, and that textual programming languages are better because you can "see more of the program on the screen at once". But how fair is this charge? A closer look shows that an argument based on information density is not decisive.

First of all, it is necessary to remember that even text itself *is* graphics, so that even textual programming languages are graphical representations of a certain kind. Consider the text of a program in C or Lisp, properly formatted and indented. There is a lot of visual information in the formatting itself. Programmers who do not know Lisp are often put off by its parentheses, but in fact good Lisp programmers typically ignore parentheses and use the visual cues of vertical indentation to understand the program.

If compactness were the only virtue, Lisp or C programs could be made far more compact by deleting excess indentation and all inessential spaces and carriage returns, but the resulting code would be unintelligible to all save the computer. The most compact visual representation possible would be to encode the program using one pixel per bit, which could be learned by people with great difficulty, just as the Morse code was for telegraphy. That we do not use any of these representations says that we are willing, at least in principle, to trade off space for legibility in visual representations, so we should not exclude pictures even if their static density on the screen seems inferior.

Anyway, it is not just the static information density that matters. It is very difficult to comprehend all at once an entire page of source code text, no matter what the programming language. Although text files are oriented toward a linear reading, programmers often need to jump non-linearly from place to place in a program when coding and editing. When programs are larger than one screenful, some means of dynamically moving through them, typically scrolling, multiple files and multiple windows, is employed. These are themselves (relatively unsophisticated) graphic techniques, which impose arbitrary boundaries between succesive temporal views. Pictures can have an advantage in dynamic display, because zoom and pan operations, detail elision, motion and animation can provide browsing techniques that are better than simple text scrolling and file search in locating and displaying dynamically just the relevant information on the screen.

The development of better graphical editors will enable better graphical programming

Currently, textual programming language have a real advantage in working with the programming environments that currently exist. A lot of engineering and optimization work has already been put into text editors that has yet to be performed for graphical editors. Conventional operating systems already have standard file formats and a host of tools for text files that do not yet exist for graphics. As an exercise, think of all the tools that your current favorite system supplies for text manipulation. Is there a graphic equivalent for each of them? Progress is such that the graphic tools will no doubt appear soon, since they are needed for such applications as desktop publishing and CAD if not for programming, but at the moment graphics is at an unfair disadvantage.

As an example, one requirement that is needed for practical control of graphical representations is zooming for control of level of detail. Although many systems do provide a graphics zoom operation, many systems are so sensitive to absolute pixel coordinates in bitmaps, fonts, icons, and other graphics operations as to make general zooming awkward. Window systems made the leap away from dependence on absolute screen position, but we have yet to achieve a similar independence from absolute scale.

It is only recently that workstations have reached enough "graphical horsepower" to make graphical programming feasible

Visual representations are potentially *better* than textual representations because they can provide interactive graphical operations like panning and zooming that would aid navigation through a much larger virtual space than would be feasible for text alone. Hypertext and hypermedia operations allow dynamically switching parts of displays based on user interaction to give selective access to a large information space in a way that simply scrolling through text never could. But these operations are much more computation-intensive than text motion operations, and they critically depend upon the compute power and speed of interactive graphics necessary for updates of large portions of the screen area at rates where motion is perceived as being near-instantaneous.

With hardware graphics accelerators that provide support for very fast block-transfer, line and shape drawing, and other rendering operations, these can now be used as part of the graphics vocabulary for programming. Before that was true, it was always the case that inserting text into a text editor would be faster, so textual representations always had an unfair advantage. We have experimented with a three-dimensional graphical representation for programming, enabled by a display that performs rendering of three-dimensional objects. With such hardware, rotating boxes is takes less time than printing out a line of text.

Conservatism is a strong force in preventing the acceptance of visual programming

Resistance to the idea of visual programming often reminds me of the kind of resistance I encounter to prospects for artificial intelligence. Seymour Papert talks about the "superhuman human" fallacy: People expect an intelligent computer to do work surpassing the best human efforts -- despite the fact that most people themselves couldn't pass this test! Similarly, people may make demands on a graphical language that no textual language could fulfill either. People might complain that visual programming won't work for programs hundreds of pages in length, despite the fact that only a very few programmers can completely understand such large programs in a conventional language!

A visual language for programming is like any other language. It is necessary to invest some effort in learning it before becoming fluent. Because people aren't accustomed to seeing programs displayed as pictures, it is sometimes difficult for them to conceptualize how the pictures relate to the programming task. This has made people a bit resistant to any proposed visual programming representation if they cannot see immediately how it works.

The success of the "desktop metaphor" for manipulating files on a disk has made it part of the computing culture and thus instantly understandable. People sometimes expect visual programming to be just as immediately understandable, despite the fact that programming is a much more complex task than desktop file manipulation. The real test of a visual programming system is if a small amount of training is rewarded with a commensurate increase in power.

A map analogy: Could a textual language replace a visual one?

Any proposal to replace textual programming languages with a graphical language faces a formidable challenge. People already have so much invested in thinking about programs as the source code of a programming language that they find it difficult to conceive of how a visual representation could serve. But the objections posed against a visual representation are often unfair.

Could a visual representation for programs ever overtake a textual representation? Let's turn the question around. Suppose we consider a kind of data for which the representation has traditionally been graphical, and ask if we could substitute a textual representation instead.

The example I wish to consider is representing geographic information in a map. A map is a graphical representation of geographic relations. Maps are highly succesful in enabling people to use geographical information. Some aspects of a map, like the shape of coastlines, are pictorial, but most features of a map are iconic and highly abstracted, such as representations of cities, roads, and topographic features. Maps are a specialized graphical language, and it is necessary to have some experience to be able to read them well.

Now suppose we proposed throwing away the graphical representation of maps in favor of a textual representation. Instead we could represent the geographical relationships by such textual devices as a table of mileages between cities, or lists of stations along train lines or exits along highways. For certain tasks, such as route planning for specific trips, the textual representation might even be more direct. We could even make an argument that the textual representation holds more information. If we print the assertions using the smallest type size used to label cities on a graphical map, we could perhaps squeeze in a higher density of information onto a given surface area of paper.

Yet nobody would agree that we should throw away the graphical representation of maps. The graphics used on maps is highly effective in making geographical relationships directly perceivable, whereas tracing through mileage tables would be tedious. Graphics allows direct perception of different levels of detail in the map; one can focus either on large-scale or small-scale features, whereas a textual representation would display all information at the same size.

The textual tables of geometric relations are analogous to the source code of programs as they are currently represented, in textual form. They display declarative infomation in a linear, visually undifferentiated form. Tracing the control flow of programs is like tracing a route through a mileage table, or making train connections.

Many of the same conceptual issues facing maps also arise in program representations. Visually differentiating data at different scales and of different types is important for level of detail control that prevents the user from being overwhelmed with a large quantity of information. Making proximity and locality relations easily perceivable is important. Hopefully, we will develop visual representations that are as effective for programs as maps are for the visual representation of geographical information.

Synergy with graphic design will be a powerful force for improving visual programming

Recently, I have been working with graphic designers at the Visible Language Worshop of the Media Laboratory at MIT. I have become convinced that a lot can be learned from graphic design which is applicable to the design of visual interfaces for programming.

Graphic design is about the art of arranging visual forms to aid communication, traditionally in print media but now increasingly in electronic media. Graphic designers invent visual languages, which is not so different in some ways from inventing computer languages. Their expertise is therefore valuable in deciding what kinds of visual representations are appropriate for communicating the ideas inherent in programing systems. Graphic designers are now routinely called in by software companies as consultants to critique the appearance of interfaces, but almost always for end-user interfaces and only rarely for interfaces used by programmers.

Visual languages are also important for extending the computational power of programming to users who, like graphic designers, are visual thinkers. As a matter of style, some people can better express their thoughts visually than verbally, and these people are often disenfranchised by text-based computer interfaces. Culturally, computer programming grew up out of the culture of mathematics and engineering, where people tend to be verbal thinkers. Now, applications like desktop publishing, multi-media design, animation, and architecture are drawing more visually-oriented people into the programming world.

Programming environments can be both textual and visual

Finally, we need not make a sharp distinction between textual and graphical programing environments. I believe the most successful programming environments of the future will present, not just a single representation of programs, but many. The computer can take care of automatically converting one representation to another. Some of these representations can be textual, others pictorial, others combining both.

We have already seen how graphically oriented operations are creeping into programming environments. Even though the representation of most programs remains textual, the text is manipulated on the screen by graphical desktops, multiple windows, and icons.

Increasingly, programs are beginning to themselves deal with a graphical environment of icons, windows, fonts, bitmaps, and drawings. The techniques for referencing these objects and operations upon them from a textual programming language -- file names, reference numbers, array data -- are becoming increasingly awkward, so pressure will increase for including direct visual reference and editing for these objects. The recent flurry of "interactive interface editors" such as the Next Interface Builder is the first step in this direction. Thus, visual programming will inevitably be adopted little by little, as the osmosis of such tools makes its way into conventional programming systems.

Seemingly radical ideas can achieve sudden acceptance

It is instructive to follow the history of other seemingly radical ideas in the history of interactive user interfaces. I predict that the acceptance of visual programming will parallel that of another related interface technique that once seemed hopelessly utopian: hypertext. The idea of hypertext dates back to the 1945 article by Vannevar Bush [Bush 45], and was dramatically popularized by Ted Nelson's book Computer Lib [Nelson 74] in the mid seventies. The first succesful commercial product in this area, Apple's Hypercard, broke very little new technical ground, but was well engineered and efficiently implemented, and widely distributed at no cost. I expect the same sort of sudden popularity will occur with a suitably practical visual programming system, without requiring any drastic technological breakthrough. Recent progress in hypertext, and more generally, hypermedia including graphics and sound, is in fact an enabling technology for visual programming. I would be very surprised if the programming environments of the 21st century did not turn out to have a strong visual component.