Visual Programming

A Promise For The Future?

Guus Bosman

July 2002

Division of Mathematics and Computer Science,
Vrije Universiteit

www.guusbosman.nl

vrije Universiteit amsterdam

Table of contents

1 Introduction 3
2 Visual Programming Languages 4
3 Programming and mental models 5
3.1 Mental models 5
3.2 Mental models of programmers 5
4 Restraints on mental models by notations 7
4.1 Empirical results 7
4.2 “Natural” solutions 7
4.3 Match-mismatch conjecture 8
5 Theory on Visual Languages 9
5.1 Cognitive processes 9
5.2 Perception research 9
5.3 Dimensions 9
5.4 Attractiveness of graphics 10
6 Multiple views 12
7 Conclusion and further research 13
8 References 14

1 Introduction

To use a computer, programs have to be written. In the early years of computing this involved cards which were programmed by ponsing holes in them. Later it became possible to write programs using the computer itself, a kind of “bootstrapping”. Although this was a big step forward, the way computers operate is quite different than the way humans do. Therefore new programming languages were designed, and in the course of 40 years many programming languages came forward. Programming was a very specialized job that required intimate knowledge of the machine. Technically, a computer has to be instructed how to calculate by a long list of instructions. However, for people this is considered not the most natural way of calculation.

It was generally found that it would be useful to improve abstraction, to allow more people to program, the so-called *end-user programming*. Typical examples of this are spreadsheets, in which a user can write quite sophisticated calculations without having to know the inside details of a computer. The biggest problem these languages try to solve is the gap between the computer and the way people think.

In the ongoing quest to make programming easier and more productive experiments were started in designing Visual Languages. In contrast to the old languages, which were written in a text-form, these early investigators were looking to other fields of engineering (for example mechanics) where the use of pictures and diagrams is very common, and considered very useful by the engineers *[Larkin 1987]*. In a Visual Language, the pictures represent the executing of a program, as does text in ordinary computer languages. In the beginning, the design of this kind of languages was difficult because of the lack of support in technology.

With today’s desktop computers, which all are equipped with sophisticated graphical systems, Visual Programming is easier to implement, and in fact a number of (commercial) products exist today. However, aside from the technical feasibility, there is also a more important question: is visual programming easier or better than the old-fashioned way? In general it is considered important to allow many people to program computers, so the focus on research lies in making programming more accessible specifically for people who haven ’t programmed before (novices).

In this paper I’ll explore literature on Visual Programming languages, focussed on comprehensibility of programming. To goal of the paper is to investigate what the current research is on visual programming: what does theory say about whether or not it's possible to improve programming or program comprehension using Visual Languages.

2 Visual Programming Languages

There are many tools and editors that have a graphical view, that help programming. However, not all of them are real visual programming languages. The following classification can be found in *[Shu 1988]*:

- Visual Environment
  - Visualisation of
    - Data or information about data
    - Program and/or execution
    - Software design
  - Visual coaching
- Visual Languages
  - For handling visual information
  - For supporting visual interactions
  - For actually programming with visual expressions
    - Diagrammatic systems
    - Iconic systems
    - Form systems

The difference between "Visual Environment" and "Visual Languages" is the kind of programming language. In a visual environment the programming language being used is in principle still a linear (text) one, only with visual aid. In this paper the second type "Visual Languages" will be discussed, specifically "for actually programming with visual expressions". Chang called the languages we discuss "Visual Programming Languages" *[Chang 1987]*. We will not discuss languages in deep detail in this paper; therefore the difference between the diagrammatic, iconic and form systems will be ignored.

A difference between a real VPL[1] and a Visual Environment is often that in a Visual Language the graphics can be manipulated, just as you might expect from a real programming language. In a Visual Environment, the graphics are just there to support comprehension, but they cannot be manipulated. They are 'read-only', and only the text, the code, is editable.

To research the effect of visual languages on the ease of programming, several matters will be looked at. First of all an understanding how programmers work is necessary. Which mental models and cognitive skills are used while programming? Although the field psychology of programming is still young, some results have already been established. An understanding is required of how people retrieve graphical information, and why graphical languages are so much investigated: why are they so popular among researchers and why do programmers seem to expect so much of graphical tools?

[1] VPL: Visual Programming Language

3 Programming and mental models

3.1 Mental models

Users build internal models or internal representations of the world they interact with. These representations are called Mental Models.

Many cognitive psychologists believe that these mental models are responsible for the processes involved in knowledge acquisition and knowledge construction when learning in a complex cognitive domain such as computer programming. Investigation of mental models can help us understand the mental representations people construct when solving a problem. Rogers (1992) writes that mental models are not models *of* the mind but models *in* the mind: they are created in the head of the problem solver. She also states that the concept of a mental model in the psychological literature is confused and confusing. However, the idea is intuitively fairly clear: thinking involves the creation and internalisation of simplified models of reality.

3.2 Mental models of programmers

In order to create a program a programmer has a number of mental models. They need one of the task to be programmed, and of the environment of the program. Then they need a mental model to describe the program itself. This model will have to be refined (more than once) in order to translate it into a computer program. It is important for designers of computer languages to understand what kind of representation of the computer system a user has in his mind (*user models* according to Rogers).

A definition of a programming language is: a computational paradigm expressed in a notation.

In the computer the program will be stored in a certain notation. This notation is called a programming language. Professional programmers typically work with big programs that (in text-form) span over many screens. This means that there is a 'window view' on the program. This makes the understanding of the program as it is written in text difficult.

Program understanding (comprehension) has been studied extensively, and from various angles. Understanding a program involves various aspects. A programmer not only has to understand what each program statement does, but he also has to understand the execution sequence of the statements (control flow), the transformational effects on data objects (data flow) and the purposes of groups of statements (functions).

A useful description of program comprehension found is: given a program in a particular language, program comprehension is a process in which the programmer uses prior knowledge about programming and information present in the program to form a dynamic, evolving model of the program which can then be applied to a task *[Good 1999]*. In this definition, 'task' is the purpose of the comprehension, in other words why the programmer is looking at the program in the first place, and what he/she wants to get out of it.

There are different theories on the process of program comprehension. A distinction can be made between top-down, bottom-up and hybrid theories. These theories give us insight of the mental models used in software comprehension.

In the early 1980's Brooks proposed one of the first models, a top-down model in which comprehension is seen as a hypothesis verification process. Sometimes simply hearing the name of a program will trigger the construction of a hypothesis about the program. This functional hypothesis leads to the development of a tree of hypotheses, based on domain knowledge and familiarity with similar programs.

N. Pennington has been influential in the field of program comprehension; in 1987 she published two papers of software comprehension that have frequently been quoted. She hypothesised a bottom-up theory of program comprehension *[Pennington 1987a,b]*. Her model of comprehension postulates the existence of two types of mental representation of a program: the *textbase* and the *situational model*. According to her theory first the textbase is built, based on a procedural reading of the program, and the domain model is then based on this textbase.

Pennington showed that in her experiment the programmers that achieved the highest comprehension scores "tended to think about *both* the program world and the domain world to which the program applies while studying the program". This is a clear indication of the process of building a mapping between the two worlds.

In general, problem-solving performance is affected by the way a problem is presented *[Whitley 1996]*. People frequently do not translate a given problem into a normalised internal (mental) representation that is independent of the original external presentation. Rather, external representations tend to constrain internal representations. We will investigate the relationship between internal representations and external representations in Chapter 4.

4 Restraints on mental models by notations

4.1 Empirical results

Software engineering and programming languages are areas that historically have suffered from a lack of empirical validation *[Whitley 1996]*. Part of this situation comes from the difficulties of carrying out studies in projects the size of typical software projects. Another factor is the resistance of computer researchers to adopt techniques outside the traditional set of computer science techniques. However, software engineering is, or should be, an empirical field. Any software engineering technique or theory of programming is judged by whether it is cost-effective or not: this means results will have to be measured.

One of the most thorough summaries of empirical studies using VPLs to date is *[Whitley 1996]*. From this paper it becomes clear that there is still a lot of work to be done in empirically researching the effect of VPLs. “Given the unproven status of visual programming, empirical studies could make a major contribution to visual programming efforts. This is so much the case that the empirical evidence problem warrants being called the biggest open problem in visual programming research.”. Also Good describes problems she had in comparing results across studies, a sign that the empirical research in this area needs a lot of work *[Good 1999]*.

Still, the empirical results known at this point have established some results, and led to new theory.

In a study on the influence of mathematical background on the easiness of learning how to program, Van der Veer (1986) found that there was a difference in the ability of “alphas” and “betas” in learning a language. They hoped that a specific (diagram like) Visual Language would make the difference between those two groups smaller. It turned out not to be so: both in the learning phase and in the additional problems alphas needed more time for the problem solving process. The structured diagram language presented some extra difficulties for alpha students (compared with the betas).

On the other hand, they found that using this VPL, some frightening aspects for programming that are present with the use of the traditional abstract-verbal programming languages disappeared for the alphas. Furthermore the results of the interviews showed that alpha’s also *felt* they learned more. With the regular language experiment they said they felt like they only learned a trick to solve an exercise, but less a structures method to solve other ones. In the interview after the experiment with a VPL the alphas felt more confident.

4.2 “Natural” solutions

The study also investigated which control structures novices seem to prefer. An established principle of programming is that a program (algorithm) can be described using three basic blocks: sequence, selection and iteration. A result in this study was that novice programmers seemed to prefer the use of dyadic selection, even when the semantic structure of the problem domain and the semantic connotation of hints might be expected to cause an alternative solution. Apparently for novices the dyadic selection within an iteration is the “natural” semantic solution for the type of problems investigated in this study.

This *[Van der Veer 1986]* study, although small, showed that some semantic solutions are preferred (at least by novices) when writing programs. This was also found in a study by Good (1999). In her paper she described results of studies about software comprehension where procedural languages were used. These studies suggested that the program comprehension could be characterised essentially in terms of control flow and low-level operations (at least in the initial stages of comprehension). However, it is quite possible that the results were influenced by that the fact that procedural languages were used in this study. Therefore she also did an investigation using Prolog, a declarative language. It was shown that even using Prolog there still seems to exist a procedural bias. Once again, her research was focused on the learning of programming by novices. *[Petre 1996a]* showed that a particular programming paradigm does not necessary affect the way experts comprehend a program.

4.3 Match-mismatch conjecture

On interpreting existing programs, Gilmore and Green (1984) showed that rule-based paradigms are better than imperative paradigms for the extracting task, but vice versa for interfering. They proposed the 'match-mismatch conjecture', according to which all notations contain many types of information and every notation highlights some kinds at the cost of obscuring other kinds. Extracting information about a program is correspondingly easy when the information matches the notation and hard when there is a mismatch.

Various other investigations seem to support this claim. In *[Moström 1998]* research was done on the writing of so-called parallel software, software that will run on multiple computers at the same time. Novice users working with a parallel language had better results than while using an procedural language, although the last one was expected to be considered more difficult. So also in this experiment there seems to be a 'natural fit' between a representation and a specific problem.

Notations are probably not good in an absolute sense. Green writes "A notation is never absolutely good, but only in relation to certain tasks" *[Green 1996]*. In other literature, the related concept of "cognitive fit" has been investigated *[Good 1999]*. This is described as a developing property of a model for problem solving. A match between the information highlighted in the representation and task leads to the use of similarly matching problem solving process (and a matching mental representation).

All in all it becomes apparent from literature that there is a connection between a physical representation and a mental presentation in the comprehension process. If the task the user has doesn't match the problem representation, there will be a mis-fit, and mental transformations will be necessary for the user in order to solve the task.

An important research topic is whether or not this match-mismatch conjecture also holds in graphical languages. An experiment by Chattratichar (2001) supports this premise: the match-mismatch hypothesis also holds when using graphical languages.

5 Theory on Visual Languages

It is intuitively obvious that it is possible for people to derive semantic information from visual displays. However, to see if visual languages will be more efficient for users it is important to find out how exactly this works. It is interesting to notice that many researchers seem to think that graphics will be good help in programming;

5.1 Cognitive processes

Programming requires understanding the program that is already there. The mental representation of a program is at a higher level than pure code. This is valid not only when the programmer writes his program, but also while working on somebody else's. It therefore requires extracting and interfering information. Larkin and Simon break the problem into three cognitive skills: search, recognition and inference [*Larkin 1987]*.

5.2 Perception research

The problems of perception have always been one the central research topics of cognitive psychology, and this research has resulted in some well-established principles. For example in guidance on such matters as choosing colours and pixel sizes for computer displays.

These early experimental observations now rest on firm theoretical ground derived from neurophysiological research. As a result, we can discuss colour perception or feature detection performance using response characteristics of the retina and optic nerve. Marr, Pogio and others integrated these low level processes in a computational theory of visual perception.

The Gestalt theorists of the early 20th century produced a description of perceptual experience that has scarcely been challenged, and is still very relevant to the design of visual displays. They described how humans restructure perception in ways that make it unified and coherent. What makes this work still relevant to visualisation is that the theory is formulated in terms of *laws of organisation* that we apply to perceptual experience. These laws deal with the way humans interpret lines, boundary closure, proximity etc. and can thus be used to explain why some displays work better than others.

5.3 Dimensions

What is the difference between the textual source code for a piece of software and a pictorial visualisation of the software? Many computer scientists point to a fundamental difference in information content between the two: text is essentially linear and one-dimensional, while pictures are two-dimensional. Moving pictures, in the case of an animated visualisation, adds a time dimension to the two dimensions of the pictorial plane. Virtual reality systems add a further dimension by persuading our binocular perceptual apparatus to see a three dimensional world.

So how much information can really be presented in a multi-dimensional software visualisation? The first point to note is that even text is not really one-dimensional. A language compiler[2] may read source code in a linear fashion, but programmers certainly do not write it in that way – they write a bit, go back and see what they have done, and modify it until it looks right. After it is complete, human readers of source code often take note of secondary notation such as indentation, vertical and horizontal patterns, distribution of white space, etc. *[Green 1996]*

Control nesting and execution sequence are reasonably sensible variables to assign to two of the scarce physical dimensions. They must both be represented by ordered values, so the dimensions colour or shape would not be useful candidates. This is how most text-based languages work today: the indentation (from left-to-right) is used to indicate control nesting; the order of the lines (top-down) is the execution sequence. That this is influenced by the way (western) people read a book seems obvious.

Visualisation technologies can incorporate more information by presenting third or fourth continuous dimensions, but each comes at sufficient cost. Therefore we must be quite careful about how we allocate variables to dimensions. That cost is both computational, and perceptual – the third dimension of perception is limited in that we can only see one side of solid objects, and the time dimension even more so because the perception of temporal events is by definition transient.

Visualisation design means trying to fit many variables into a few dimensions. A fundamental problem of Software Visualisation is of course that software is one of the most complex intellectual domains and there are far more variables than dimensions.

5.4 Attractiveness of graphics

Myers (1990) offers a typical description of the attraction of graphics: "…graphics tends to be a higher-level description of the desired actions (often de-emphasising issues of syntax and providing a higher level of abstraction) and may therefore make the programming task easier even for professional programmers."

Petre (1996b) describes an interview where programmers of all levels were interviewed about their preferences among graphical and textual representations. Most important remarks about graphical representations were:

• richer; providing more information with less clutter and in less space

• providing the 'gestalt' effect: providing an overview; making structure more visible; clearer

• having a higher level of abstraction, a closer mapping to the problem domain

• more fun; less formal

Richness: Graphical representations appear potentially richer than textual ones. The experts observed in *[Petre 1996b]* believed that the graphical properties and the secondary notation made electronics schematics and comparable programming notations richer than any textual equivalent.

[2] Compiler: computer program that translates human-readable code into machine language.

'Gestalt' overview: Graphics may benefit from a 'gestalt' response, an informative impression of the whole that provides insight into the structure. Because people need help in grasping complex structures quickly, this supposed 'gestalt' attribute makes graphics appealing. Yet in the reading comprehension experiments, programmers did not recognise structural similarities among the graphical representations and often found it difficult to compare two graphical programs. In contrast, those programmers who did notice structural similarities among the programs presented were all looking at the *textual* representations *[Petre 1996b]*. This is an important fact, see also below in §6.

Mapping to the domain: Graphical representations appear to offer potential for 'externalising the objects of thought' -- for providing a more direct mapping between internal and external representations by providing representations close to the domain level that make structures and relationships accessible. If relations among objects are visually or spatially grasped, it might be easier to derive a mental model of a system structure from a graphical representation than from a textual one. However, this can be also a bit dangerous, as there is the risk of losing too much abstraction: sometimes for novice users the temptation to draw the actual object, instead of an abstraction of it is too big. This happens for example with designing chips: instead of drawing the diagram of the connections between components, the novice user sometimes draws an actual chip (where they take into account location, crossings of wires etc). This is undesirable as it is more complicated to understand and more difficult to change later.

Fun: Graphical representations may just seem more fun. The importance of likeability should not be underestimated; it can be a strong motivator. In general, affect may be as important as effectiveness. The *illusion* of accessibility may be more important than the reality. This relates of to the effect noticed in the paper *[Van der Veer 1986]*. Although the gap between alphas and betas didn't disappear the alphas felt more comfortable learning to program using a VPL. They also felt that they understood the principle of programming better, instead of only learning a trick (see §4.1).

6 Multiple views

An obvious solution to use when it turns out that a single representation doesn't provide for all the tasks a user (programmer) has, is to use multiple views. For example, a programming tool could present an algorithm as program text and in a visual form at the same time. However, as *[Petre 1996b]* states: "Presenting two representations is not automatically better than one. In fact, all else being equal, one representation is likely to be better than two". As mentioned in §5.4 it is not always easy to recognize structure between different representations. Therefore 'switching' between the representations can be quite difficult for the programmer. Besides that there are also practical reasons to prefer a single representation, because that allows the whole computer screen to be used and thus decreases the 'window-view' problem.

All this means that a multiple-view solution is no silver bullet; it won't be a good solution in all situations. However, multiple-view tools might be valuable if a single representation doesn't provide all the information the programmer needs. It is therefore unfortunate that most research tends to be focussed on VPLs in a single view: the studies would look at the effect of using diagrams etc *in isolation*. It could be interesting if more research was done into multiple-view VPLs, especially since current programming tools (which are often just Visual Environment, see §2) come with various different views. Some modern ones even allow editing of all these views, so blending the difference between Visual Environments and true VPLs with multiple views. Research in this area started in the 1990's *[Sajaniemi 1999, Meyers 1993]* and will in my opinion influence the way modern tools work in the next years.

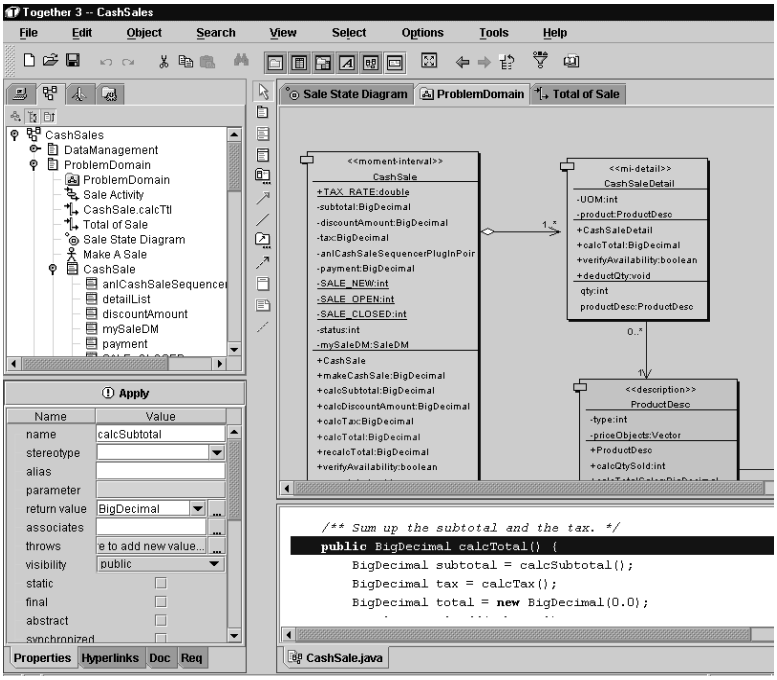

Figure 1: Between a Visual Environment and a VPL: A modern round-trip engineering tool. The programmer can edit both the graphical display, as well as the text-based code. Changes in one view will be updated in the other one.

7 Conclusion and further research

In §4.2 we found that novice users have a bias towards certain programming constructions. They tend to think in a procedural way, and prefer some semantic constructions (dyadic selection for example) above others. However, for some tasks alternative representations work better. There are two factors explaining the efficiency of alternative representations. First, people perform better when presented with information arranged in a consistent and organized manner. Second, the more efficient representations tend to be the ones that make information explicit. These two principles apply to all forms of representation, textual as well as visual *[Whitley 1996]*.

There is a lot to win in researching visual languages. Research has shown that for some tasks a visual language can be useful. However, the language influences the mental model a user creates. This has lead to the 'match-mismatch' theory by Greene: extracting information about a program is easy when the information matches the notation and hard when there is a mismatch. Thus, there is a relationship between the way information is represented, and the mental model users make. If the representation doesn't match the task the user has, comprehension will be more difficult because a mental transformation is necessary (see §4.3). This implies that in a well-designed VPL the task the programmer has should match the representation the VPL offers.

The most important point is that there are many different aspects on writing a program, and different notations support different aspects of a language or as task. The 'match-mismatch' theory holds, so is suggested by research by *[Chattratichart 2001]*, also for Visual Languages. The task human-computer-interaction scientists face is to find notations that match with specific kinds of information. Sometimes a very detailed view is required, sometimes a very broad. Information sometimes needs to be inferred, sometimes directly extracted.

For practical uses - that is for programming environments that are made today - it is important to keep in mind that research in the field of VPLs has basically just started. Therefore I would make the suggestion that for a production-type programming environment a VPL would not be good enough yet. However, it definitely might serve as an addition to the 'good-old' text-based programs. In other words: there should be room for multiple views in such tools. In the classification of Shu (see §2) this would in fact not be a VPL, but a Visual Environment. However, there is a trend towards a merger between these two: pure VPLs that don't use text are unpractical and more important, the Visual Environments get closer to real VPLs. A big difference between the two, the 'read-only' aspect of a Visual Environment is already disappearing in the most modern tools; the so-called round-trip engineering environments (see figure 1, page 12). The complications that arise while using multiple-views should be addressed too: the relationship between true VPLs and text-based programs should be investigated and not only VPLs in isolation.

Designers of these modern tools should definitely take a close look to the field of VPLs. Although the field is young, and empirical results are lacking, the use of graphics as alternative views of a program can benefit the user.

8 References

Chattratichar, J, Kuljis, J. (2001) Some Evidence for Graphical Readership, Paradigm Preferenece, and the Match-Mismatch Conjecture in Graphical Programs, Department of Information Systems and Computing, Brunel University, UK. Paper presented at the 10th Annual Workshop Doctoral Consortium Programme April 2001 of the Psychology of Programming Interest Group

Chang, S. K. (1987), Visual Languages: A Tutorial and Survey, Published in Visualization in Programming, 5th Interdisciplinary Workshop in Informatics and Psychology Schörding, Austria, May 20-23, 1986. Lecture Notes in Computer Science, Springer-Verlag, ISBN 3-540-18507-0

Good, J. (1999) Programming Paradigms, Information Types and Graphical Representations: Empirical Investigations of Novice Program Comprehension. Ph. D. Thesis University of Edinburgh, 1999

Green, T. R. G., Petre, M., (1992) When Visual Programs are Harder to Read than Textual Programs. In G.C. van der Veer, M.J. Tauber, S. Bagnarola, and M. Antavolits (Eds.) Human-Computer Interaction: Tasks and Organisation, Proc. ECCE-6 (6th European Conference on Cognitive Ergonomics). CUD, Rome

Green, T. R. G., Petre, M., (1996) Usability Analysis of Visual Programming Environments: a 'cognitive dimensions' framework. Journal of Visual Languages and Computing , 7(2)

Dix, A., Finlay, J., Abowd, G. and Beale, R., (1998) Human-Computer Interaction 2nd edition, Prentice Hall, London, 1998; ISBN 0-13-239864-8

Larkin J. H., Simon H. A., (1987) Why a diagram is (sometimes) worth ten thousand words, Cognitive Science, 11

Meyers, S.D. (1993) Representing Software Systems In Multiple-View Development Environments, Department of Computer Science, Brown University, Providence Rhode Island 02912, Ph. D. Thesis.

Myers, B,.A, (1990), Taxonomies of Visual programming and Program Visualization. Journal of Visual Languages and Computing, 1

Moström, J. E., Carr, D.A (1998), Programming Paradigms and Program Comprehension by Novices, Luleå University of Technology, Sweden. Paper presented at the 10th Annual Workshop Doctoral Consortium Programme 5th January 1998 of the Psychology of Programming Interest Group

Pandey, R. K., Burnett, M. M. (1993) Is It Easier to Write Matrix Manipulation Programs Visually or Textually? An Empirical Study. Proceedings of the IEEE Symposium on Visual Languages, Bergen, Norway. ISBN 0-81863-970-9

Pennington N. (1987), Comprehension Strategies in Programming. In G. M. Olson, S. Sheppard, E. Soloway (Eds), Empirical Studies of Programmers: Second Workshop, Ablex

Pennington N. (1987a) Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. Cognitive Psychology, 19

Petre, M. (1996a) Programming Paradigms and Culture: Implications of Expert Practice. In M. Woodman (Ed), Programming Language Choice: Practice and Experience, International Thomson Computer Press

Petre, M., Blackwell, A., Green, T.R.G., (1996b), Cognitive Questions in Software Visualisation, In J. Stasko, J. Domingue, B. Price, and M. Brown (Eds), Software Visualisation: Programming as a Multi-Media Experience. MIT Press, 1997

Rogers, Y., Rutherford, A., Bibby, P. A. (1992). Models in the mind: Theory, Perspective and Application. Londen: Academic Press. ISBN 0-12-592970-6

Sajaniemi, J. Getting Rid of the Single Notation Paradigm with Multiple Views, Paper presented at the 11th Annual Workshop Doctoral Contsortium Programme 4th January 1999 of the Psychology of Programming Interest Group

Shu, Nan. C. (1988) Visual Programming. Van Nostrand Rheinhold, New York. ISBN 0-442-28014-9

Van der Veer, G.C., Van Beek, J., Cruts G. A. N. (1986), Learning Structured Diagrams – Effect of Mathmatical Background, Instruction, and Problem Semantics. Dept. Psychology, Cognitive Psychology Division, Vrije Universiteit Amsterdam. Published in Visualization in Programming, 5th Interdisciplinary Workshop in Informatics and Psychology Schörding, Austria, May 20-23, 1986. Lecture Notes in Computer Science, Springer-Verlag, ISBN 3-540-18507-0

Visual Programming Environments, Paradigms and System, Ephraim P. Glinert, ISBN 0-8186-5973-4

Whitley, K. N. (1996) Visual Programming Languages and Empirical Evidence For and Against, Department of Computer Science, Vanderbilt University. Journal of Visual Languages and Computing 8

Acknowledgements

I would like to thank Alexandra Tabova and Petra Eussen with their assistance in reviewing drafts of this paper. My appreciation also goes to Objects By Design who let me use a picture of their website (http://www.objectsbydesign.com/tools/tj/tj_intro.html) as an illustration.