Journal of Visual Languages and Computing (1990) 1, 97–123

Taxonomies of Visual Programming and Program Visualization*

BRAD A. MYERS†

School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213–3890, U.S.A.

There has been great interest recently in systems that use graphics to aid in the programming, debugging, and understanding of computer systems. The terms 'Visual Programming' and 'Program Visualization' have been applied to these systems. This paper attempts to provide more meaning to these terms by giving precise definitions, and then surveys a number of systems that can be classified as providing Visual Programming or Program Visualization. These systems are organized by classifying them into three different taxonomies.

1. Introduction

IT IS WELL-KNOWN THAT conventional programming languages are difficult to learn and use, requiring skills that many people do not have [1]. However, there are significant advantages to supplying programming capabilities in the user interfaces of a wide variety of programs. For example, the success of spreadsheets can be partially attributed to the ability of users to write programs (as collections of 'formulas').

As the distribution of personal computers grows, the majority of computer users now do not know how to program. They buy computers with packaged software and are not able to modify the software even to make small changes. In order to allow the end-user to reconfigure and modify the system, the software may provide various options, but these often make the system more complex and still may not address the 'users' problems. Easy to use software, such as 'Direct Manipulation' systems [2] actually make the user–programmer gap worse since more people will be able to use the software (since it is easy to use), but the internal program code is now much more complicated (due to the extra code to handle the user interface).

Therefore, we must find ways to make the programming task more accessible to users. One approach to this problem is to investigate the use of graphics as the programming language. This has been called 'Visual Programming' or 'Graphical Programming.' Some Visual Programming systems have successfully demonstrated that nonprogrammers can create fairly complex programs with little training [3].

*The research described in this paper was partially funded by the National Science and Engineering Research Council (NSERC) of Canada while I was at the Computer Systems Research Institute, University of Tronoto, and partially by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976 under contract F33615-87-C-1499 and monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, OH 45433-6543. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency of the US Government.

† Part of the work for this article was performed while the author was at the University of Toronto in Toronto, Ontario, Canada.

1045-926X/90/010097 + 27 $03.00/0

Another class of systems try to make programs more understandable by using graphics to illustrate the programs after they have been created. These are called 'Program Visualization' systems and are usually used during debugging or when teaching students how to program.

This paper, which is updated and revised from references [4] and [5], attempts to provide a more formal definition of these terms, and discusses why graphical techniques are appropriate for use with programming. Then, the various approaches to Visual Programming and Program Visualization are illustrated through a survey of relevant systems. This survey is organized around three taxonomies. Finally, some general problems and areas for further research are addressed.

2. Definitions

2.1. Programming

In this paper, a computer 'program' is defined as 'a set of statements that can be submitted as a unit to a computer system and used to direct the behaviour of that system' [6]. While the ability to compute 'everything' is not required, the system must include the ability to handle variables, conditionals and iteration, at least implicitly.

2.2. Interpretive *vs.* Compiled

Any programming language system may either be 'interpretive' or 'compiled'. A compiled system has a large processing delay before statements can be run while they are converted into a lower-level representation in a batch fashion. An interpretive system allows statements to be executed when they are entered. This characterization is actually more of a continuum rather than a dichotomy since even interpretive languages like Lisp typically require groups of statements (such as an entire procedure) to be specified before they are executed.

2.3. Visual Programming

'Visual Programming' (VP) refers to any system that allows the user to specify a program in a two-(or more)-dimensional fashion. Although this is a very broad definition, conventional textual languages are not considered two dimensional since the compilers or interpreters process them as long, one-dimensional streams. Visual Programming does *not* include systems that use conventional (linear) programming languages to define pictures, such as, Sketchpad [7], CORE, PHIGS, Postscript [8], the Macintosh Toolbox [9], or X-11 Window Manager Toolkit [10]. It also does not include drawing packages like Apple Macintosh MacDraw, since these do not create 'programs' as defined above.

2.4. Program Visualization

'Program Visualization' (PV) is an entirely different concept from Visual Programming. In Visual Programming, the graphics are used to create the program itself, but in Program Visualization, the program is specified in a conventional, textual manner, and the graphics is used to illustrate some aspect of the program or its run-time

execution. Unfortunately, in the past, many Program Visualization systems have been incorrectly labeled Visual Programming (as in [11]). Program Visualization systems can be classified using two axes: whether they illustrate the *code*, *data* or *algorithm* of the program, and whether they are *dynamic* or *static*. 'Data Visualization' systems show pictures of the actual data of the program. Similarly, 'Code Visualization' illustrates the actual program text, by adding graphical marks to it or by converting it to a graphical form (such as a flowchart). Systems that illustrate the 'algorithm' use graphics of show *abstractly* how the program operates. This is different from data and code visualization, since with algorithm visualization the pictures may not correspond directly to data in the program and changes in the pictures might not correspond to specific pieces of the code. For example, an algorithm animation of a sort routine might show the data as lines of different heights, and swaps of two items might be shown as a smooth animation of the lines moving. The 'swap' operation may not be explicitly in the code, however.

'Dynamic' visualizations refers to systems that can show an animation of the program running, whereas 'static' systems are limited to snapshots of the program at certain points.

If a program created using Visual Programming is to be displayed or debugged, clearly this should be done in a graphical manner, which might be considered a form of Program Visualization. However, it is more accurate to use the term Visual Programming for systems that allow the program to be *created* using graphics, and Program Visualization for systems that use graphics *only* for illustrating programs after they have been created.

2.5. Visual Languages

'Visual Languages' refer to all systems that uses graphics, including Visual Programming and Program Visualization systems. Although all these terms are somewhat similar and confusing, it is important to have different names for the different kinds of systems, and these are the names that are conventionally used in the literature.

2.6. Example-Based Programming

A number of Visual Programming systems also use 'Example-Based Programming'. Example-Based Programming refers to systems that allow the programmer to use examples of input and output data during the programming process. There are two types of Example-Based Programming: 'Programming by Example' and 'Programming With Example'. Programming by Example refers to systems that try to guess or *infer* the program from examples of input and output or sample traces of execution. This is often called 'automatic programming' and has generally been an area of Artificial Intelligence research. Programming With Example systems, however, require the programmer to specify everything about the program (there is no inferencing involved), but the programmer can work out the program on a specific example. The system executes the programmer's commands normally, but remembers them for later reuse. Halbert [3] characterizes Programming With Examples as 'Do What I Did' whereas inferential Programming by Example might be 'Do What I Mean'.

Of course, whenever code is executed in any system, test data must be entered to

run if on. The distinction between normal testing and Example-Based Programming is that in the latter the system requires or encourages the user to provide the examples *before* programming begins, and then applies the program to the examples as it develops.

3. Advantages of Using Graphics

Visual Programming and Program Visualization are very appealing ideas for a number of reasons. The human visual system and human visual information processing are clearly optimized for multi-dimensional data. Computer programs, however, are conventionally presented in a one-dimensional textual form, not utilizing the full power of the brain. Two-dimensional displays for programs, such as flowcharts and even the indenting of block structured programs, have long been known to be helpful aids in program understanding [12]. A number of Program Visualization systems [13–16] have demonstrated that two-dimensional pictorial displays for data structures, such as those drawn by hand on a blackboard, are very helpful. Clarisse [17] claims that graphical programming uses information in a format that is closer to the user's mental representations of problems, and will allow data to be processed in a format closer to the way objects are manipulated in the real world. It seems clear that a more visual style of programming could be easier to understand and generate for humans, especially for nonprogrammers or novice programmers.

Another motivation for using graphics is that it tends to be a higher-level description of the desired actions (often deemphasizing issues of syntax and providing a higher level of abstraction) and may therefore make the programming task easier even for professional programmers. This may be especially true during debugging, where graphics can be used to present much more information about the program state (such as current variables and data structures) than is possible with purely textual displays. Also, some types of complex programs, such as those that use concurrent processes or deal with real-time systems, are difficult to describe with textual languages so graphical specifications may be more appropriate.

The popularity of 'direct manipulation' interfaces [2], where there are items on the computer screen that can be pointed to and operated on using a mouse, also contributes to the desire for Visual Languages. Since many Visual Languages use icons and other graphical objects, editors for these languages usually have a direct manipulation user interface. The user has the impression of more directly constructing a program rather than having to abstractly design it.

Smith [12] discusses at length many psychological motivations for using visual displays for programs and data.

4. Taxonomies of Visual Languages

This paper presents three taxonomies. The first, discussed in Section 5, is for systems that support programming, and classifies them as to whether they use Visual Programming and Example-Based Programming. The second, discussed in Section 6, lists the various ways that Visual Programming systems have represented the program. The third taxonomy, discussed in Section 7, is for Program Visualization systems, and shows whether the systems illustrate the code, data or algorithm of programs.

Of course, a single system may have features that fit into various categories and some systems may be hard to classify, so these taxonomies attempt to characterize the systems by their most prominent features. Also, the systems discussed here are only representative; there are many systems that have not been included (additional systems are described in references [18–22]). Since there are so many visual language systems, it would be impossible to survey them all in a single article, but hopefully the 50 or so discussed here will give the reader an overview of the work that has been done.

5. Taxonomy of Programming Systems

Table 1 shows a taxonomy of some programming systems divided into eight categories using the orthogonal criteria of:

- Visual Programming or not;
- Example-Based Programming or not; and
- Interpretive or Compiled.

5.1. Not EBP, Not VP, Compiled and Interpretive

These are the conventional textual, linear programming languages that are familiar to all programmers, such as Pascal, Fortran, and Ada for compiled and LISP and APL for interpretive.

5.2. Not EBP, VP, Compiled

One of the earliest 'visual' representations for programs was the flowchart. Grail [23] could generate programs directly from computerized flowcharts, but the contents of boxes were ordinary machine language statements. Since then, there have been many flowchart languages. For example, FPL (First Programming Language) is reported to be 'particularly well suited to helping novices learn programming' because it eliminates syntactic errors [36]. Other flowchart languages are IBGE [38] for the Macintosh, and OPAL [46] which allows doctors to enter knowledge about cancer treatments into an expert system (see Figure 1). OPAL handles iterations, conditionals and concurrency in an easy-to-understand manner. The GAL system [33] uses a flowchart-variant called Nassi–Shneiderman flowcharts [67] and is compiled into Pascal.

An early effort that was not based on flowcharts was the AMBIT/G [25] and AMBIT/L [26] graphical languages. They supported symbolic manipulation programming using pictures. Both the programs and data were represented diagramatically as directed graphs, and the programming operated by pattern matching. Fairly complicated algorithms, such as garbage collection, could be described graphically as local transformations on graphs.

A new variant on graphs is called 'HiGraphs' [46], which allows the nodes to contain other nodes, and allows the arrows to split and join (see Figure 2). HiGraphs can also be restricted to certain forms to creat specific visual programming languages. For example, Miro [48] is a HiGraphs language for defining security constraints in operating systems (for determining which users can access which files). Another application is the programming of computer user interfaces in StateMaster [50].

Table 1. Classification of programming systems by whether they are visual or not, whether they have Example-Based Programming or not, and whether they are compiled or interpretive. Starred systems (*) have inferencing (Programming *by* Example), and non-starred Example-Based Programming systems use Programming *with* Example. The systems are listed in approximate chronological order.

V.P. Status	Compiled	Interpretive
(a) *Not Example-Based Programming*:		
Not VP	All Conventional Languages: Pascal, Fortran, etc.	LISP, APL, etc.
VP	Grail [23] AMBIT/G/L [25, 26] Query by Example [27, 28] FORMAL [31] GAL [33] FPL [36] IBGE [38] MOPS-2 [40] OPAL [42] Proc-BLOX [44] HiGraphs [46] Miro [48] StateMaster [50] MPL [51]	Graphical Program Editor [24] Spreadsheets PIGS [29, 30] Pict [32] PROGRAPH [34, 35] State Transition UIMS [37] PLAY [39] Action Graphics [41] Forms [43] VERDI [45] LabVIEW [47] SIL-ICON [49]
(b) *Example-Based Programming*:		
Not VP	1/O pairs* [52]	Tinker [53] Editing by Example* [54]
VP	Traces* [55]	Pygmalion [12] Smallstar [3, 56] Rehersal World [57, 58] Graphical Thinglab [59] Music System [60] HI-VISUAL [61] ALEX* [62] Peridot* [63, 64] InterCONS [65] Fabrik [66]

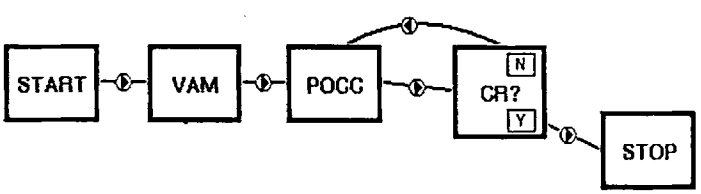


Figure 1. An OPAL program for defining a single cycle of VAM chemotherapy followed by cycles of POCC chemotherapy until the parameter CR (complete response) becomes true [42]

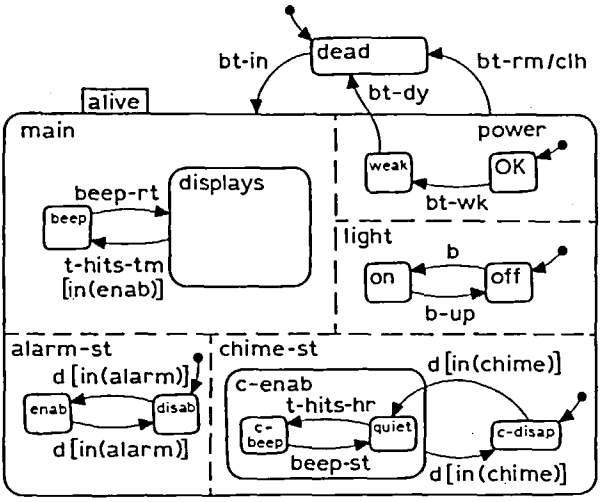


Figure 2. A HiGraphs program describing the operation of a digital watch [46]

You might think that a system called 'Query by Example' would be a 'Programming by Example' system, but in fact, according to this classification, it is not. Query by Example [27] allows users to specify queries on a relational database using two-dimensional tables (or forms), so it is classified as a Visual Programming system. The examples in QBE are what Zloof called variables. They are called examples because the user is supposed to give them names that refer to what the system might fill into that field, but they have no more meaning than variable names in most conventional languages. The ideas in QBE have been extended to mail and other nondatabase areas of office automation in 'Office by Example' (OBE) [28]. A related

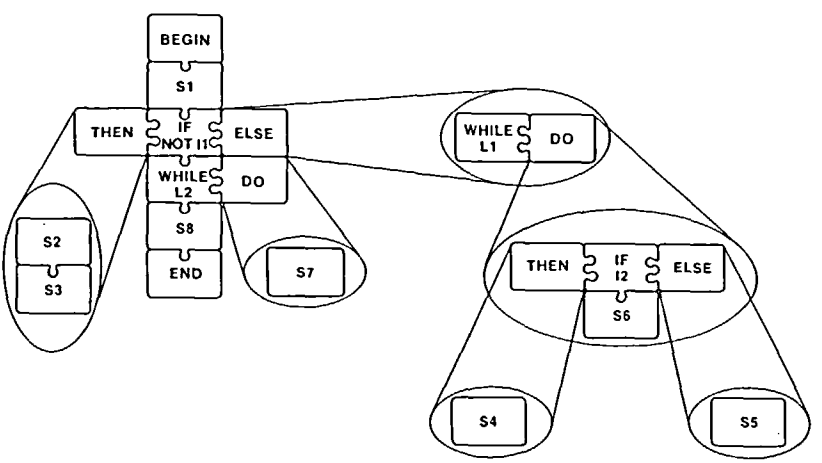


Figure 3. A Proc-BLOX display for some Pascal-like program constructs [44]. The jigsaw puzzle pieces will only fit together in ways that form legal programs

forms-based database language is FORMAL [31] which explicitly represents hierarchical structures.

The MOPS-2 system [40] uses 'coloured Petri nets' to allow parallel systems to be constructed and stimulated in a visual manner. Petri nets may help when programming real-time software, as described in [68]. Berztiss [69] discusses how to lay out Petri nets automatically.

Another interesting way to present program constructs is using tiles that look like jigsaw pieces, and will only fit together in ways that form legal programs. One version of this is Proc-BLOX [44] shown in Figure 3.

The MPL system [51] allows graphical representations of matrices to be combined with conventional Prolog programs. The program is entered with a modified text editor that allows symbolic representations of the matrices to be drawn graphically, and then the resulting file is compiled and run. This is a good example of combining the use of graphics with text.

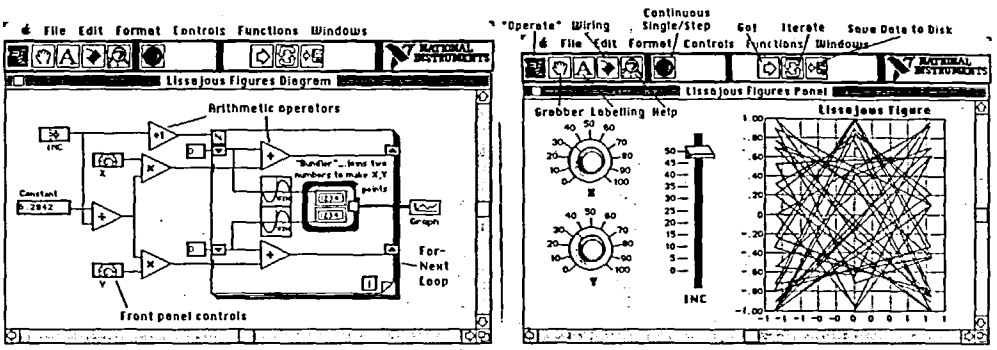
5.3. Not EBP, VP, Interpretive

Probably the first Visual Programming system was William Sutherland's Graphical Program Editor [24] which represented programs somewhat like hardware logic diagrams that could be executed interpretively. Some systems for programming with flowcharts have been interpretive. Pict [32] uses conventional flowcharts, but is differentiated by its use of colour pictures (icons) rather than text inside the flowchart boxes. PIGS [29] uses Nassi–Shneiderman flowcharts and has been extended to handle multi-processing in Pigsty/I-PIGS [30]. Another variant of flowcharts is used by the PLAY system [39], which allows children to create animations by using a 'comic strip' representation of the actions to be performed. The VERDI system [45] uses a form of Petri nets to specify distributed systems. With VERDI, the user can see an animation of the program running by watching tokens move around the network.

A number of visual programming systems use 'dataflow diagrams'. Here, the operations are typically put in boxes, and the data flows along the wires connecting them. One example is PROGRAPH [34], which is a structured, functional language that claims to alleviate the usual problem with functional languages where 'the conventional representation in the form of a linear script makes it almost unreadable' [35]. Another data flow language is Lab-VIEW [47], which is a commercial product running on Apple Macintoshes for controlling external instruments. LabVIEW provides procedural abstraction, control structures, and many useful primitive components such as knobs, switches, graphs, and arithmetic and transcendental functions (see Figure 4).

A number of systems for automatically generating user interfaces for programs (User Interface Management Systems [70]) allow the designer to specify the user interface in a graphical manner. An example of this is the state transition diagram editor by Jacob [37]. Most other UIMSs require that designers specify the programs using some textual representations, so they do not qualify as Visual Programming systems.

Spreadsheets, such as those in VisiCalc or Lotus 1-2-3, were designed to help nonprogrammers manage finances. Spreadsheets incorporate programming features and can be made to do general purpose calculations [71] and therefore qualify as a



(a) (b)

Figure 4. A LabVIEW window (a) in which a program to generate a graph has been entered. The resulting user interface after the program has been hidden is shown in (b)

very-high level Visual Programming language. Some of the reasons that spreadsheets are so popular are (from [43] and [1]):

1. the graphics on the screen use a familiar, concrete, and visible representation which directly maps to the user's natural model of the data,
2. they are nonmodal and interpretive and therefore provide immediate feedback,
3. they supply aggregate and high-level operations,
4. they avoid the notion of variables (all data is visible),
5. the inner world of computation is suppressed,
6. each cell typically has a single value throughout the computation,
7. they are nondeclarative and typeless,
8. consistency is automatically maintained, and
9. the order of evaluation (flow of control) is entirely derived from the declared cell dependencies.

The first point differentiates spreadsheets from many other Visual Programming languages including flowcharts which are graphical representations derived from textual (linear) languages. With spreadsheets, the original representation is graphical and there is no natural textual language.

Action Graphics [41] uses ideas from spreadsheets to try to make it easier to program graphical animations. The 'Forms' system [43] uses more a more conventional spreadsheet format, but adds sub-sheets (to provide procedural abstraction) which can have an unbounded size (to handle arbitrary parameters).

A different style of system is SIL-ICON [49], which allows the user to construct 'iconic sentences' consisting of graphics arranged in a meaningful two-dimensional fashion, as shown in Figure 5. The SIL-ICON interpreter then parses the picture to determine what it means. The interpreter itself is generated from a description of the legal pictures, in the same way that conventional compilers can be generated from BNF descriptions of the grammar.

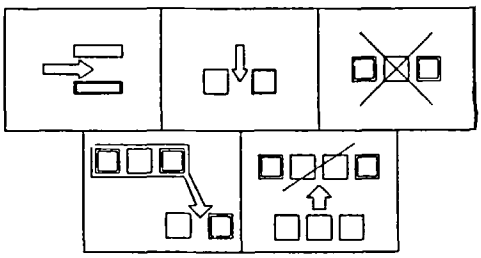


Figure 5. Five different 'iconic sentences' that SIL-ICON can interpret. They mean: insert a line, insert a string, delete a string, move a string to a new place, and replace a string. The user constructs these pictures from primitives such as rectangles, lines and arrows [49]

5.4. EBP, Not VP, Compiled

Some systems have attempted to infer the entire program from one or more examples of what output is produced for a particular input. One program [52] inferred simple recursive LISP programs from a single I/O pair, such as (A B C D)⇒ (D D C C B B A A). This system was limited to simple list processing programs, and it is clear that systems such as this one cannot generate all programs, or even be likely to generate the correct program [72].

5.5. EBP, Not VP, Interpretive

Tinker [53] is a 'pictorial' system that is not classified as VP. The user chooses a concrete example, and the system executes Lisp statements on this example as the code is typed in. Although Tinker uses windows, menus, and other graphics in its user interface, it is not a VP system since the user presents all of the code to the system in the conventional, linear, textual manner. For conditionals, Tinker requires the user to give two examples: one that will travel down each branch. Tinker notices that two contradictory paths have been specified and prompts the user to type in a test of distinguish when each branch is desired.

The Editing by Example (EBE) system [54] is based on ideas from input/output pairs. Here, the system generates a small program that describes a sequence of editing operations. This program can then be run on any piece of text. The system compares two or more examples of the editing operations in order to deduce what are variables and what are constants. The correct programs usually can be generated given only two or three examples, and there are heuristics to generate programs from single examples. EBE creates the programs from the *results* of the editing operations (the input and output), rather than *traces* of the execution, to allow the user more flexibility and the ability to correct small errors (typos) while giving the examples. EBE seems to be relatively successful, chiefly because it limits the domain in which it performs inferencing.

5.6. EBP, VP, Compiled

Some inferencing systems that attempt to cover a wider class of programs than those that can be generated from I/O pairs have required the use to specify the data

structures and algorithms and then run through a computation on a number of examples. The systems attempt to infer where loops and conditionals should go to produce the shortest and most general program that will work for all of the examples. One such system is by Bauer [55], which also decides which values in the program should be constants and which should be variables. It is visual since the user can specify the program execution using graphical traces. Unfortunately, these systems tended to create incorrect programs, and it was difficult to check what the system had done without studying the generated code.

5.7. EBP, VP, Interpretive

Pygmalion [12] was one of the seminal VP and EBP systems. It provides an 'iconic' and 'analogical' method for programming: concrete display images for data and programs, called icons, are manipulated to create programs.[a] The emphasis is on 'doing' pictorially, rather than 'telling'. Thinglab [73, 74] was designed to allow the user to describe and run complex simulations easily. A VP interface to Thinglab is described in [59]. Here the user can define new constraints among objects by specifying them graphically. Also, if a class of objects can be created by combining already existing objects, then it can be programmed by example in Thinglab.

Smallstar [3, 56] uses EBP to allow the end user to program a prototype version of the Star office workstation [75]. When programming, the user simply goes into program mode, performs the operations that are to be remembered, and then leaves program mode. The operations are executed in the actual user interface of the system, which the user already knows. Since the system does not use inferencing, the user must differentiate constants from variables and explicitly add control structures (loops and conditionals). This is done on a textual representation of the program created while the user is giving the example. Halbert reports that Star users were able to create procedures for performing their office tasks with his system.

The goal of Rehersal World [57, 58] is to allow teachers who do not know how to program to create computerized lessons easily. Interactive graphics are heavily used to provide a 'collaborative, evolutionary and exploratory' environment where programming is 'quick, easy and fun'. The metaphor presented to the user is a *theatre*, where the screen is the *stage* and there are predefined *performers* that the user can *direct* to create a *play*. The teacher developing the program sees at every point exactly what the student-user of the play well see. In addition, the teacher can have additional performers in the *wings* (so the student will not see them) that provide auxiliary functions such as flow control. Everything is made visible to the teachers, however, which allows their thinking to be concrete, rather than abstract as in conventional programming environments. When a new performer is needed, often its code can be created using examples, but when this is not possible, some Smalltalk code must be written. The static representation for all performers is Smalltalk code, which can be edited by those who know how.

[a] Pygmalion is also credited with inventing the use of icons in computer interfaces. Icons were later used by Smith and others in commercial products such as the Xerox Star and Apple Lisa and Macintosh.

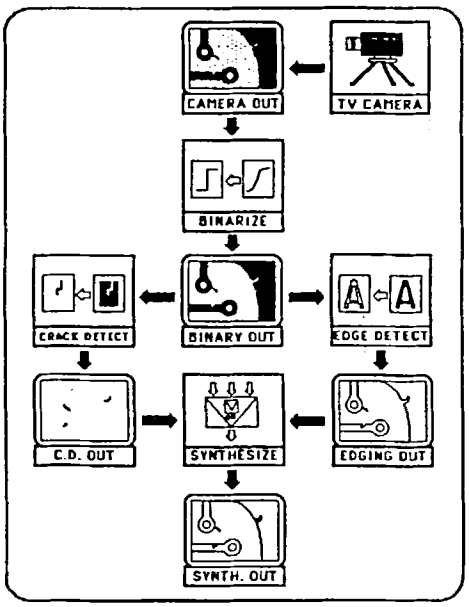


Figure 6. A HI-VISUAL program for performing image processing [61]

HI-VISUAL [61] allows the user to construct data flow programs out of iconic pictures (see Figure 6). It is classified as EBP because the user supplies sample data before programming starts, and the system executes the program on the data as each icon is added to the program.

A related system uses direct manipulation to configure icons and circuit diagrams to define sound processing systems [60]. This system is classified as Programming With Example because the resulting sound is continuously played while the circuit is being constructed.

The ALEX system [62] allows matrix manipulation algorithms to be specified by example. The user points to a typical element, row, or column in graphical presentation of a sample matrix, and then specifies how to process it. The system then generalizes this operation to operate on the entire array.

Peridot [63, 64] is a tool for creating user interfaces by demonstration without programming. The user draws a picture of the desired interface and the system generalizes this picture to produce a parameterized procedure (see Figure 7). The user gives example values for any parameters so the system can display a concrete instance of the user interface. Peridot allows a nonprogrammer to create menus, scroll-bars, buttons, sliders, etc., and it can create most of the interaction techniques in the Apple Macintosh Toolbox.

Two data flow systems support Programming with Example. InterCONS [65] and Fabrik [66] both were developed in Smalltalk and allow the user to wire together low-level primitives like arithmetic operators and higher-level user interface elements like scroll bars and buttons. These systems allow the user to input sample data as the program input, and they continually adjust the output data based on the input and the program constructed thus far. Fabrik also handles undefined values on wires by

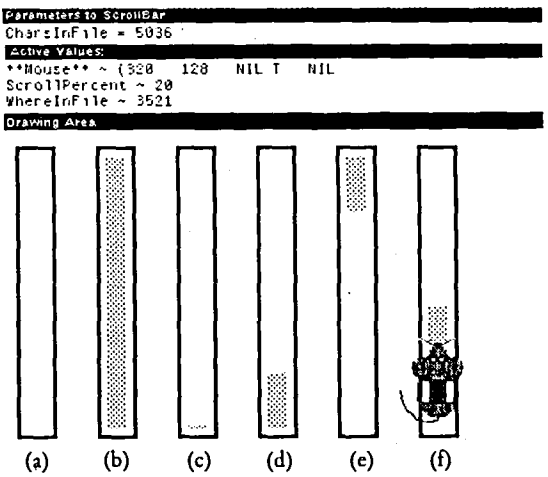


Figure 7. Creating a scroll bar using Peridot. In (a), the background graphics have been created. The grey bar will represent percent of file visible in the window. The two extemes of the full file (b) and none of the file (c) are demonstrated. This will depend on the active value *ScrollPercent* which ranges from 100 to 0 (d). Next, the two extremes of seeing the end of the file (d) and the beginning of the file (e) are demonstrated. The active value *WhereInFile* (which varies from the value of the paramter *CharsInFile* down to one) controls this (f). The designer then demonstrates (f) that the bar should follow the mouse when the middle button is down using the 'simulated mouse' [63]

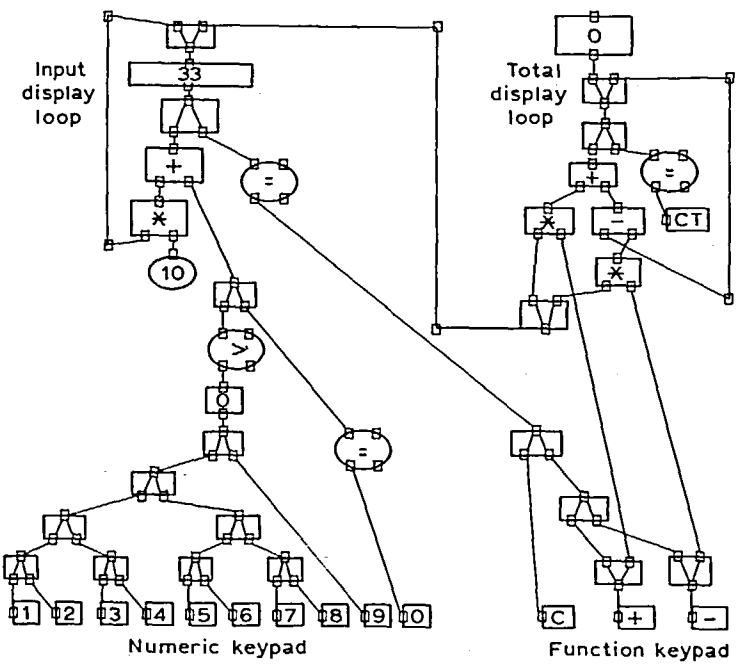


Figure 8. A desk calculator program in InterCONS [65]

drawing them with dotted lines. Figure 8 shows an example of an InterCONS program for a calculator.

6. Classification by Specification Technique

Another way to classify programming systems is by what kind of representation they use for the code. Table 2 lists the systems discussed here by what specification technique they use. As new Visual Programming systems are designed, this list is likely to grow, since new forms for the specification can be invented.

6.1. Discussion

Many of the categories listed in Table 2 should be clear, but some need additional explanation.

The 'Textual Language' specification style is clearly used by all conventional programming languages. It is also used by Tinker since it is not a Visual Programming Language. Smallstar is a example-based-programming system and the system generates the appropriate code while the user is demonstrating the program. Smallstar uses a textual language (augmented with a few decorative icons) to record the user's program. Many of the other example-based-programming systems are listed in the figure as having 'no' textual language. This is because they generate code in a conventional computer language (e.g. Lisp for I/O Pairs and Peridot) which is not shown to the users.

The 'Iconic Sentences' are a separate category because here the positions of the picture are meaningful, and not just how they are connected with arrows as with flowcharts and graphs.

Table 2. Classification of programming systems by specification style. Classifications marked with a star (*) primarily show the data of the program, rather than the code. References for these systems are shown in Table 1.

Specification Technique:	Systems:
Textual Languages	Pascal, Ada, Fortran, Lisp, Ada, etc. Tinker, Smallstar
Flowcharts	Grail, Pict, FPL, IBGE, OPAL
Flowchart derivatives	GAL, PIGS, SchemaCode, PLAY
Petri nets	MOPS-2, VERDI
Data flow graphs	Graphical Program Editor, PROGRAPH, Graphical Thinglab, Music System, HI-VISUAL, LabVIEW, Fabrik, InterCONS
Directed graphs	AMBIT/G/L, State Transition UIMS, Bauer's Traces
Graph derivatives	HiGraphs, Miro, StateMaster
Matrices	ALEX, MPL
Jigsaw puzzle pieces	Proc-BLOX
Forms	Query by Example, FORMAL
Iconic Sentences	SIL-ICON
Spreadsheets*	VisiCalc, Lotus 1-2-3, Action graphics, "Forms"
Demonstrational*	Pygmalion, Rehersal World, Peridot
None*	I/O Pairs, Editing by Example

In 'Demonstrational' systems, the program is defined by graphics that change in time. The meaning and behavior of the icons is demonstrated temporally, and the system remembers what the user has done. For example, in Pygmalion, to demonstrate that 3 should be added to the value in a variable, the user would drag the icon for the variable into one of the input slots of the adder icon, and a '3' to the other input slot. There is no visible representation of the actions.

The systems classified as using Demonstrational, Spreadsheets, and no language ('None') actually show the *data* of the program, rather than the code. The current values of the data is visible on the screen, and the code that caused the data to get to be that way is hidden. Sometimes, but not often, there is a way to discover previous states of the data. This is in contrast to most other systems (including data flow diagrams), where the code of the program is represented and the data is implicit. The AMBIT languages are somewhat unique however, because here both the code and data is shown in a pictorial manner.

7. Taxonomy of Program Visualization Systems

The systems discussed in this section are not *programming* systems since code is created in the conventional manner. Therefore, none of the systems discussed below appears in the previous sections. Graphics is used here to *illustrate* some aspect of the program after it is written. Table 3 shows some Program Visualization systems classified by whether they attempt to illustrate the code, data or algorithm of a program, and whether the displays are static or dynamic. Some systems fit into multiple categories, because they illustrate multiple aspects or have different modes.

7.1. Static Code Visualization

The earliest example of a visualization is undoubtably the flowchart. As early as 1959, there were programs that automatically created graphical flowcharts from Fortran or assembly language programs [76]. An entirely different approach is taken by SEE [77]

Table 3. Classification of Program Visualization Systems by whether they illustrate the code, data or algorithm, and whether they are static or dynamic.

	Static	Dynamic
Code	Flowcharts [76] SEE Visual Compiler [77] PegaSys [79] TPM [82]	BALSA [16] PV Prototype [78] MacGnome [80] Object-Oriented Diagrams [81] TPM [82]
Data	TX2 Display Files [83] Incense [14, 85]	Linked Lists [84] MacGnome [13]
Algorithm	Stills [87]	Two Systems [86] Sorting out Sorting [15] BALSA [16, 88] Animation Kit [89] PV Prototype [78] ALADDIN [90] Animation by Demonstration [91] TANGO [92]

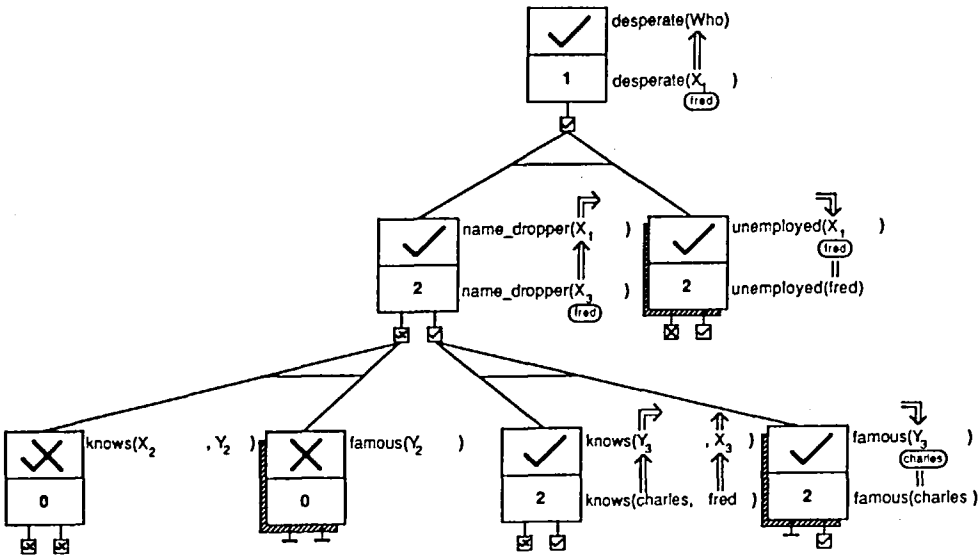


Figure 9. A Prolog program visualized by TPM [82]

which attempts to make conventional program text easier to read by adding multiple fonts, nice formatting, and other graphics.

In PegaSys [79], pictures are formal documentation of programs and are drawn by the user and checked by the system to ensure that they are syntactically meaningful and, to some extent, whether they agree with the program. The program itself, however, must still be entered in a conventional language (Ada).

The Prolog logic-programming language has a quite different execution model than conventional languages. In order to try to make it more understandable, TPM (the Transparent Prolog Machine) [82] generates pictures of the execution of Prolog programs. TPM will produce nicely formatted pictures after a program has completed (so it is classified as 'static'), but it will also show an animation of the code executing on less well-formatted pictures (so it is also listed as 'dynamic'). Figure 9 shows a sample of a TPM picture.

7.2. Dynamic Code Visualization

Most systems in this class do not actually animate the code itself, but rather dynamically show what parts of the code are being executed as the program is run using some sort of highlighting. Examples are BALSA [16], PV Prototype [78] and MacGnome [80]. Figure 10 shows the BALSA highlighting the execution of a recursive procedure.

The Object-Oriented Diagraming system [81] has a somewhat different focus. It is aimed at illuminating the message-passing structure in object-orientated programs. The system displays objects as boxes (see Figure 11), and arrows show whether the message is handled by the object class, or by one of its super-classes.

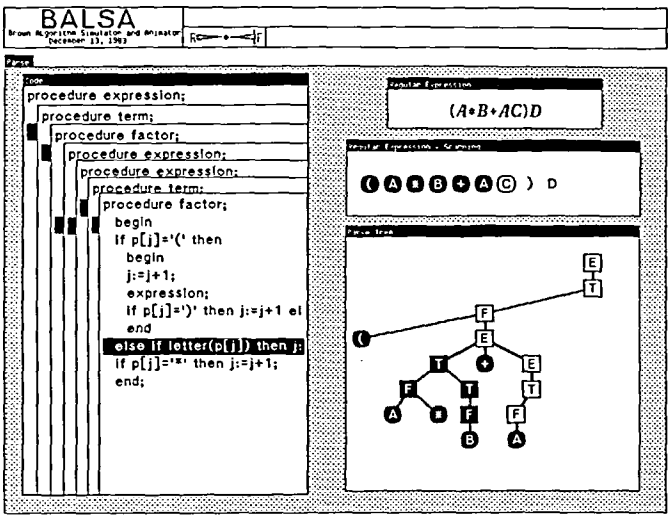


Figure 10. On the left is a code visualization from BALSA showing the highlight bar that follows the execution and the recursive nesting of procedures. On the right is the algorithm animation [16]

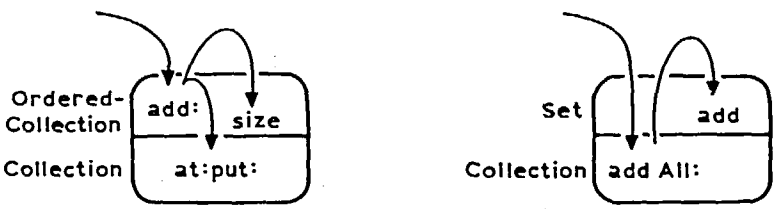


Figure 11. Display of message passing from [81]. Each rounded box is one object instance, and super-classes are shown below sub-classes. The arrows show whether the message was handled by the object class itself (e.g. *add:* which calls *at:put:* of its parent class) or whether it is handled by the super-class (e.g. *addAll:*)

7.3. Static Data Visualization

A very early system for the TX-2 computer could produce static pictures of the display file to aid in debugging [83]. Incense [14, 85] automatically generated static pictorial displays for data structures. The pictures included curved lines with arrowheads for pointers and stacked boxes for arrays and records, as well as user-defined displays (see Figure 12). The goal was to making debugging easier by presenting data structures to programmers in the way that they would draw them by hand on paper.

7.4. Dynamic Data Visualization

One of the earliest data visualization systems was the L6 movie of list manipulations [84]. This system actually falls between dynamic and static since the software created frames that were filmed. The hardware was not fast enough to animate the structures changing. The MacGnome system, however, shows the pictures changing as the data

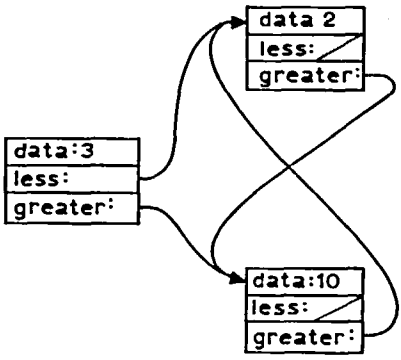


Figure 12. A display produced automatically by Incense of three records containing pointers [85]

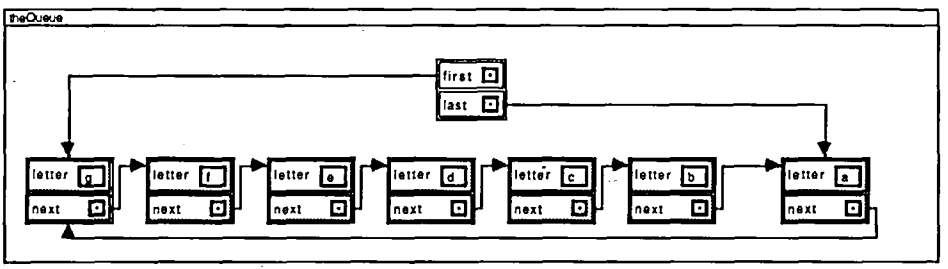


Figure 13. A data visualization automatically produced by MacGnome [13] of a queue of characters implemented as a linked list of records

is modified [13]. It runs on the Macintosh, and is similar to Incense in that it automatically produces displays for data structures from the types of the variables; no extra code is needed to generate the pictures. The user simply points to a variable with the mouse, and a picture of its data is automatically displayed (see Figure 13).

7.5. Static Algorithm Visualization

A visualization system that produces static snapshots of the algorithm is Stills [87]. The user added special commands to the source algorithm, and the system generated troff output which could be sent to printers.

7.6. Dynamic Algorithm Visualization

Most algorithm visualizations systems are dynamic since they produce animations of the algorithm in action. The first few systems in this class, like the early data visualization systems, created movies of the algirhtms (e.g. sorting) and were used for teaching computer science algorithms [86, 15].

Unlike data visualization systems, all algorithm animation systems require that the programmer explicitly add information to the code to control the animations. In the famous BALSA system from Brown University [16], special instructions were added to the code to signal important events. This system was designed to each students

about programming, and produces the illustrations in real time on an Apollo personal workstation (see Figure 10). An updated version, called BALSA-II, runs on the Macintosh and allows the user to control the animation using Macintosh-style menus [88]. The code of the algorithm must still be augmented to tell the system about important events.

The 'PV Prototype' [78] was designed to aid in debugging and program understanding, and it supports dynamic displays of data and easier construction of user-defined displays. Another system, called Animation Kit, has similar goals. It is written in Smalltalk and features smooth transitions from one state to another [89].

A recognized problem with these systems is that it is difficult to specify what the data animations should look like. ALADDIN [90] attempts to alleviate this problem by allowing a declarative specification of the desired views using a catalog of pre-defined graphical and animation primitives. A different approach was used by Duisberg [91] in the Animation by Demonstration system, which allows the desired animations to be specified by demonstration. The user draws a sample picture and then demonstrates an example of the animation to be performed. This animation can then be triggered when a message is sent to an object in the underlying Smalltalk environment. The system uses gestures and a music-like score editor to control the timing of the animations. TANGO [92] uses a similar approach and allows much of the animations to be created using a graphical editor instead of by writing code.

8. Evaluation of Visual Programming and Program Visualization

Although there is a great deal of excitement about Visual Programming and Program Visualization, as well as a large number of working systems, there is still a lot of skepticism about the success and prospects of the field. For example, Frederick Brooks wrote:

> 'A favourite subject for PhD dissertations in software engineering is graphical, or visual, programming—the application of computer graphics to software design.... Nothing even convincing, much less exciting, has yet emerged from such efforts. *I am persuaded that nothing will.* In the first place,... the flowchart is a very poor abstraction of software structure.... It has proved to be useless as a design tool.... Second, the screens of today are too small, in pixels, to show both the scope and the resolution of any seriously detailed software diagram.... More fundamentally,... software is very difficult to visualize. Whether one diagrams control flow, variable-scope nesting, variable cross-references, dataflow, hierarchical data structures, or whatever, one feels only one dimension of the intricately interlocked software elephant.' ([93], pp. 15–16, emphasis added)

In a similar vein, referring to the MacGnome system (discussed in Section 7.4), Edsger Dijkstra wrote:

> 'I was recently exposed to... what... pretended to be educational software for an introductory programming course. With its "visualizations" on the screen, it was... an obvious case of curriculum infantilization.... We must expect from that system permanent mental damage for most students exposed to it.' [94]

Visual Languages are new paradigms for programming, and clearly the existing systems have not been completely convincing. The challenge clearly is to demonstrate

that Visual Programming and Program Visualization can help with real-world problems. The key to this, in my opinion, is to find appropriate domains and new domains to apply these technologies to. For general-purpose programming by professional programmers, textual languages are probably more appropriate. However, we will find new domains and new forms of Visual Language where using graphics will be beneficial. The systems discussed in this paper show that some successful areas so far include, for Visual Programming:

- helping to teach programming (FPL, Pict, etc.),
- allowing non-programmers to enter information in limited domains (OPAL, spreadsheets),
- allowing non-programmers to construct animations (PLAY) and simple computerized lessons for computer-aided instruction (Rehearsal World),
- helping with the construction of user interfaces (Peridot, State Transition UIMS), and
- most significantly, financial planning with spreadsheets.

and for Program Visualization:

- helping to teach algorithms involving data structures (Sorting out Sorting, BALSA),
- helping to teach program concepts, such as Prolog code execution (TPM), and
- helping to debug programs (MacGnome).

9. General Problems and Areas for Future Research

As described in the previous section, the largest area for future research is to prove that Visual Languages will actually help users. In addition, there are a number of more technical problems that most of these systems share.

9.1. All Visual Languages

The problems mentioned in this section apply to many Visual Programming and Program Visualization systems.

Difficulty with Large Programs or Large Data

Almost all visual representations are physically larger than the text they replace, so there is often a problem that too little will fit on the screen. This problem is alleviated to some extent by scrolling and various abstraction mechanisms.

Need for Automatic Layout

When the program or data gets to be large, it can be very tedious for the user to have to place each component, so the system should lay out the picture automatically. Unfortunately, for many graphical representations, generating an attractive layout can be difficult, and generating a perfect layout may be intractable. For example, generating an optimal layout of graphs and trees is NP-Complete [95]. More research is needed, therefore, on fast layout algorithms for graphs that have good user interface characteristics, such as avoiding large scale changes to the display after a small edit.

Lack of Formal Specification

Currently, there is not formal way to describe a Visual Language. Something equivalent to the BNFs used for textual languages is needed. This would provide the field with a 'hard science' foundation, and may allow tools to be created that will make the construction of editors and compilers for Visual Languages easier. Chang [49, 96], Glinert [97] and Selker [98] have made attempts in this direction, but much more work is needed.

Tremendous Difficulty in Building Editors and Environments

Most Visual Languages require a specialized editor, compiler, and debugger to be created to allow the user to use the language. With textual languages, conventional, existing text editors can be used and only a compiler and possibly a debugger needs to be written. Currently, each graphical language requires its own editor and environment, since there are no general purpose Visual Language editors. These editors are hard to create because there are no 'editor-compilers' or other similar tools to help. The 'compiler-compiler' tools used to build compilers for textual languages are also rarely useful for building compilers and interpreters for Visual Languages. In addition, the language designer must create a system to display the pictures from the language, which usually requires low-level graphics programming. Other tools that traditionally exist for textual languages must also be created, including pretty-printers, hard-copy facilities, program checkers, indexers, cross-referencers, pattern matching and searching (e.g. 'grep' in Unix), etc. These problems are made worse by the historical lack of portability of most graphics programs.

Lack of Evidence of Their Worth

There are not many Visual Languages that would be generally agreed are 'successful', and there is little in the way of formal experiments or informal experience that shows that Visual Languages are good. It would be interesting to see experimental results that demonstrated that visual programming techniques or iconic languages were better than good textual methods for performing the same tasks. Metrics might include learning time, execution speed, retention, etc. Fortunately, preliminary results are appearing for the advantages of using graphics for teaching students how to program [36].

Poor Representations

Many visual representations are simply not very good. Programs are hard to understand once created and difficult to debug and edit. This is especially true once the programs get to be a non-trivial size.

Lack of Portability of Programs

A program written in a textual language can be sent through electronic mail, and used, read and edited by anybody. Graphical languages require special software to view and edit; otherwise they can only be viewed on hard-copy.

9.2. Specific Problems for Visual Programming

A primary problem for many Visual Programming languages is that they are 'unstructured' in the software engineering sense. This is because many of them:

- use gotos and explicit transfer of control (often through wires),
- only have global variables,
- have no procedural abstraction,
- if they have procedural abstraction, they may not have parameters for the procedures, and
- have no place for comments.

Another problem is that many Visual Programs do not integrate with programs created in different languages, such as text. A Visual Program might be appropriate for some aspects of the programming task but not others. An exception is MPL (Section 5.2) which uses a Visual Language for matrices and a textual language for everything else. Another approach is for the compiler for the Visual Programming Language to generate conventional computer programs (e.g. in C), so they can be combined with other programs.

9.3. Specific Problems for Program Visualization

Difficulty in Specifying the Display

Newer Program Visualization systems are beginning to ease the task of specifying the display, but it can still be very difficult to design and program the desired graphics. Some systems, such as BALSA-II make it easy to choose from a pre-defined set of displays, but creating other displays can still be very difficult because it involves making low-level calls to the graphics primitives.

Problem of Controlling Timing

For dynamic data visualization, it is difficult to specify when the displays should be updated. Issues of aesthetics in timing are very important to produce useful animations.

10. Conclusions

Visual Programming and Program Visualization are interesting areas that show promise for improving the programming process, especially for non-programmers, but more work needs to be done. The success of spreadsheets demonstrates that if we find the appropriate paradigms, graphical techniques can *revolutionize* the way people interact with computers.

Acknowledgements

For help and support of this article, I would like to thank Bernita Myers. I would also like to thank the British Computer Society Displays Group for making it possible for me to attend the Symposium on Visual Programming and Program Visualisation in London where an earlier version of this paper was presented.

References

1. C. Lewis & G. M. Olson (1987). Can principles of cognition lower the barriers to programming? In: *Empirical Studies of Programmers* Vol. 2, Ablex.
2. B. Shneiderman (1983) Direct manipulation: A step beyond programming languages. *IEEE Computer.* 16, 57–69.
3. D. C. Halbert (1984) Programming by Example. PhD thesis, University of California, Berkeley, 83pp.
4. B. A. Myers (1986) Visual Programming, Programming by Example, and Program Visualization; A Taxonomy. In: *Proceedings SIGCHI '86: Human Factors in Computing Systems* ACM Press, Boston, MA., pp. 59–66.
5. B. A. Myers (1988) *The State of the Art in Visual Programming and Program Visualization* Carnegie Mellon University Computer Science Department Technical Report No. CMU-CS-88-114.
6. *Dictionary of Computing* Oxford: Oxford University Press, 1983.
7. I. E. Sutherland (1963) SketchPad: A man–machine graphical communication system. *AFIPS Spring Joint Computer Conference* 23, 329–346.
8. Adobe Systems, Inc. *Postscript Language Reference Manual* Addison-Wesley, Menlo, 1985, 321pp.
9. Apple Computer, Inc. Inside Macintosh. Addison-Wesley, Menlo, CA. 1985.
10. J. McCormack and Paul Asente. (1988) An Overview of the X Toolkit. In: Proceedings of the ACM SIGGRAPH Symposium on User Interface Software, Banff, Alberta, Canada. Oct, 17–19, ACM Press, 1988. pp 46–55.
11. R. B. Grafton & T. Ichikawa, eds. *IEEE Computer,* Special Issue on Visual Programming. 18, 1985. pp. 6–94
12. D. C. Smith (1977) *Pygmalion: A Computer Program to Model and Stimulate Creative Thought.* Birkhauser: Basel, Stuttgart, 187 pp.
13. B. A. Myers, R. Chandhok & A. Sareen (1988) Automatic Data Visualization for Novice Pascal Programmers, 1988 IEEE Workshop on Visual Languages, *October 10–12, 1988,* Pittsburgh, PA. Computer Society Order Number 876, IEEE Computer Society Press, Terminal Annex, P.O. Box 4699, Los Angeles, CA, pp. 192–198.
14. B. A. Myers (1983) Incense: A System for Displaying Data Structures: Computer Graphics: SIGGRAPH '83 Conference Proceedings, 17, 115–125.
15. R. Baecker (1981) Sorting out Sorting. 16mm color, sound film, 25 minutes. Dynamics Graphics Project, Computer Systems Research Institute, University of Toronto, Toronto, Ontario, Canada. Presented at ACM SIGGRAPH '81. Dallas, TX. Aug. 1981.
16. M. H. Brown & R. Sedgewick (1984) A System for Algorithm Animation. Computer Graphics: SIGGRAPH '84 Conference Proceedings, 18, 177–186.
17. O. Clarisse & S.-K. Chang (1986) VICON: A Visual Icon Manager. In: *Visual Languages* Plenum Press, New York, pp. 151–190.
18. S.-K. Chang, T. Ichikawa & P. A. Ligomenides (eds) (1986) *Visual Languages* Plenum Press, New York.
19. R. R. Korfhage (ed.) 1986 IEEE Workshop on Visual Languages. June 25–27, 1986. Dallas, Texas. Computer Society Order Number 722, IEEE Computer Society Press, Terminal Annex, P.O. Box 4699, Los Angeles, CA, 179 pages.
20. E. Jungert, ed. 1987 Workshop on Visual Languages. August 19–21, 1987. Linkoping, Sweden. IEEE Computer Society.
21. N.-C. Shu (1988) *Visual Programming.* New York: Van Nostrand Reinhold Company.
22. A. S. Berztiss, ed. 1988 IEEE Workshop on Visual Languages. October 10–12, 1988. Pittsburgh, PA. Computer Society Order Number 876, IEEE Computer Society Press, Terminal Annex, P.O. Box 4699, Los Angeles, CA 90051.
23. T. O. Ellis, J. F. Heafner & W. L. Sibley (1969) The Grail Project: An Experiment in Man–Machine Communication. Rand Report RM-5999-Arpa.
24. W. R. Sutherland (1966) On-line Graphical Specification of Computer Procedures. MIT PhD thesis. Lincoln Labs Report TR-405.
25. C. Christensen (1968) An Example of the Manipulation of Directed Graphs in the AMBIT/G Programming Language. In: *Interactive Systems for Experimental Applied*

Mathematics (Melvin Klerer and Juris Reinfelds, eds) New York: Academic Press, pp. 423–435.
26. C. Christensen (1971) An Introduction to AMBIT/L, A Diagramatic Language for List Processing, Proceedings of the 2nd Symposium on Symbolic and Algebraic Manipulation. Los Angeles, CA. Mar. 23–25, 1971. pp. 248–260.
27. M. M. Zloof & S. Peter de Jong (1977) The System for Business Automation (SBA): Programming Language, *CACM* 20(6), June, pp. 385–396.
28. M. M. Zloof (1981) QBE/OBE: A Language for Office and Business Automation, *IEEE Computer.* 14(5), May, pp. 13–22.
29. M. C. Pong & N. Ng (1983) Pigs—A System for Programming with Interactive Graphical Support. *Software—Practice and Experience.* 13, 847–855.
30. M.-C. Pong (1986) A Graphical Language for Concurrent Programming, IEEE Computer Society Workshop on Visual Languages. IEEE CS Order No. 722. Dallas, Texas. June 25–27, pp. 26–33.
31. N. C. Shu (1985) FORMAL: A Forms-Oriented Visual-Directed Application Development System, *IEEE Computer,* 18, 38–49.
32. E. P. Glinert & L. Tanimoto (1984) Pict: An Interactive Graphical Programming Environment, *IEEE Computer.* 17, 7–25.
33. M. B. Albizuri-Romero (1984) GRASE—A Graphical Syntax-Directed Editor for Structured Programming, *SIGPLAN Notices.* 19, 28–37.
34. T. Pietrzykowski, S. Matwin & T. Muldner (1983) The Programming Language PROGRAPH: Yet Another Application of Graphics, Graphics Interface '83, Edmonton, Alberta. May 9–13, pp. 143–145.
35 T. Pietrzykowski & S. Matwin (1984) PROGRAPH: A Preliminary Report. University of Ottawa Technical Report TR-84-07. April, 1984. 91 pages.
36. Nancy Cunniff, R. P. Taylor & J. B. Black (1986) Does Programming Language Affect the Type of Conceptual Bugs in Beginners' Programs? A Comparison of FPL and Pascal, Proceedings SIGCHI '86: Human Factors in Computing Systems. Boston, MA. April 13–17, 1986. pp. 175–182.
37. R. J. K. Jacob. (1985) A state transition diagram language for visual programming. *IEEE Computer* 18, 51–59.
38. T. H. Taylor & R. P. Burton (1986) An icon-based graphical editor *Computer Graphics World* 9, 77–82.
39. S. L. Tanimoto & M. S. Runyan (1986) PLAY: An Iconic Programming Systems for Children In: *Visual Languages* New York: Plenum Press, pp. 191–205.
40. T. Ae, M. Yamashita, W. C. Cunha, & H. Matsumoto (1986) Visual User-Interface of A Programming System: MOPS-2, IEEE Computer Society Workshop on Visual Languages. IEEE CS Order No. 722. Dallas, Texas. June 25–27, 1986. pp. 44–53.
41. J. M. Moshell, C. E. Hughes, L. W. Lacy & R. L. Lewis. (1987) A Spreadsheet-Based Visual Language for Freehand Sketching of Complex Motions, 1987 Workshop on Visual Languages. August 19–21, 1987. Linkoping, Sweden. IEEE Computer Society. pp. 94–104.
42. M. A. Musen, L. M. Fagen & E. H. Shortliffe (1986) Graphical Specification of Procedural Knowledge for an Expert System, IEEE Computer Society Workshop on Visual Languages. IEEE CS Order No. 722 Dallas, Texas, June 25–27, 1986. pp. 167–178.
43. A. L. Ambler (1987) Forms: Expanding the Visualness of Sheet Languages, 1987 Workshop on Visual Languages. August 19–21, 1987. Linkoping, Sweden. IEEE Computer Society. pp. 105–117.
44. E. P. Glinert (1987) Out of Flatland: Towards 3-D Visual Programming, Proceedings of FJCC '87—1987 Fall Joint Computer Conference. IEEE Computer Society. Dallas, Texas, October 25–29, 1987. pp. 292–299.
45. Mike Graf (1987) A Visual Environment for the Design of Distributed Systems, 1987 Workshop on Visual Languages. August 19–21, 1987. Linkoping, Sweden. IEEE Computer Society. pp. 330–344.
46. D. Harel (1988) On Visual Formalisms, *CACM* 31(5), May, pp. 514–530.
47. National Instruments. LabVIEW. 12109 Technology Blvd. Austin, Texas, 78727.

48. M. W. Maimone, J. D. Tygar & J. M. Wing (1988) Miro Semantics for Security, 1988 IEEE Workshop on Visual Languages. October 10–12, 1988. Pittsburgh, PA. Computer Society Order Number 876, IEEE Computer Society Press, Terminal Annex, P.O. Box 4699, Los Angeles, CA 90051. pp. 45-51.
49. S. K. Chang, M. Tauber, B. Yu & J. S. Yu (1989) A Visual Language Compiler, *IEEE Transactions on Software Engineering*. May, pp. 506–525.
50. P. D. Wellner (1989) Statemaster: A UIMS based on Statecharts for Prototyping and Target Implementation, Proceedings SIGCHI '89: Human Factors in Computing Systems. Austin, Tex. April 30 May 4, 1989. pp. 177–182.
51. R. Yeung (1988) MPL—A Graphical Programming Environment for Matrix Processing Based on Logic and Constraints, 1988 IEEE Workshop on Visual Languages. October 10–12, 1988. Pittsburgh, PA. Computer Society Order Number 876, IEEE Computer Society Press, Terminal Annex, P.O. Box 4699, Los Angeles, CA 90051. pp. 137–143.
52. D. E. Shaw, W. R. Swartout & C. C. Green (1975) Inferring Lisp Programs from Examples, Fourth International Joint Conference on Artificial Intelligence. Tbilisi, USSR. Sept. 3–8, 1975. 1, pp. 260–267.
53. H. Lieberman (1982) Constructing Graphical User Interfaces by Example, Graphics Interface '82, Toronto, Ont. Mar. 17–21, 1982. pp. 295–302.
54. R. P. Nix. (1985) Editing by example. *ACM Transactions on Programming Languages and Systems*. 7, 600–621.
55. M. A. Bauer (1978) A Basis for the Acquisition of Procedures. PhD Thesis, Department of Computer Science, University of Toronto. 310 pages.
56. D. C. Halbert (1981) An Example of Programming by Example. Masters of Science Thesis. Computer Science Division, Dept. of EE&CS, University of California, Berkeley and Xerox Corporation Office Products Division, Palo Alto, CA. June, 55 pages.
57. L. Gould & W. Finzer (1984) Programming by Rehersal. Xerox Palo Alto Research Center Technical Report SCL-84-1. May, 133 pages.
58. L. Gould & W. Finzer. Programming by rehersal, *Byte* 9, 187–210.
59. A. Borning (1986) Defining Constraints Graphically, Human Factors in Computing Systems: Proceedings SIGCHI '86. Boston, MA. Apr. 13–17, 1986.
60. P. Desain (1986) Graphical Programming in Computer Music, Proceedings of the International Computer Music Conference. Royal Conservatory, The Hague, Netherlands. Oct. 20–24, 1986. pp. 161–166.
61. M. Hirakawa, S. Iwata, I. Yoshimoto, M. Tanaka & T. Ichikawa (1987) HI-VISUAL Iconic Programming, 1987 Workshop on Visual Languages. August 19–21, 1987. Linkoping, Sweden. IEEE Computer Society. pp. 305–314.
62. D. Kozen, T. Teitelbaum, W. Chen, J. Field, W. Pugh, B. Vander Zanden (1987) ALEX-An Alexical Programming Language, 1987 Workshop on Visual Languages. August 19–21, 1987. Linkoping, Sweden. IEEE Computer Society. pp. 315–329.
63. B. A. Myers (1987) Creating interaction techniques by demonstration. *IEEE Computer Graphics and Applications*, 7, 51–60.
64. B. A. Myers (1988) Creating User Interfaces by Demonstration. Boston: Academic Press.
65. D. N. Smith (1988) Visual Programming in the Interface Construction Set, 1988 IEEE Workshop on Visual Languages. October 10–12, 1988. Pittsburgh, PA. Computer Society Order Number 876, IEEE Computer Society Press, Terminal Annex, P.O. Box 4699, Los Angeles, CA 90051. pp. 109–120.
66. F. Ludolph, Y.-Y. Chow, D. Ingalls, S. Wallace & K. Doyle (1988) The Fabrik Programming Environment, 1988 IEEE Workshop on Visual Languages. October 10–12, 1988. Pittsburgh, PA. Computer Society Order Number 876, IEEE Computer Society Press, Terminal Annex, P.O. Box 4699, Los Angeles, CA 90051. pp. 222–230.
67. I. Nassi & B. Shneiderman (1973) Flowchart techniques for structured programming. *SIGPLAN Notices*. 8, 12–26.
68. T. Ae & R. Aibara (1987) A Rapid Prototyping of Real-Time Software Using Petri Nets, 1987 Workshop on Visual Languages. August 19–21, 1987. Linkoping, Sweden. IEEE Computer Society. pp. 234–241.

69. A. T. Berztiss (1987) Specification of Visual Representations of Petri Nets, 1987 Workshop on Visual Languages. August 19–21, 1987. Linkoping, Sweden. IEEE Computer Society. pp. 225–233.
70. B. A. Myers (1989) User interface tools: introduction and survey. *IEEE Software* **6**, 15–23.
71. A. Kay (1984) Software. *Scientfic American*. September.
72. A. W. Biermann (1976) Approaches to automatic programming. In: *Advances in Computers* (Morris Rubinoff and Marshall C. Yovitz, eds) New York: Academic Press. pp. 1–63.
73. A. Borning (1979) Thinglab—A Constraint-Oriented Simulation Laboratory. Xerox Palo Alto Research Center Technical Report SSL-79-3. July, 1979. 100 pages.
74. A. Borning (1981) The programming language aspects of thinglab; a constraint-oriented simulation laboratory. *Transactions on Programming Language and Systems*, **3**, 353–387.
75. D. C. Smith, C. Irby, R. Kimball, B. Verplank & E. Harslem (1982) Designing the star user interface. *Byte Magazine* April. pp. 242–282.
76. L. M. Haibt (1959) A Program to Draw Multi-Level Flow Charts, Proceedings of the Western Joint Computer Conference. San Francisco, CA. 15, Mar. 3–5, 1959. pp. 131–137.
77. R. Baecker & A. Marcus (1986) Design Principles for the Enhanced Presentation of Computer Program Source Text, Human Factors in Computing Systems: Proceedings SIGCHI '86. Boston, MA. Apr. 13–17, 1986.
78. G. P. Brown, R. T. Carling, C. F. Herot, D. A. Kramlich & P. Souza (1985) Program visualization: graphical support for software development. *IEEE Computer* **18**, 27–35.
79. M. Moriconi & D. F. Hare (1985) Visualizing Program Designs Through PegaSys. *IEEE Computer* **18**, 72–85.
80. R. Chandhok, *et al.* (1985) Programming Environments based on structure editing: The Gnome approach, Proceedings of the National Computer Conference (NCC '85). AFIPS, 1985.
81. W. Cunningham & K. Beck (1986) A Diagram for Object-Oriented Programs, OOPSLA '86 Proceedings. September 29–October 2, 1986. Portland, Oregon. SIGPLAN Notices. 21(11), November. pp. 361–367.
82. M. Eisenstadt & M. Brayshaw (1987) The Transparent Prolog Machine: an execution model and graphical debugger for logic programming, to appear in Journal of Logic Programming. Human Cognition Research Laboratory Technical Report No. 21a. The Open University. Milton Keynes, MK7 6AA, England. October, 1987.
83. R. M. Baecker (1968) Experiments in On-Line Graphical Debugging: The Interrogation of Complex Data Structures, (Summary only). First Hawaii International Conference on the System Sciences. Jan. pp. 128–129.
84. K. C. Knowlton (1966) L6: Bell Telephone Laboratories Low-Level Linked List Language. Black and white shoud files, Bell Laboratories, Murray Hill, NJ.
85. B. A. Myers (1980) Displaying Data Structures for Interactive Debugging. Xerox Palo Alto Research Center Technical Report CSL-80-7. June, 1980. 97pp.
86. R. M. Baecker (1975) Two systems which produce animated animated representations of the execution of computer programs. *SIGCSE Bulletin* **7**, 158–167.
87. J. L. Bentley & Brian W. Kernighan (1987) A System for Algorithm Animation; Tutorial and User Manual. AT&T Bell Laboratories Computing Science Technical Report No. 132. 600 Mountain Avenue, Murray Hill, NJ 07974. January, 1987.
88. M. H. Brown (1988) Exploring Algorithms Using Balsa—II, *IEEE Computer.* **21**(5), May, pp. 14–36.
89. R. L. London & R. A. Druisberg (1985) Animating programs in smalltalk. *IEEE Computer* **18**, 61–71.
90. A. Hyrskyakari & K.-J. Raiha (1987) Animation of Algorithms Without Programming, 1987 Workshop on Visual Languages. August 19–21, 1987. Linkoping, Sweden. IEEE Computer Society, pp. 40–45.
91. R. A. Duisberg (1987) Visual Programming of Program Visualizations, 1987 Workshop on Visual Languages. August 19–21, 1987. Linkoping, Sweden. IEEE Computer Society, pp. 55–56.

92. J. T. Stasko (1989) TANGO: A Framework and System for Algorithm Animation. PhD Dissertation. Brown University, Department of Computer Science, Providence, RI 02912. Technical Report No. CS-89-30, May, 1989. 257pp.
93. F. P. Brooks, Jr. (1987) No silver bullet: essence and accidents of software engineering. *IEEE Computer* 20, 10–19.
94. E. W. Dijkstra (1989) On the Cruelty of Really Teaching Computing Science, The SIGCSE Award Lecture, *CACM* 32, 1403–1404.
95. D. S. Johnson (1982) The NP-Completeness Column: an ongoing guide, *Journal of Algorithms.* 3. pp. 89–99.
96. S.-K. Chang, G. Tortora, B. Yu & A. Guercio (1987) Icon Purity—Toward a Formal Theory of Icons, 1987 Workshop on Visual Languages. August 19–21, 1987. Linkoping, Sweden. IEEE Computer Society. pp. 3–16.
97. E. P. Glinert & J. Gonczarowski (1987) A (formal) model for (iconic) programming environments In: *Human-Computer Interaction–interact '87.* Elsevier Science Publishers (North Holland). pp. 283–290.
98. T. Selker & L. Koved (1988) Elements of Visual Language, 1988 IEEE Workshop on Visual Languages. October 10–12, 1988. Pittsburgh, PA. Comput r Society Order Number 876, IEEE Computer Society Press, Terminal Annex, P.O. Box 4699, Los Angeles, CA 90051. pp. 38–43.