The British Computer Society • Human-Computer Interaction Group
"Interfaces", Autumn '95

Noddy's Guide to ...

Visual Programming

Thomas Green

What is Visual Programming?

According to Myers [1]: "'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." This computer-centered viewpoint is fairly standard in the computing science (CS) fraternity. Typographers, on the other hand, might take a more person-centered stance, arguing that programming languages lie on a continuum, depending on how much use can be made of graphical elements (e.g. blank lines, indenting) to communicate something to the reader. I shall use VPL (visual programming language) in Myers's sense here.

'Visual' is really a misnomer for 'graphical', but unfortunately it caught on. However, the advent of Visual Basic (which is *not* a VPL, because the program code is in text) is changing the accepted usage – certainly the Internet discussion group (see **Resources**) is now swamped by queries about Visual Basic – and VP may have to rename itself.

Systems for providing visualizations of text programs are rather different; they either create a picture of the program structure, or provide a dynamic illustration of program execution. [4] [10] [13] give examples. Although a computer-centered viewpoint construes software visualization as different from 'proper' VPLs, because the computer executes textual code, it raises similar cognitive issues about information presentation and has similar aims. No space for it here, though.

VP past and present

VP started with attempts to produce executable flowcharts, partly influenced by the belief that flowcharts would be good for teaching novices. Many other possibilities have since opened up (see **Styles**) and the claims of flowcharts are now less well thought of; also people's ideas about 'the programming situation' have broadened [11]. For some reason, there was little idea-exchange between early HCI and early VP.

Today, lively research has spawned a specialist journal and a conference series (see **Resources**). Research systems are proliferating, although mostly developed by visionary computer scientists rather than HCI folk. Object-oriented, concurrent, and real-time systems have been developed. Commercial VP languages include Prograph and LabVIEW, both used as language-of-choice by small but enthusiastic communities (including where I work), and others are in preparation or have been released; most are for specialised use, but at least some general-purpose commercial programming has been done in Prograph. Many technical problems have been solved but some big ones remain (see **Issues**). Occasional papers appear on HCI issues and cognitive issues in VP but the area is still overwhelmingly technology-led.

Why is VP interesting?

Burnett et al. [2] nimbly describes the goals of VPL designers as trying to make it easier to express and/or understand programs, through simplicity, concreteness, explicitness, and responsiveness. Modern VPLs accordingly (i) reduce the number of concepts needed to program (e.g. no variables), (ii) allow data objects to be explored directly, (iii) explicitly depict relationships, and (iv) give immediate visual feedback of updated computations during editing. So it is claimed; non-VP fans might argue that modern textual languages have similar aims.

The CS community has at least 3 reasons for interest. (i) It finds VP technically challenging, so there are all the usual reasons for technology-led research ('here's an interesting problem, go off and do a thesis on it'). (ii) VP supports the data-flow model of computation (see **Styles**) much better than does traditional text. (iii) Although 'programmers are users too,' conventional HCI has made little explicit penetration into the design of programming environments. The VP movement can be seen as an 'alternative HCI' leading to more usable programming environments. The CS view of VP has often been based on naive cognitive models, but happily, VP-enthusiasts are not too dogmatic.

The HCI community should be interested because (i) here is another good way to interact with computers, this time on a rather harder topic than text-editing, (ii) there are many unsolved interaction problems (see **Issues**), and (iii) VP may open computing power to many more people –

MRC Applied Psychology Unit
15 Chaucer Road, Cambridge CB2 2EF

although whether those people will be end-users, novices, 'gardeners' (see [11] for ideas here) or everyday programmers is not yet established. Despite that 'should', there are few VP papers in the CHI and HCI series.

Cognitive psychologists are interested (well, some of us are) because (i) CS folk develop wonderful systems and it's frustrating that they don't have that extra pinch of cognitive thinking, (ii) the problems of information display and manipulation are intense, and (iii) earlier hypotheses about mental representations of programs and about the cognitive processes of program design can be tested in new situations which are likely to force modifications to existing views – possibly with consequences extending to many other types of design activity.

Styles

A good classification of VP languages needs more than one dimension [1], but here are some samples.

(i) The *control-flow* model: executable flowcharts have declined in popularity – deservedly so, according to Curtis et al's well-controlled study [3] – but 'R-technology', developed in the former Soviet Union, offers an intriguing blend of graphical elements for control with Pascal-like elements for declarations and formulas, with 500 active users in 1989 [9].

(ii) In the *data-flow* model, data travels from input sources to operators and ultimately to output sinks. Each operator acts as soon as it has data ready. The model is usually represented as a box-and-line diagram (see Figure 1). Because the data travels around in packets, it is easy to build primitives which operate on aggregate data (e.g. a primitive to sum a whole array). It is also easy for the programmer to insert 'test probes' at selected spots to get a picture of the program in operation.

Representing flow of control in data-flow programs is difficult. Iteration is represented in Prograph by putting iteration-controls around operators which may themselves be sub-programs; in LabVIEW it is represented by a frame enclosing the operations to be repeated, with various types of iteration control for while-loops and counted-loops. LabVIEW has two conditionals: boolean operators (as in Figure 1), and little flip-boxes which contain alternative sub-programs, selected by a boolean input (equivalent to if-then-else or to case statements, but with only one arm showing at a time). Neither technique is particularly successful for complex cases [6]. Prograph, instead of flip-boxes, puts the alternatives in separate windows, thereby spawning a deep tree of windows and sub-windows. Some systems have used a visual representation of a *nand* operator.

(iii) *Visual production systems*, like textual production systems, contain a number of rules with left-halves and right-halves, together with an internal 'world'. Any time the situation in the 'world' matches rule X's left half, that rule fires, and the world is transformed in the way shown in its right half. (Various methods for resolving contention have been found.) In visual production systems the left and right halves are specified as pictures. ChemTrains [8] is an example of this technique. KidSim [15] is an Apple project which adds ideas about agents and programming by demonstration, creating what they call 'programming without a programming language'. Parallel research by David Gilmore (unpublished) agrees that KidSim excites and empowers young children but suggests optimism should be tempered with caution about how much programming they learn.

(iv) Other declarative VPLs are now being investigated, *constraint* and *logic-based*. Figure 2 shows a declarative program in Forms/3; the key feature of this, as in a text-based declarative language, is that the programmer merely states the relations that must be satisfied. Note the use of fill patterns as identifiers or 'names'. Although Forms/3 looks unfamiliar, it fared well in an empirical program-writing comparison against Pascal and improved APL [12]. Other examples include various visual versions of Prolog. No similar commercial languages are available, so far as I know, so anecdotal evidence is scarce.

(v) The humble *spreadsheet* contains many of the features of VPLs – dataflow, aggregate operators, and a visual formalism – so although it contains no explicit graphical lines showing data-flow between cells, it bears examination. Nardi [11] praises it as an easy-to-use vehicle for collaborative working, for reasons that should apply equally well to full VPs; Hendry & Green [7] agree with many of Nardi's points but describe some worrying user difficulties.

A few current issues

(i) Is VP any good? Although lab studies have not produced much evidence in its favour, some people like and use VPLs. Visual tracers likewise. So the lab studies have missed something. What? (Maybe it's fun, like a video game.)

(ii) Does the visual component really contribute anything extra? Sometimes 'visual' systems do nothing that can't be done as well or better with straight text; perhaps the real issues are how layout and locality can be used to convey meaning [14].

(iii) Scaling-up – the bogey of VPL buffs, and a key problem for real acceptance. Although certainly not limited to toy problems, VP-in-the-large is still to come. Perhaps it won't. Is there a real-life domain for VPLs? See [2] for discussion of many sides to this question (efficiency, screen real estate, abstraction, persistence, etc.)

(iv) Supporting the programming process: it is easier to design an elegant data representation than one that is easy to use. In this as in many other fields, there are designers who only think about *adding* information, not about other programming tasks. (a) Building a VPL program may be easy (if you know what you want), but modifying it may

run into difficulties. Informally, times for an equivalent change to a Basic program and its LabVIEW counterpart (performed by experienced users) were 10 times faster for the Basic. (b) Comprehending a VPL may be difficult, too, partly because the facilities are usually poor. (c) 'Power tools' for global editing, project development, and version control need to be developed.

(v) We need alternative representations, especially ascii for integrating with e-mail.

(vi) What about audio, 3D graphics and virtual reality?

Will VPLs go anywhere?

Although some general-purpose commercial programs have been written in VPLs, they will never oust C++, let alone Cobol. If VP is to get established, it must be in new user communities, where older technologies have not already got established. End-user programming may be one such, especially using a visual macro language embedded in an application, or for handling novel tasks such as video- or music-editing, or (like LabView) in specialist areas; another market may be the control of domestic and workplace gadgetry, such as speech-driven dialogues, intelligent heating and control systems, agents for data-collection, etc. These new markets will be much more demanding of good HCI than the traditional Cobol and C++ communities; fortunately the VP community is now becoming aware of that.

Resources

Videos would be good but I don't know of any comprehensive archive (any of my readers know of one?). There are probably some scattered through the SIGGRAPH archives.

Books/journals: The most comprehensive book-length overview is [5]. The *Journal of Visual Languages and Computing*, published by Academic Press, mostly technical but with special issues on cognitive aspects of VP (1993) and HCI issues (call for papers now out, guest editor Wayne Citrin, `citrin@cs.colorado.edu`). Special issues: IEEE *Computer* (in preparation); *Dr Dobbs Jnl* (Dec., 1995).

Internet: `comp.lang.visual` is intended for discussion of VP, but at the time of writing it is swamped by queries about Visual Basic and Visual C++. An excellent Frequently-Asked-Questions message, by David McIntyre, gives definitions, issues, references, pointers to active sites, ftp-able information, and a few WWW addresses for information from individual labs, some of which offer ftp-able documents.

Some info about *WWW* is collected at:

`http://www.cogs.susx.ac.uk/users/ianr/vpl.html`

Meetings: Since 1984 there have been annual IEEE meetings on VP. The most recent was *Visual Languages '93* (IEEE Computer Society Press #3970-02); at the time of writing *Visual Languages '94* was just about to start. The *Empirical Studies of Programmers* meetings (published by Ablex) have recently included a few VP papers, as have CHI, UIST and HCI.

References

[1] Burnett, M. M. and Baker, M. J. (1994) A classification system for visual programming languages. Oregon State University Dept of Computer Science Tech. Rep. 93-60-14. A proposal for classification, not a survey. Ftp details from WWW at: `http://www.cs.orst.edu/~burnett/vpl.html`

[2] Burnett, M. M., Baker, M. J., Bohus, C., Carlson, P., van Zee, P. and Yang, S. (1994) The scaling-up problem for visual programming languages. *IEEE Computer*, March 1995 (to appear). Wide-ranging discussion with plentiful examples.

[3] Curtis, B., Sheppard, S., Kruesi-Bailey, E., Bailey, J. and Boehm-Davis, D. (1989) Experimental evaluation of software documentation formats. *J. Systems and Software*, 9 (2), 167-207. Excellent paper-based study; no advantage for graphical formats.

[4] Domingue, J., Price, B. A. and Eisenstadt, M. (1992) A framework for describing and implementing software visualization systems. *Proc. Graphics Interface '92*, Vancouver. Analysis and re-implementation of classic systems by the most active UK group.

[5] Glinert, E. P. (1990) *Visual Programming Environments: vol. 1. Paradigms and Systems, vol. 2. Applications and Issues.* Los Alamitos, CA: IEEE Computer Society Press. The biggest survey. Do a supplement, please.

[6] Green, T. R. G., Petre, M. and Bellamy, R. K. E. (1991) Comprehensibility of visual and textual programs: a test of 'Superlativism' against the 'match-mismatch' conjecture. In J. Koenemann-Belliveau, T. Moher, and S. Robertson (Eds.), *Empirical Studies of Programmers: Fourth Workshop.* Norwood, NJ: Ablex. Pp. 121-146. A very specific comparison of conditional structures (text was better). Main point: no single notation is best for all purposes, whether text or visual.

[7] Hendry , D. G. and Green, T. R. G. (1994) Creating, comprehending, and explaining spreadsheets: a cognitive interpretation of what discretionary users think of the spreadsheet model. *Int. J. Human-Computer Studies,* 40(6), 1033-1065. A distillation of interviews.

[8] Lewis, C., Rieman, J. and Bell, B. (1991) Problem-centered design for expressiveness and facility in a graphical programming system. *Human-Computer Interaction*, 6(3-4), 319-355. ChemTrains - the story of its making. From a highly inventive HCI group.

[9] McHenry, W. K. (1990). R-Technology: a Soviet visual programming environment.

Journal of Visual Languages and Computing, 1(2), 199-212. The system and its context.

[10] Myers, B. A. (1990) Taxonomies of visual programming and program visualization. *J. Visual Languages and Computing*, 1 (1) 97-123. A catalogue raisonné.

[11] Nardi, B. (1993) *A Small Matter of Programming: Perspectives on End-User Computing*. MIT Press. Spreadsheets – situated use – collaboration – a people-centred view.

[12] Pandey, R. and Burnett, M. M. (1993) Is it easier to write matrix manipulation programs visually or textually? An empirical study. *IEEE Symp. Visual Languages*, 1993, Bergen, Norway. The only empirical study of programming in a declarative VPL.

[13] Price, B.A., Baecker, R. M. & Small, I. S. (1993) A principled taxonomy of software visualization. *J. Visual Languages and Computing*, 4 (3) 211-266. Twelve systems compared in impressive detail under 6 categories each with subcategories.

[14] Raymond, D. (1991). Characterizing visual languages. *Proc. 1991 IEEE Workshop on Visual Languages*. (Kobe, Japan). A shrewd sceptic examines the hype, argues that VPs should be analogue, not notational.

[15] Smith, D. C., Cypher, A. and Spohrer, J. (1994) KidSim: programming agents without a programming language. *Comm. ACM* 37(7) [July], 55-66. Pretty pictures, interesting system. Not over-critical but very insightful.

Acknowledgements

Alan Blackwell, Simon Buckingham Shum, Margaret Burnett, John Domingue, David Gilmore, and Darrell Raymond (alphabetic order) all deserve thanks. Prograph is a trademark of TGS Systems Ltd., LabVIEW of National Instruments Corp.

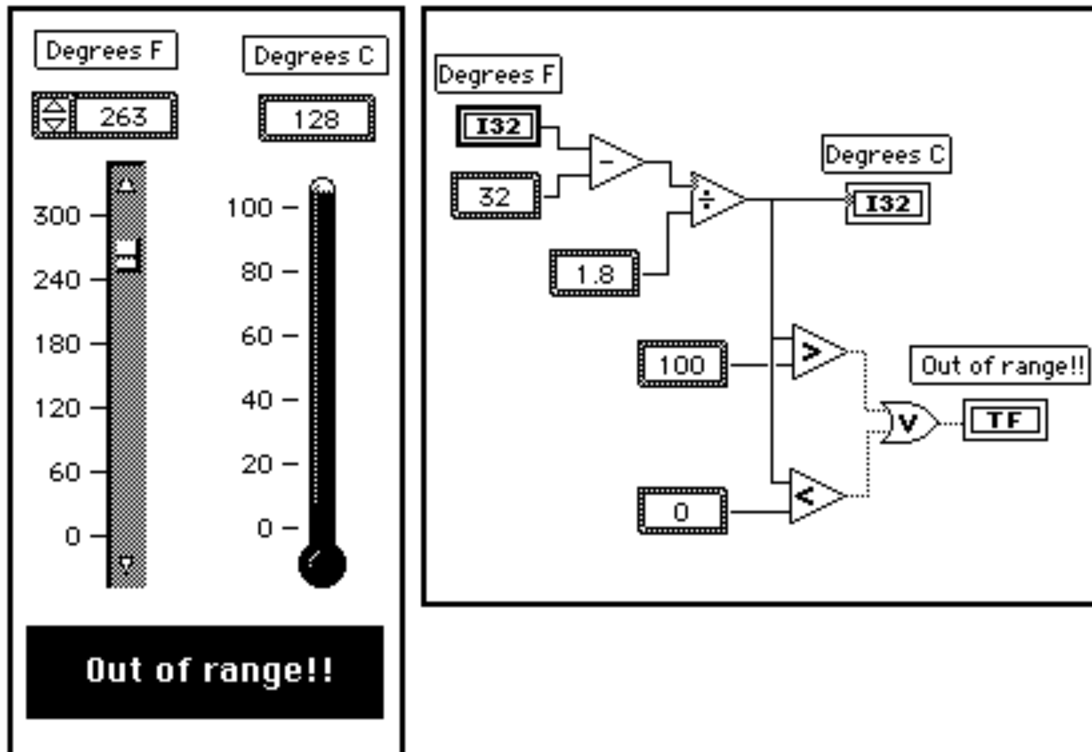

Figure 1: A LabVIEW program for temperature conversion. Data is entered through the panel (left box) by moving the ‘degrees F’ slider or adjusting the thumbwheel. The program (right box) subtracts 32, divides by 1.8, and sends the result to the thermometer-style output on the panel. Out-of-range temperatures cause a warning sign to be highlighted. Editing and execution tools not shown.

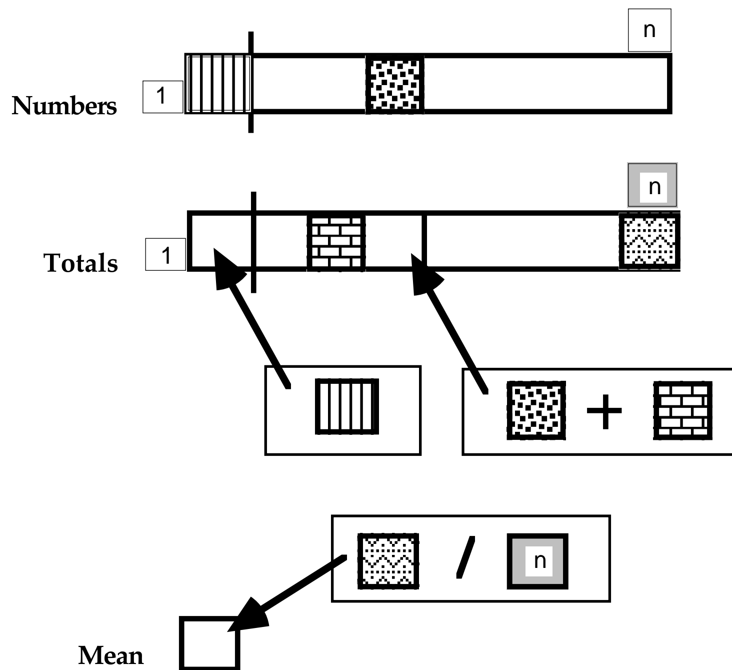

Figure 2: Finding the mean in Forms/3 (redrawn from [12]). Numbers is a 1-dimensional data array, divided into cells numbered 1 to n. The first cell and a 'typical' cell have been shaded for reference purposes in the program. To compute the overall sum, a second data array has been constructed, called Totals: the first cell (partitioned from the other cells by a vertical line, to show it has a different formula) has the value of the first cell of Numbers; all other cells contain the formula "sum of corresponding cell in Numbers plus previous cell of Totals". (Arrays and cells have thick borders, formulas have thin borders.) The mean is computed by dividing the overall total, found in the last cell of Totals, by the number of cells, n. All references to cells are indicated by shading.