Cognitive Questions in Software Visualisation

M. Petre[1], A. F. Blackwell[2] and T. R. G. Green[2]

June 1996. To appear in 'Software Visualization: Programming as a Multi-Media Experience', edited by Stasko J., Domingue, J., Price, B., and. Brown, M. MIT Press, probably 1997

Introduction

Software visualisation is nifty stuff; but is it the powerful cognitive tool it is often assumed to be? This chapter attempts to moderate the understandable enthusiasm for software visualisation and to raise some of the questions for which the discipline doesn't yet have answers. The chapter is structured as a list of questions with discussion. The questions are not a comprehensive analysis of cognitive challenges in software visualisation. Rather, the chapter attempts to provide a list sufficiently provocative to give designers pause, in order:

a) to establish that good software visualisation isn't simply a matter of mimicking paper-based tasks or doing what is technically easy—and certainly isn't 'solved' yet; but also

b) even simple tools can improve software comprehension, if they're the right ones.

1. What is visualization suitable for?

Are all aspects of software amenable to visualisation? Software visualisation is trying to find simplicity in a complex artefact (e. g. , thousand-line code), to produce a selective representation of a complex abstraction. What makes it difficult is that it is presenting information artefacts, not physical objects (or even the pseudo-physical objects which superficial interface analogies suggest). It is presenting a logical construction, rather than a physical one. In contrast to the visualisation of a mathematical function, most software visualisation has no simple generating function. The complexity lies not just in the information to be visualised, but in the information's context of use; software visualisation is used to show different sorts of information (relating to source code, to data, to data structures, to execution) for different uses and with different aims, among them:

i) presenting large data sets:

Like the data visualisation in domains such as radiography and meteorology, some software visualisation involves the presentation of large, mainly homogeneous data, e. g. , program performance data. This form of visualisation attempts to make data available for interpretation by presenting a data picture and so capitalising on perceptual effects, such as foreground/background effects, pop-out, detection of discontinuities. And yet, although these

[1] Centre for Informatics Education Research, Dept of Computing and Mathematics, Open University, Milton Keynes, MK7 6AA, UK
[2] MRC Applied Psychology Unit, 15 Chaucer Road, Cambridge CB2 2EF, UK

are basic mechanisms of perception, experience in other domains makes clear that 'reading' the perceptual qualities of data visualisations is a matter of considerable skill.

ii) demonstrating the virtual machine:

Some software visualisation attempts to create a visible, dynamic, machine model, often relying on kinematic or mechanical metaphors. This form of visualisation is intended to make the behaviour of the interpreted program visible, often for pedagogic purposes.

iii) changing the perspective:

Some visualisation tries to bring large software engineering problems within the scope of a single view, so that a user can grasp particular tasks or functions in relation to the software as a whole. They aim for the sort of perspective change one might characterise as "the helicopter over the landscape": putting one's locale in perspective as part of a much larger landscape by providing a viewpoint high above ground level, and giving an opportunity to locate other features and functions in the landscape. This is an attempt at complexity control, helping to keep a large problem "in a single head" by visualising the overall structure and providing some assistance for navigating or traversing that structure.

iv) display-based reasoning:

Some software visualisation provides an alternative formalism, not a data picture or a machine model, but a regular, symbolic re-presentation of software with a new emphasis, in order to support an otherwise ill-supported style of reasoning. The notion is that an effective display can ease the user's reasoning; the likelihood is that having an effective display *changes* the user's tactics, if not the nature of the user's reasoning. The display becomes a focus for reasoning, for example by replacing some internal representations with external ones, and hence allowing the user to use different tactics in finding, recalling, examining, or comparing information (Davies, 1993, 1996).

This may be a continuum of uses, but what is required by a presentation of a large homogeneous data set may be very different from what is required by a formalism for depicting machine behaviour. We don't yet know if the differences amount to different judgements on a consistent set of criteria, or if different criteria apply. We must beware assuming that the space is adequately defined by a simple model of perception and perceptual inference. We must know more about uses, about tasks within uses, and about representations for tasks.

Simply restated, the question of suitability is multi-faceted:

- What sort of visualisation — depiction, description, presentation, abstraction —
- is suitable for which information
- and which uses
- by which users?

2. Does Visualisation Mean Pictures?

How do we distinguish software visualisation systems from other ways of viewing software? The obvious definition is that visualisation systems present software in graphical form. This position is expressed by Myers as follows:

> "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". (Myers 1990)

There are cases, of course, where this definition might not be literally true. What if the program has been specified diagramatically, using a visual programming language? Most software visualisations do depict the behaviour of textual programs, but they would be equally valid if used to depict the same algorithm implemented in a visual language. There are a number of such situations where the relationship between text and graphics in visualisation is not obvious. Consider some of the following cases:

- A textual program is re-expressed in a graphical form – representing identifiers by icons, and syntactic constructs by connections or containment.
- In an even more likely situation, the identifiers are still represented by their textual names, but are connected by graphical elements to indicate syntax, as in a flow chart.
- In some kinds of flowchart, lines or other graphical annotations are superimposed on a program listing in order to clarify the program structure.
- Advanced "pretty printers" clarify program structure by automatically indenting lines, or by changing the type styles of program elements.

The first of these cases obviously meets the criteria of Myers' definition (and would be recognised by most of us as a software visualisation system), but we feel increasingly uncomfortable in referring to the other cases as software visualisation. In an extreme example, imagine a system that takes a complicated spaghetti-like visual program, and makes it easy to understand by formatting it as nicely structured text, introducing meaningful words to refer to the arbitrary icons, and describing the relationships between the parts using a constrained and well-defined vocabulary of keywords. This seems to be a useful visualisation, but is actually the reverse of the commonly accepted use of the term.

There is no longer a clear dividing line between graphics and typography in the print media. It is also increasingly hard to distinguish between graphical and textual programming systems (to an extent that puzzles Visual Basic users when they are excluded from a visual programming forum). Is there a dividing line after which we can confidently say that something is not a visualisation system? Rather than defining the amount of graphics that constitutes a visualisation, it is better to encourage a "broad church" based on characteristics that are independent of the difficult divide between text and graphics. This allows us to recognise the importance of facilities such as the 'code view' that was included in the Balsa system. The code view simply presented pretty-printed program text, dynamically displayed and highlighted as the program executed. In an environment concentrating on animated graphics, these textual views were still found to be helpful to a wide range of users (Brown & Sedgewick, 1985).

The essence of visualisation from this point of view is the provision of an alternative representation or alternative emphasis within a representation. Alternative representations include changes in structure as well as changes in modality, while alternative emphases include the explicit presentation of temporal behaviour through facilities such as instantaneous animation or historical integration. In a mechanical domain, Mayer and Gallini (1990) found evidence for improved understanding of devices when explanatory text and illustrations were integrated by the inclusion of simple information about causal state changes.

As an interesting exercise in visualisation design, try asking how many of the following objectives could be achieved without using graphics:

- Emphasise data structure rather than algorithm or vice-versa.
- Illustrate causal relationships.
- Create a profile of resource utilisation.
- Search through an execution trace.
- Contrast two space-packing heuristics.

All of these classic "visualisations" can be communicated in textual terms, although this becomes more difficult in the case of spatial application domains. Multiple representations and emphases are characteristic of text as much as they are of pictures. The nature of visualisation should therefore reside in the structure of these transformations in multi-modal presentation far more than from the presentation mode itself. This could be seen as an appeal for would-be visualisers to respect Knuth's ideal of literacy in programming as much as the novelty of new pictorial conventions (Knuth, 1992). In either case literacy, whether verbal or visual (Dondis 1973) relies far more on understanding of structure and dynamics than it does on presentation mode.

3. Is SV a way into 'the expert mind' or a way out of our usual world view?

What is visualisation for? What is the relationship of software visualisation to mental representation? In order to consider these questions, we shall discuss a 'space of intentions' for software visualisations. This is simply a device for discussion; the distinctions are only valued briefly as a means of examining the relationship between internal and external representations. The distinctions are in any case not clear; they overlap in this discussion, and visualisations produced to serve one intention might well serve many categories of intention. The section is in two halves, one concerned with experts, the other with non-experts.

Consider what software visualisation for use by experts might be intended to do.

1. To externalise images of thought, to "get things out of the head". This intention is to produce visualisations closely related to the expert's mental images, whether externalisation's of actual mental images, or illustrations of mental images, closely related but not actually what the expert sees in his 'mind's eye'.

   These could be thought of as external memory extensions, to 'download' and 'set aside' information whose importance is no longer immediate. 'Setting aside' concepts or portions of a problem that have been thought about is one of the techniques that experts use to keep problems tractable.

2. To provide a focus for communication between experts. This intention is to produce visualisations close enough to the expert's thinking to facilitate discussion of problems and solutions with peers. These could be the externalisations intended by (1), they could derive from local representational idiom, from standard representations in the domain or in familiar tools, or they could simply be recognisably useful representations of focal information (e. g. , concepts or phenomena). This is the least distinct intention, because nearly any representation can assist communication, particularly among experts.

3. To provide tools for thinking. This intention is to produce visualisations that complement and supplement experts' thinking, rather than mimic their internal representations.

a) These 'supplementary representations' can display information that needs to be taken into account but which the expert would prefer not to hold internally. These might be a sort of 'external memory', allowing experts to free their working memory by using display-based reasoning strategies.

b) The 'supplementary representations' might provide alternative views of the problem, perhaps casting the information in terms of another programming paradigm. Experts are observed to shift deliberately among paradigms, styles of reasoning, and representations in order to consider different aspects of a design. But shifting paradigm—particularly to an unfamiliar paradigm—may not be easy; the visualisation may provide a route to thinking outside the expert's usual world view.

c) The 'supplementary representations' can provide a direct visual mapping to the problem, so that graphically sensible things in the representation map to semantically sensible things in the algorithm or problem. Occasionally the algorithm can be mapped onto a physical arrangement (e. g. , binary chop into a sorted list), but this sort of physical analogy is a special case only. This intention is to make it possible to manipulate or reason about graphical transformations and infer semantic transformations from that information (e. g. , a geometric relationship that relates to a complex equation relationship in an algorithm, such as an informative plot of coefficients in a complex equation, when coefficients that fall inside a depicted circle have special importance—an example from visualisations useful in filter design). For example, Bauer and Johnson-Laird (1993) found that use of an appropriate graphical representation could improve performance in logical deduction.

d) The 'supplementary representations' can provide navigational assistance, helping the expert to keep track of the relationship of a sub-problem to the whole and to other parts. It can also remove the need to add new labels to keep track of relationships (Larkin and Simon, 1987)

4. To harness the computer as a collaborator in problem solving. For problems without an apparent solution, experts often explore solutions in tandem with the machine, by programming it to enact a simplification, to analyse a sub-problem, or to process a partial solution (e. g. , finding repetitive patterns in a signal). This intention is to make visible the activity of the computer in these tasks, so that the expert can relate what it does to the larger problem, in order to determine whether it contributes to a final solution. This can encompass a partial automation of the interpretation of a visualisation, alerting the programmer to events or emerging phenomena in the visualisation domain, not just in the problem domain. As a visualisation of machine activity, these need have little to do with the expert's internal imagery, although the expert must be able to relate the computer's activity to his understanding of the problem.

The discussion so far assumes that visualizations for experts reflect or augment a rich and largely appropriate mental imagery. For the sake of this discussion, we assume that non-experts have a less developed or less appropriate internal imagery, and that therefore visualisations for non-experts often have an instructive role. What might the intentions for software visualisation for use by non-experts include?

5. To provide a glimpse of an expert's imagery. To 'look inside' the expert mind and possibly make expert insight visible to the non-expert. (cf. (1)) This intention is to help non-experts 'see' information more expertly and hence inform them.

6. To provide a visualisation of an expert's reasoning process. This intention is to make explicit thinking that experts do invisibly or 'effortlessly', and hence to provide a sort of 'scaffolding' for non-experts when their own 'vision' is immature. Of course a visualisation tool won't turn a novice into an expert, but it can teach by making portions of expert reasoning explicit and accessible. Although based on how experts behave, this intention is to support non-experts in developing their own reasoning, for example by providing additional ways of

'seeing', implying different ways of reasoning, indicating which information is of importance in which contexts (cf. (3b)).

The modest version of this intention provides tools for novices that are consistent with expert behaviour, so that as novices become more experienced they also grow into effective, professional behaviour. The more directive version biases the tools to a particular, rationalised style of reasoning, providing a visualisation of a chosen paradigm, in order to influence users' thinking.

7. To provide a glimpse of how programs or computers work, a visualisation of execution. (See Section 1, ii). This is an important role in environments designed for novices. Du Boulay, O'Shea and Monk (1981) show the importance of simplicity and visibility of function (the 'glass box') in environments designed for novices, while Jones (1993) extends their work, showing with reference to several types of visualisation that novices also need to have views of state, procedure and function.

8. To provide additional ways of seeing, depicting behaviour, or apprehending relationships non-experts find difficult to envision. In doing so, the visualization provides a sort of 'imagery' training, showing what information is useful and offering ways of 'seeing' it. Like (6), this intention is a sort of scaffolding, supporting acquisition of skills by providing structure in the visualisation, although this intention need not refer to expert behaviour, but rather focuses on the needs of less experienced users. This too is an important role in environments for novices (Cañas et al. , 1994).

This discussion, although one hopes instructive, is uncomfortable, because the distinctions among intentions are not crisp, and because distinctions among intentions need not imply distinctions among resultant visualisations. A visualisation that allows an expert to extend his own thinking is likely to be a suitable basis for communication with his peers (of whom he is his closest example), and may well promote insight by making something visible to a non-expert. Nevertheless, although some visualisations may be generic, many are not, and it is not appropriate to *assume* that what comes out of an expert's mind will somehow magically launch a non-expert into expert reasoning. Visualisations are not useful without insight about their use, about their significance and limitations. Further, one needs insight about one visualisation's relationship to other visualisations, understanding about how they do and do not map to each other.

4. Why are experts often resistant to other people's visualisations?

One of the recurrent themes in the psychology of programming is comparison of novice and expert behaviour, and often detection of differences between them. So it is unsurprising that one of the questions in software visualisation is whether visualisations designed for experts will be suitable for novices, and vice versa. One way to slice into this question is to consider why experts create tools for themselves, and to study those visualisations.

Many of the most familiar visualisation tools made by 'other people' than the programmer, such as debuggers or execution models, visualise aspects of a single paradigm. These can be particularly useful to novices in developing a reasoning model of the virtual machine, say, or to experts searching for particular phenomena. But the things that one can visualise 'clearly' within one domain tend not to be what experts build their own tools for. Experts working close to the edge of what's possible have a greater tendency to break both their applications and their development systems than novices, and hence they are likely to have explored the paradigm, to have found its boundaries and problems.

Most errors that consume expert time are not within-paradigm, but looking below the paradigm at internal operation in order to debug abnormal behaviour of entities that are correct within the paradigm, i. e. , unknown side-effects, rather than syntactic errors. So visualisations within-paradigm are uninformative.

As a result, experts have a tendency to create a visualisation for a particular problem (e. g. , specific data structure)—even if it will never be useful for another problem. One of us (MP) has observed that experts have a capacious willingness to grow a new, specific tool or to modify an old, specific one.

What puts experts sufficiently in need of visualisation tools to invest in them, even to the extent of creating them?

- taking over someone else's ('half-baked') program
- really big problems
- mixed-technology problems or problems that cross domains (i. e. , different computers, different operating systems, different programs — hence often different representations)

Many of the most difficult problems—and those most difficult to reason about—involve the concatenation of tools or programs, e. g. , outputting probe results into files (or pipes) to another tool, rather than just displaying results on screen. Hence, experts spend time modelling the behaviour of partially known objects, e. g. , this computer must talk to that remote peripheral, which we don't know much about. So software visualisation is a way of coping with unknowns, with the unseen: visualisation of partially visible, often heterogeneous, systems. This needn't mean that the visualisations are necessarily complex or elaborate. Software comprehension, although itself complex, may still be ameliorated by simple tools, if they're the right ones. The advantage of simple tools is that they are less likely to carry the 'baggage' of a particular paradigm-and so they might assist the expert in finding things outside the paradigm.

Why don't they use other people's visualisations more often? In part because of the specificity of the problems they most need to visualise. But in part because visualisation tools are not very useful without insight about their use and significance. Experts know that they need to understand what tools are for, what their limitations are, when not to use them-and they can judge the cost of acquiring this meta-knowledge as too prohibitive.

What do experts demand from visualisation tools? MP's observations indicate that experts want:

- control: Experts want to control their focus, and hence the focus of the visualisation. They want to be able to 'set-aside' concepts or portions of a problem that they know about or that are otherwise outside their focus.
- scale: Visualisations must cope with 'real-sized' problems. This emphasises the importance of navigation tools and search tools.
- speed: Visualisations must be sufficiently responsive. Experts' visualisations tend to be functional rather than pretty; experts will not buy beauty at the expense of efficiency.
- truth: Visualisations must maintain accuracy, and experts resist 'someone else's simplification'. Experts express frustration with novice environments which do not preserve accuracy or which misleadingly simplify and so fail to reflect the 'real world'.

In short, what experts want from visualisation tools is pretty much what they want from conventional programming languages (Petre, 1991).

5. Are visualizations trying to provide a representation that is more abstract, or more concrete?

Is visualisation a magic window into the invisible world of abstractions? Many people feel intuitively that software is hard to understand because it is so mathematical – it deals with symbols rather than things. If it is abstraction that makes software (and mathematics) difficult, perhaps we can make it easier to understand by presenting it in concrete pictorial form, as when an arithmetic exercise asks us to subtract two apples from five apples.

On the other hand, the creation and interpretation of pictures is itself an exercise in abstraction (Arnheim, 1970). The first pictures we create as children (or even as adults) are not copies of images from our visual fields, but constructions made from the standard symbolic elements in the childhood pictorial vocabulary – heads, arms, houses and smiling suns. Many adults still find it difficult to create realistic drawings because they maintain the habit of using cartoon-like symbols (Edwards 1979).

These two views of the relationship between vision and abstraction are difficult to reconcile, and we must ask whether we can form any coherent view of the nature of abstraction in visualisation. It is tempting to believe that we can, if only because we have established the convention of direct manipulation interfaces in which we treat images on a computer screen as if they were physical objects. At the same time it is undeniable that the pictures we see when viewing a software visualisation have little actual resemblance to concrete objects in the physical world.

Perhaps we must explicitly acknowledge the dual nature of visualisation. Any visualisation relies on a pictorial language, and language is fundamentally about abstraction; like icons and Greek letters, words are arbitrary symbols that have a conventional correspondence to classes of objects in the world and relations between those classes. The main function of language is to provide us with a means of manipulating these abstractions. At the same time as providing a language, visualisations also provide a model – a depiction of a device whose behaviour we can envisage in terms of the constraints and causality of the physical world.

Is it possible for a visualisation environment to satisfy both of these requirements equally well? The pictorial language function provides a way of moving away from conventional software representations in order to express more general principles – it takes the concrete elements of a specific software situation, and expresses them in general, abstract, terms so that they are easier to assimilate. The device model function provides a way of visualising behaviour with the help of localised influence, attention and causation – by making the abstract more concrete, it is easier to apprehend.

The possible functions of abstraction in working with visualisations are difficult to ascertain. The relative priorities of these functions are also difficult to establish. It is hard to say in general what level of abstraction is suitable for software workers. Abstractions come at different granularities, and the probability that a given abstraction is relevant to a problem often seems inverse to its complexity. Should we therefore aim to create a vocabulary of abstraction, or a mechanism for hiding it? Should we facilitate (or even require) the creation of new abstractions, or is visualisation primarily a communicative device for conveying one person's abstract view of a particular situation?

6. What model are we representing?

Myers (1990) makes a necessary distinction between visualisations that represent the code, the data, or the algorithm. But that is not enough. We should also distinguish between representations

of external structure versus representations of cognitive structure, and representations of structure versus representations of purpose or expectation.

Early experiments such as that by Soloway and Ehrlich (1984), now well-known, showed that Pascal novices learn to think in terms of 'plan' structures, which are groupings of statements that together achieve a stated goal or purpose. The classic example is the 'Running Total' plan. Although this research has been around for over a decade, very few attempts at software visualisation have attempted to display representations of cognitive structures. Exceptions are Bonar and Liffick's (1990) Bridge system and more recently Ebrahimi (1992); Bowles et al. (1994) describe a Prolog system for novices that represents the code as 'techniques', which are very similar to the plan concept. This last has received some degree of empirical evaluation by Ormerod and Ball (1996), indicating that the idea is good but that the problems of interacting with such high-level representations (especially, editing them) are not yet fully resolved.

Although the systems mentioned above were all program construction environments, rather than tools for visualising existing software, they point a direction. And indeed, recent work by Rist and Bevemyr (1990; see also Rist, 1996) has shown that static analysis can extract these 'plan' structures from Pascal programs algorithmically, which leads to the possibility of plan visualisation tools.

If the 'plan' structure is indeed how programmers conceptualise programs, then visualising code in those terms would be a great improvement over visualising it in lower-level structures of individual statements or control structures. The difference would be comparable to working with a drawing package instead of a paint package – the drawing package groups pixels into lines, circles, etc. , and allows those structures to be picked out and if necessary manipulated individually.

But, *are* these structures cognitive? Certainly, the original claim was that 'programming plans' were a 'natural' cognitive construct; but another view is that 'plans' are built only because Pascal is a procedural language, in which the programmer's purposes are diluted and dispersed here and there. The structures extracted by Rist's analysis are, at bottom, based on dataflow. Perhaps all that the Pascal environments are doing is helping users understand dataflow, providing a re-representation. Maybe, then, a dataflow language would not need to be conceptualised in any different terms than the ones it already exists in.

The contrary viewpoint is that programmers only build 'plan' representations of software for certain particular tasks (Bellamy and Gilmore, 1990). If they are doing other tasks, then they will use a different representation, such as control-flow representation. If that analysis is correct, then a dataflow language would need to be parsed for control flow and for sequential information. That would correspond to the Prolog techniques editor, which categorises simple Prolog functions in terms of types of recursion etc.

Whichever of those views may turn out to be nearer the truth, the fact is that at present there is a shortage of software visualisations explicitly built around cognitive representations. And outside the domain of code, there is not even much idea what a cognitive representation would look like.

7. What kind of tasks are we supporting?

It is a truism of HCI that systems should be designed to support the tasks the user needs/wants to do. But descriptions of visualisation systems rarely specify any particular task that they are intended to support.

What would be required to support the task of algorithm comprehension? To understand an algorithm means building a mapping between the user's conception of events and entities in the

program, and the user's conception of events and entities in the work domain. Both of those domains will be represented in cognitive terms, not in external terms (for example, the program domain might be represented in terms of the plan structures described in the previous section). Thus Robertson et al. (1990) showed that programmers understand code by searching through it in various complex ways, focusing on small functionally coherent groupings, presumably plan-like. Even more telling, Pennington (1987) took a sample of professional programmers and showed that of those programmers, the ones who achieved 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 'cross-referencing' strategy was sharply revealed in the high scorers, much less apparent in the low scores - a clear indication of the process of building a mapping between the two worlds.

Thus supporting users in understanding an algorithm requires more than illustrating the primitive code: it requires illustrating the algorithm is a useful form. Two axioms about information representation are:

> (1) Finding and using information is easier when the form of the information sought has a 'cognitive fit' with the form of the information presented.

Vessey (1990) shows that the mental representation formed in problem solving depends on the (external) representations of both the problem and the problem-solving task. When these two external representations are different in form, the problem-solver has to do extra work to translate between them; whereas "matching representation to task leads to the use of similar, and therefore consistent, problem-solving processes, and hence to the formation of a consistent mental representation. There will be no need to transform the mental representation to accommodate the use of different processes to extract information from the problem representation and to solve the problem. " (p. 221)

> (2) If the presentation is designed to highlight some kinds of information, then it is likely to obscure other kinds.

For example, a programming language using a procedural model highlights the sequential information but makes the declarative structure harder to extract. This point was made a long time ago (e. g. Green et al. 1981). By axiom (1), the kinds that are not highlighted will be harder to obtain.

Perhaps these axioms explain why animation systems using code steppers, such as the derivatives of Balsa (Brown and Sedgewick, 1985), have not yet proved stunningly successful methods to support the comprehension of algorithms (Fix and Sriram, 1996). For an animation constrains their users to view the code *in the order of execution*, whereas which has a poor cognitive fit with the plan-and-goal structures that users are trying to extract from the code. The animation highlights sequence, which is the wrong information for algorithm comprehension, and obscures the plan structures.

If we are going to make use of these two axioms effectively, we need to ask ourselves:

- What tasks do programmers actually do? (Pennington and Grabowski (1990) have made a helpful start on this one.)
- How do we provide support for those tasks?
- How will those tasks change when visualisation systems are available? For change they will: Caroll et al. (1991) give numerous and persuasive illustrations of the 'task-artifact' cycle in which improving the artifact, to support existing tasks, necessarily creates new tasks and new possibilities for the user.

8. What do we know about perception, anyway?

When we talk about visualisation, we make a natural assumption that these pictures we create can help people learn things. It is intuitively obvious that people can derive semantic information from visual displays. It is also reasonable to say that it is easier to derive certain kinds of information from some displays than others. It is not at all clear, however, what the brain is doing with the light that falls on our retinas in order to create this information. If only we knew that, both the justification and the strategy for visualisation research would be far more certain.

The problems of perception have always been one of the central research topics of cognitive psychology, and this research has resulted in some reasonably well-established principles. Human factors experts such as Wickens have been demonstrating for years that this research is relevant to information displays, and there are a number of useful results that are germane to software visualisation. Some of the theories that we now use originated over 100 years ago, and it is worthwhile to ask to what degree more recent cognitive theories of perception have been assimilated into the conventional wisdom of HCI, and thence to the software visualisation community.

Nineteenth century experiments in psychophysics formed the foundation of experimental psychology, addressing such questions as how different two levels of brightness must be before we perceive them as different. The idea that the human visual system has fundamental limits is now familiar, and human factors specialist can produce tables of empirical data for guidance on such matters as choosing colours or pixel sizes for computer displays.

These early experimental observations now rest on firm theoretical ground derived from neurophysiological research (Gordon, 1989). As a result, we can discuss colour perception or feature detection performance by way of the response characteristics of the retina and optic nerve. These low level processes were integrated by Marr (1982) in a computational theory of visual perception, in terms that are very accessible to computer scientists. Marr's theory mainly concerns perception of 3-D objects, but his description of the way that 3-D information is derived from a "2 $^{1}/_{2}$-D" visual field should be applied to describe the way that, for example, texture in a computer display contributes to our interpretation of the objects depicted. Ullman's computational model of how boundaries are interpreted (Ullman 1984) is also directly applicable to 2-D displays.

Marr's theory, as with others in cognitive psychology, treats cognitive functions as if they were a series of filters that operate on the "raw data" arriving in our eyes, eventually turning it into information. The situation is rather more complex, however – the world does not arrive at our optic nerve as if it were a photograph. Gibson's theory of ecological optics, later elaborated by Neisser (1976), described the way in which we shift our attention on the basis of the structure of the environment, creating a cognitive map that we use to interact with the world. The term he used to describe these possibilities for interaction, *affordances*, has been popularised by Norman for use in HCI (Norman, 1988), and is now widely used to describe the design of direct manipulation interfaces. Designers of more complex visual interactions should perhaps read behind Norman's interpretation to see how metaphorical worlds can be presented in a display.

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 the way in which we restructure our perception in ways that make it unified and coherent (Benjafield, 1992). What makes their work still relevant to visualisation is that they formulated this theory in terms of *laws of organisation* that we apply to perceptual experience. These laws can be used to explain why some displays work better than others: when they provide for the way that we interpret continuous lines, boundary closure, proximity and so on. Wickens (1995) has taken these as his inspiration in defining *perceptual proximity* within displays, so that the elements of the display can be made to correspond to the relationships between the elements of a task. Wickens is most concerned with the aviation context, but his work should certainly be considered as a basis for

visualisation design. Palmer and Rock's (1994) theory of uniform connectedness provides an alternative development of Gestalt principles.

The way that meaning is derived from symbols is of course properly the domain of semioticians and semanticians. Kosslyn (1989) has carried out an analysis in cognitive terms of the symbol systems used to create graphs, and much of this work would be directly applicable to typical software visualisations. Oberlander (1996) has similarly carried out a semantic analysis of graphical design notations that extends the Cognitive Dimensions of Green (1989) and Green and Petre (in press), particularly the use of secondary notation by designers (Petre and Green, 1992). He considers the ways in which certain arrangements of graphical elements carry with them further implications. This analysis is derived from Grice's maxims of communication – a convoluted piece of wiring, for example, is expected to mean something because the reader assumes that there must have been a reason why the designer violated the Gricean maxim of brevity. Software visualisations are a context of communication between the designer and user of the visualisation, so these pragmatic principles are just as relevant as the lower-level perceptual elements that have already been assimilated by the HCI community.

9. What if there aren't enough 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 (Blackwell, 1996). Text is essentially linear and one-dimensional, while pictures are two dimensional. Moving pictures, in the case of an animated visualisation, add a time dimension to the two dimensions of the pictorial plane. Virtual reality systems add a further dimension by persuading our binocular perceptual apparatus that it is seeing a three dimensional world.

Computer scientists have regularly assumed that information content increases exponentially with dimensionality. This is certainly true of on-line storage requirements; the pixels of digitised images are slowly clogging the Internet, and on-line digitised video is only seen regularly by those with luxurious access to bandwidth and storage. To a computer scientist, then, the real-time video channel of the human eye supplies a kind of super network connection from computer to brain. An ACM report on visualisation to the NSF was explicitly criticised for making this argument, on the grounds that the information contained within a real-world situation is very different from the information required to make any kind of image of that situation (Lewis, 1991).

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 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 (Green, Bellamy & Parker 1987). After it is complete, human readers of source code take note of secondary notation (Green and Petre, in press) such as indentation, vertical and horizontal patterns, distribution of white space, and the many creative uses of ASCII text that can be seen within comment delimiters.

According to Bertin (1981), the art of presenting graphical information lies largely in assigning the available dimensions of the plane to the independent scales and categories of information that are to be presented. Physical dimensions are a scarce commodity in our universe, however, and Bertin identifies only eight variables that can be utilised in a printed graphic image. Apart from the X and Y dimensions, there are two more ordered variables: the density of ink used to make marks, and the size of the marks. There are also four categorical dimensions: texture, colour, orientation and shape of marks. Source code is just one way of making this assignment in a software visualisation. In source code, groups of shapes (ASCII characters) are given specific meanings and organised

according to standard conventions – typically sequence of execution along the Y dimension, and control nesting in the X dimension. Some program editors also assign colour or density values to differentiate between information that is relevant to the compiler or relevant to the programmer.

Control nesting and execution sequence are reasonably sensible variables to assign to two of our scarce physical dimensions. They must both be represented by ordered values, so colour or shape could not easily be substituted for them. Execution sequence is related to flow of time, which is a continuous variable that is by convention assigned to either X or Y – so users expect it. Finally, both control nesting and execution sequence are fundamental properties of computing machines, so this conventional depiction of software is sufficiently general to visualise a program written in any language equivalent to a Turing machine. As with most properties of Turing machines, however, this generality is also at a sufficiently low level to make it uninteresting – but it has already allocated the two most versatile variables of the graphical plane.

Visualisation technologies can incorporate more information by presenting third or fourth continuous dimensions, but each comes at sufficient cost that 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 perception of temporal events is necessarily transient. How irritating that software is one of the most complex intellectual domains, and that there are hundreds of independent variables that we must juggle in our heads and would prefer to assign to one of those four dimensions to be visualised.

Visualisation design means trying to fit many variables into a few dimensions. There are far more variables than dimensions, so we should encourage an attitude of dimensional restraint in software visualisation. Rather than rushing to the latest technology in order to use up all the dimensions of human perception, a more economical line of research might be to discover how much can be achieved by the subtle application of two.

Some popular software visualisations are particularly unsubtle, in that they use the most valuable plane dimensions to portray topographic domain information such as the map used by a travelling salesman, or the floor of a bin-packing warehouse. The allocation of the plane should be the most careful decision taken in designing a new visualisation, but simply using it to portray the problem domain is a 'no-brainer'. Such a visualisation tells you little about the algorithm, and may even be deceptive – ordinary students find it so easy to solve the travelling salesman problem when given a visual presentation of the problem that they may not realise why the choice of algorithm is significant (MacGregor and Ormerod, 1996).

The information that is available in a situation is ultimately dependent on how that situation is decoded by the observer (Lewis, 1991). Dimensional restraint will be best discovered by avoiding unsubtle algorithms with built-in continuous variables, and devising new coding strategies for informational challenges. This means breaking out of the shallow (but entertaining) concerns of building sexy virtual reality systems and thinking a lot harder about what to do with the two dimensional display devices that are already in front of our eyes.

10. Are representations good for everyone? What is the importance of individual skill and variation?

The unifying aim of software visualisation is to make information apparent — ideally to make selected, elusive information 'visible', even obvious. By doing so, one hopes that users will more readily be able to find, recognise and interpret the information they need. The hope is that software visualisation can capitalise on native human perception to make relevant information

salient. But we must interpret what we perceive, and what we perceive is influenced by what we've learned about interpretation.

Readership skills:
Does visibility lead to effective interpretation? There is increasing evidence, from sources ranging from data interpretation to visual programming, that visibility doesn't guarantee effective interpretation, particularly for novices. Interpreting representations is an acquired skill (Petre, 1995; Petre & Green, 1993). Arnheim (1969) makes this argument in the context of interpreting paintings; the issue is more critical in the precise interpretation of software visualisations. Novice 'readers' tend to lack search and inspection strategies, and tend to be distracted by surface features. Worse, 'visibility' is confused with significance, and confused with relevance to the immediate task, so that novices can be misled by mis-cueing. Expert 'readers' are better attuned to the ways in which underlying structures are cued in the surface layout, and they are better able to recognise what is relevant and to disregard what is irrelevant.

Sweepingly stated, visibility is not solely a characteristic of a representation; visibility is (in part) in the eye of the beholder. Experts are distinguished by their acquired ability to 'see': to perceive as salient the information relevant to a task, and to choose what *not* to see—to ignore inessentials.

Graphical cultures:
Those cues to inspection and navigation through visualisations that do exist rely on what Petre and Green (1992) have called 'secondary notation' — layout and presentation conventions that are not formally defined, and often not explicit. Readers are informed by their knowledge of context and culture. Relying on cues governed by conventions implicit in a culture, means making use of information 'invisible' to the uninitiated. This may make visualisations difficult to access. Little of graphical culture is codified, and some conventions vary among sub-cultures.

Presentation skills:
The hope in visualisation is that expertise in presentation can compensate for inexpertness in readership. It may be that disciplined and insightful use of conventions and 'secondary notation' can direct attention to relevant information, assuming that readers can somehow be made to 'see' the cues. This entails identifying and capturing what is most important, and guiding the reader by associating perceptual cues with this carefully selected information. This also means providing support in the environment for the acquisition of readership skills and of the culture, through explicit conventions and tools.

'Cognitive style':
It has become an axiom in the psychology of programming that "individuals differ". Results will differ among individual users as well as individual visualisations. Whether or not one accepts the notion that there are recognisable 'cognitive styles' that influence an individual's reading and reasoning (e. g. , making graphical representations appear more or less congenial than textual ones), one must recognise that individual readership is profoundly influenced by experience. Knowing what to expect, where to look, and what to look for—the cognitive components of an inspection—affects individual strategies, and increasing expertise is evidently reflected in changes of perceptual strategy. Hence, visualisation is acutely vulnerable to weaknesses in individual expressive, perceptive, and interpretive skill.

11. When are two representations better than one?

Software visualisation systems usually present two or more representations of the same information – e. g. an algorithm presented as program text and in a visual form. Presenting two representations is not automatically better than one. In fact, all else being equal, one representation is likely to be

better than two. A single representation uses less screen space, avoids problems of switching from one representation to the other and of finding the right place in each one, avoids problems of working which bit of one is equivalent to which bit of the other, and so on. And since many problems can be solved with minimal resources, using a single representation is bound to remain a popular option for some conditions.

Programmers would rely on a single representation if it were sufficient, but when one is not, then users must learn how to do "inter-representational reasoning", working between the two representations, comprehending the correspondence between them, and keeping track of both at the same time. There is not much research on the cognitive difficulties involved; most research on how diagrams help reasoning is about how diagrams can be used in isolation – i. e. how one representation can be used, not about how two or more can be used together, and what effects result.

So why use two or more? And why should either of them be graphical? We shall consider how programmers might use multiple representations.

(i) The simple case

In the simple case, the representations will be used separately. For some of the user's tasks, or (more likely in expert practice) some part of a task, a single representation may be sufficient, and so one will be all that is used. Different sub-tasks may be suited by different representations, and so the user may switch representations with tasks. Different problems are solved with different representations. Experts often behave in this way, switching among representations (and among programming paradigms) in order to focus on particular aspects or sub-tasks of a problem (Petre, 1996).

(ii) Multiple identical representations

Multiple representations can be useful to the reader just by providing different views on a conceptual space, even where the representation format of each view is identical. Engineers and architects give detailed information about three dimensional objects by the use of multiple views or elevations. The same principle has been shown to be applicable to more abstract spaces by Shah and Carpenter (1995), who found that even experienced readers of graphs find it difficult to appreciate the interaction of three variables when they are presented by several lines on a single, conventional line graph.

(iii) Bridging representations

One representation may be more suitable for expressing the problem, but less congenial to the user (see Section 10), and so a second representation helps the user to reason about the first. For example, many people have difficulty interpreting contour maps, but are assisted by a 3-D rendering, which helps them to relate the terrain they see to the contour line depiction. The bridging representation may assist the user to extend search and reasoning strategies appropriately. For example, if the user has a search strategy that works for one representation, a different representation may either facilitate indexing, or force exploration.

(iv) Heterogeneous inference

A more interesting case is what has been called 'heterogeneous inference' (Stenning and Oberlander, 1995), where two forms of representation need to be used simultaneously for best results. The user needs the alternation between representations in order to encompass the whole of the problem.

Stenning, Cox and Oberlander (1995) have further developed these ideas in the context of a pedagogic environment for learning first-order predicate calculus called Hyperproof (Barwise and Etchemendy, 1994). In Hyperproof, a simple domain of geometric solids is displayed, and users learn how to construct proofs about that world. Proofs can be constructed using standard logic calculus or the users can take advantage of the graphical display. The argument put forward by

Stenning and Oberlander is that the two forms of representation (logical and graphical, in their system) are complementary as regards two of the most important cognitive processes required in reasoning. These processes are abstraction, the construction of expressions that apply to more than a single instance, and mental modelling, the hypothesized process by which people reason. Abstraction is easy in the logical calculus, which makes it easy to say "All blue cubes are bigger than all red spheres", but less easy to say "That blue cube there is an exception"; abstraction is harder in the graphical representation, which makes it easy to refer to particular instances, such as a big red sphere, but difficult – even with special graphical notations used in Hyperproof - to handle facts about all solids of a certain type.

And it is exactly that limited power to represent abstractions that is, they claim, the advantage of graphical representations for constructing mental models. A mental model, in the sense they use it (derived from the work of Byrne and Johnson-Laird, 1990) is an internal representation of the states of the world, actual and possible. Reasoning about a process in the world requires setting up a mental model of each of the possible starting states. The more states that have to be represented, the harder the reasoning will be. When the external representation is graphical, its limited power of expressing abstractions will make it easier to construct the required set of mental models, for the simple reason that the external 'sentences' will unpack into fewer different possibilities and hence fewer mental models will need to be constructed.

(v) Useful awkwardness

Finally, there is the possibility that a second form of representation can supply not just different information, but also a 'useful awkwardness'. Having to make the mental transference between representations (and possibly between paradigms) forces reflection beyond the boundaries and details of the first representation and an anticipation of correspondences in the second. The deeper level of cognitive processing can reveal glitches that might otherwise have been missed. Far too little is known about this process. The translation can sometimes be provocative, sometimes obstructive, but we cannot at present predict which.

12. How can we reconcile warring conventions? And how do conventions arise?

When the user of a software visualisation interprets what he or she sees, that interpretation is structured and informed by a broad general knowledge of graphical conventions, many of which do not need to be made explicit in order to be understood. Some graphical conventions are so firmly established that it does not occur to us to question that for example larger areas represent higher numbers or that time flows from left to right. The conventions established by illustrators for printed visualisations of quantitative information been collected and systematised in classic texts such as Tufte, (1983) and Bertin (1981).

Despite the guidance of such sytematisers, only some of the knowledge that allows us to interpret graphical information displays is independent of cultural context (Tversky 1991). This cultural dependency is certainly not recognised by context-free mathematical analyses which treat different linear dimensions as fundamentally undistinguished, and able to be assigned arbitrarily.

An example of conflict between mathematical and metaphorical conventions lies in the case where altitude is an independent variable in a data set, as often occurs in meteorological data. The independent variable is normally placed on the X axis, but meteorologists quite reasonably preserve the vertical metaphor by placing altitude on the Y axis. This results in a visual presentation that can only be readily interpreted by a specialist audience (Gattis & Holyoak, 1996).

Such conflicts easily arise in a domain such as meteorology, where a notation has evolved over time through a number of media, but metaphorical conflicts are already apparent between older software visualisations. Consider the case of a push-down stack, which is often expressed in terms of the plate-stacker analogy, where another plate is added to the top of the pile, pushing down the older items *below* it. This is in contrast to the normal illustration of a machine stack, which is conventionally illustrated in a memory space with higher addresses (where the stack starts) at the top of the page. The older items in the stack therefore appear *above* the new items at the "top".

The two conventions for visualising stacks naturally arise from requirements to emphasise different aspects of the situation. There are many sources of such alternative conventions that will be encountered by the users of larger scale visualisations. These include textbook illustrations which are often designed to elucidate by analogy, design notations which are increasingly promoted by CASE tool vendors, and the metaphors of technical jargon, which have evolved from regular usage in order to provide efficient communication between experts. Users may also come to a visualisation with their own images adapted from unrelated domains (Ford 1994). Where such an original image is useful, it can eventually be absorbed as a new convention. These intuitive choices of new representations are unfortunately the only way in which conventions appear to arise, despite the fact that we know enough to apply principles of notational design (Scaife & Rogers, in press).

There could be a useful harvest of originality to be reaped from this diversity, but different conventions are often incompatible. It is more likely that once they become widespread the habits of their use will result in complete entrenchment, as has become the case with the QWERTY keyboard. Where such a convention is owned by a commercial entity, the advantages of such entrenchment are those of a captive user base, meaning that CASE tool vendors, for example, are tempted to promote notational elements that can be drawn only using their own products (at one time, there were suspicions that the 'clouds' of the Booch object-oriented design method were prescribed with such a purpose in mind).

In the rest of this chapter we have discussed the ways in which the conventions used for visualisation have an influence on the way we approach cognitive tasks when using them. This influence has already been noted in more general problem solving, where there is evidence that students produce incorrect solutions to certain kinds of problem if they do not follow established notational conventions, or are unclear about the correspondence between the convention they choose and their problem domain (Cox & Brna, in press).

This discussion of the ways in which conventions arise and then compete does not hold much hope for an improved situation in future. The combination of cultural dependency, conflicting metaphors, commercial interests and unsupported intuitions mean that software visualisation can apparently look forward to further battle between conventions, with wider popularisation of specific visualisations simply widening the battlefield. The people with most influence on the future are those designing new visualisations today, who should be conscious of the conflict between their desire for unfettered originality and the value of widely accepted conventions.

13. Why do people like graphical widgets, anyhow?

Given the growing evidence that graphical representations can be harder work and produce poorer performance than textual ones, why are they so appealing? It may be simply that graphical representations provide an alternative to text. Myers (1990) offers a typical description of the attraction: ". . . 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. " (p. 100)

In a recent survey, programmers of all levels were interviewed about their preferences among graphical and textual representations. Graphical representations were described as:

- 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 accessible; easier to understand; faster to grasp
- more comprehensible
- more memorable
- more fun
- 'non-symbolic'; less formal

Richness: Graphical representations appear potentially richer than textual ones. The experts observed by Petre and Green (1996) believed that the graphical properties and the secondary notation made electronics schematics and comparable programming notations richer than any textual equivalent and argued that our experimental tasks did not exploit this richness.

'Gestalt' overview: Graphics may benefit from a 'gestalt' response, an informative impression of the whole which provides insight into the structure. Because people need help in grasping complex structures quickly, this purported '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.

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. Kosslyn (1978) and Rohr (1986) suggest that if relations among objects are visually or spatially grasped, it is easier to derive a mental model of a system structure from a graphical representation than from a textual one. But meeting that potential is a challenge. Experience in digital electronics warns that a typical novice error when learning to draw 'schematics') logical representations of the circuit) is too 'literal' a transcription of the domain, a failure to abstract; novices often reflect the eventual physical layout rather than the logical layout. In effect, they draw a picture of the artefact, rather than depict the structure of the solution.

Accessibility and comprehensibility: It may be that an analog representation *appears* more accessible than a descriptive one—a notation incorporating pictures may seem less daunting than one comprising abstract symbols. A representation that exploits perceptual cueing takes advantage of the abilities of the human visual system. The analog quality appeals to mundane experience, making the notation appear less esoteric. Cynically, this is a variation of the 'Cobol effect' familiar in the history of programming languages: because the vocabulary (in this case the component shapes) looks familiar and ordinary, novices believe they can understand the program. Yet graphical representations can take longer to read and understand, and they are often misunderstood by novices, who cannot 'see' the available cues.

Fun: Graphical representations may just seem more fun or gratifying; the fact that secondary notation is 'outside the rules' allows the programmer more freedom to 'play around' with layout. The overheads of planning layouts may not matter if there is satisfaction simply in the tinkering required.

The importance of sheer likeability should not be underestimated; it can be a compelling motivator. In general, affect may be as important as effectiveness. The *illusion* of accessibility may be more important than the reality.

14. Can I take a version to bed?

The first generation of packaged software application included on-line help systems which raised windows on the screen that were not only written in a minute font, but gave users no chance to compare information in one section with information in another. The help files couldn't be printed out and perused off-line, so working the application in anger was a frustrating experience in which the user's immediate plan of action was continually interrupted by forays into an impoverished information retrieval system.

This illustrates the problems of information that cannot be printed out and taken away to be used, when, and how, it is convenient. Users may want to know that information in order to prepare a teaching course, for instance; or they may want to be just lie in bed and peruse it until it has soaked into their heads.

There is a moral to these examples: do not presume upon your user's contexts. They know when, why and how they want to use the information; you will never guess in advance. What does this tell us about software visualisation? At present we only know how to design for one context. Many existing environments are tied to one particular task, such as code inspection, exploratory programming or teaching.

To see how important context can be, consider some other examples of information appliances that can be used in bed. Mobile phone manufacturers, desperate for new sales, invented pagers to create a different context of use for their technology. This was also a benefit to users, because pagers made telephone interchange asynchronous. Sony invented the Walkman to change the context of use of sound reproduction. Software visualisation builders are in danger of designing a tool that is not available in the context in which it is most needed, since we know from observation that people reason about programs in non-standard environments. There is a whole culture of professional programmers that solve their problems over pizza, in bed, or in the shower, and we ignore this fact at out peril.

Many programmers still prefer to use physical printout during code program comprehension and debugging, not just dynamic screens. Possibly this trend will be changing, with the rise of object-oriented programming (which is non-linear in a way that makes hard copy difficult to follow) and component assembly programming, but we believe that as soon as adequate tools are available for presenting useful linearised views of OO programs, they will once again be taken home to brood over, make notes on, and spill coffee on.

Conclusion

Each of the authors has contributed three gnomic pearls:

It’s not all sewn up;

don’t assume too much;

there is no panacea.

You can't highlight everything;

think what each feature is for;

the mental representation matter.

Neuropsychologists have still not located the programming centre of the brain;

real programs are bigger (and deeper and messier) than you think;

language generates 90% of the pain, but only does 10% of the work.

And finally, a question for the reader: Why has there been no sequel to 'Sorting out sorting'?

REFERENCES

Arnheim, R. (1969) *Visual Thinking*. 1st ed. : University of California Press; 2nd ed. : London: Faber and Faber.

Barwise, J. and Etchemendy, J. (1994) *Hyperproof*. CSLI Lecture Notes. Chicago: Chicago University Press.

Bauer, M. I and Johnson-Laird, P. N. (1993) How diagrams can improve reasoning. *Psychological Science* 4(6) 372-378.

Bellamy, R. K. E. and Gilmore, D. J. (1990) Programming plans: internal or external structures. In K. J. Gilhooly, M. T. G. Keane, R. H. Logie and G. Erdos (Eds.) *Lines of Thinking: Reflections on the Psychology of Thought*. (Vol 1.) Wiley.

Benjafield, J. G. (1992). *Cognition*. Prentice Hall.

Bertin, J. (1981). *Graphics and Graphic Information Processing*. (Tr. W. J. Berg & P. Scott) Berlin: Walter de Gruyter

Blackwell, A. F. (1996). Metacognitive Theories of Visual Programming: What do we think we are doing? To appear in *Proc. IEEE Workshop on Visual Languages, VL'96*

Bonar, J. and Liffick, B. W. (1990) A visual programming language for novices. In S. -K. Chang (Ed.) *Principles of Visual Programming Systems*. Prentice-Hall.

Bowles, A. W. , Robertson, D. , Vasconcelos, W. W. , Vargas-Vera, M. and Bental, D. (1994) Applying Prolog programming techniques. *Int Journal of Human-Computer Studies*, 41(3), 329-350.

Brown, M. H. & Sedgewick, R. (1985). Techniques for algorithm animation. *IEEE Software*, January 1985, 28-39.

Byrne, R. M. J and Johnson-Laird, P. N. (1990) Models and deductive reasoning. In *Lines of Thinking,* Eds. K J Gilhooly, M T G Keane, R H Logie and G Erdos.

Cañas, J. J. , Bajo, M. T. and Gonzalvo, P. (1994) Mental models and computer programming. *Int. J. Human Computer Studies* 40, 795-811.

Carroll, J. M. , Kellogg, W. A. and Rosson, M. B. (1991) The task-artifact cycle. In J. M. Carroll (Ed.) *Designing Interaction: Psychology at the Human-Computer Interface.* Cambridge: Cambridge University Press.

Cox, R. & Brna, P. (1995). Supporting the use of external representations in problem solving: The need for flexible learning environments. *Journal of Artificial Intelligence in Education*, 6(2), 239--302.

Davies, S. P. (1993). Externalising information during coding activities: Effects of expertise, environment and task. In C. R. Cook, J. C. Scholtz and J. C. Spohrer (Eds.), *Empirical Studies of Programmers*: Fifth Workshop. Norwood, NJ: Ablex Publishing Co.

Davies, S. P. (1996) Display-based problem-solving strategies in computer programming. In W. D. Gray and D. A. Boehm-Davis (Eds), *Empirical Studies of Programmers: sixth workshop*. Newark, N. J. : Ablex.

Dondis, D. A. (1973). *A Primer of Visual Literacy*. Cambridge, MA: MIT Press.

du Boulay, B. , O'Shea, T. and Monk, J. (1981) The black box inside the glass box: presenting computing concepts to novices. *Int. J. Man-Machine Studies* , 14, 237-249.

Ebrahimi, A. (1992) VCPL: a visual language for teaching and learning programming. *Journal of Visual Languages and Computing*, 3, 299-317.

Edwards, B. (1979). *Drawing on the Right Side of the Brain*. Los Angeles: J. P. Tarcher.

Fix, V. and Sriram, P. (1996) Empirical studies of algorithm animation for the selection sort. In W. D. Gray and D. A. Boehm-Davis (Eds), *Empirical Studies of Programmers: sixth workshop*. Newark, N. J. : Ablex.

Ford, L. (1993). How programmers visualise programs. In C. R. Cook, J. C. Scholtz and J. C. Spohrer (Eds.), *Empirical Studies of Programmers*: Fifth Workshop. Norwood, NJ: Ablex Publishing Co.

Gattis, M. & Holyoak, K. J. (1996). Mapping conceptual to spatial relations in visual processing. *Journal of Experimental Psychology: Learning, Memory and Cognition*, **22**(1), 231-239

Gordon, I. E. (1989). *Theories of Visual Perception*. Wiley

Gray, W. and Anderson, J. R. (1987) Change-episodes in coding: when and how do programmers change their code? In G. M. Olson, S. Sheppard and E. Soloway (Eds), *Empirical Studies of Programmers: Second Workshop*. Ablex.

Green, T. R. G. (1989) Cognitive dimensions of notations. In A. Sutcliffe and L. Macaulay (Eds.) *People and Computers V*. Cambridge University Press.

Green, T. R. G. and Petre, M. (In press) Usability analysis of visual programming environments. To appear in *J. Visual Languages and Computing.*

Green, T. R. G. , Sime, M. E. , and Fitter, M. J. (1981). The art of notation. In M. J. Coombs and J. Alty (eds.) *Computing Skills and the User Interface*. London: Academic Press.

Green, T. R. G. , Bellamy, R. K. E. & Parker, J. M. (1987). Parsing and Gnisrap: a model of device use. *Empirical Studies of Programmers: Second Workshop*. Norwood, NJ: Ablex.

Jones, A. (1993) Conceptual models of programming environments: how learners use the glass box. *Instructional Science*, 21, 473-500.

Knuth, D. E. (1992). *Literate Programming*. Stanford University: CSLI Publications.

Kosslyn, S. M. (1989). Understanding charts and graphs. *Applied Cognitive Psychology*, **3**(3), 185-226.

Kosslyn, S. M. (1978) Imagery and internal representation. In: E. Rosch and B. B. Lloyd (Eds.), *Cognition and Categorization*. Erlbaum. 227-286.

Larkin, J. H. and Simon, H. A. (1987) Why a diagram is (sometimes) worth 10,000 words. *Cognitive Science*, 11, 65-100.

Lewis, C. M. (1991). Visualization and situations. In *Situation Theory and Its Applications*. J. Barwise, J. M. Gawron, G. Plotkin & S. Tutiya (Eds): Stanford University: CSLI

MacGregor, J. N. , & Ormerod, T. C. (1996). Human performance on the travelling salesman problem. *Perception and Psychophysics*, 58(4), 527-539

Marr, D. (1982). *Vision: A computational investigation into the human representation and processing of visual information*. New York: Freeman

Mayer, R. E. & Gallini, J. K. (1990). When is an illustration worth ten thousand words? *Journal of Educational Psychology*, 82(4), 715-726.

Moher, T. G. , Mak, D. C. , Blumenthal, B. , and Leventhal, L. M. (1993) Comparing the comprehensibility of textual and graphical programs: the case of Petri nets. In: C. R. Cook, J. C. Scholtz, and J. C. Spohrer (Eds.), *Empirical Studies of Programmers: Fifth Workshop*. Ablex. 137-161.

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

Neisser, U. (1976). *Cognition and Reality: Principles and implications of cognitive psychology*. San Francisco: Freeman.

Norman, D. A. (1988) *The Psychology of Everyday Things.* New York: Basic Books.

Oberlander, J. (1996). Grice for graphics: pragmatic implicature in network diagrams. *Information Design Journal* , **8**(6), 163-179.

Ormerod, T. C. and Ball, L. J. (1996) An empirical evaluation of TEd, a techniques editor. for Prolog programming. In W. D. Gray and D. A. Boehm-Davis (Eds), *Empirical Studies of Programmers: sixth workshop*. Newark, N. J. : Ablex.

Palmer, S. & Rock, I. (1994). Rethinking perceptual organization: the role of uniform connectedness. *Psychonomic Bulletin & Review*, 1(1), 29-55.

Pennington, N. (1987) Comprehension strategies in programming. In G. M. Olson, S. Sheppard, and E. Soloway (eds.)*Empirical Studies of Programmers: Second Workshop,* Ablex, pp. 100-113.

Pennington, N. and Grabowski, B. (1990) The tasks of programming. In J. -M. Hoc, T. R. G. Green, D. J. Gilmore and R. Samurçay (Eds.) *The Psychology of Programming* Academic Press, London.

Petre, M. (1991) What experts want from programming languages. *Ergonomics* (Special Issue on Cognitive Ergonomics), 34 (8), 1113-1127.

Petre, M. (1996) Programming paradigms and culture: implications of expert practice. In: M. Woodman (Ed.), *Programming Language Choice: Practice and Experience.* Chapman and Hall.

Petre, M. (June 1995) Why looking isn't always seeing: readership skills and graphical programming. *Communications of the ACM*, 38 (6), 33-44.

Petre, M. and Green, T. R. G. (1992) Requirements of graphical notations for professional users: electronics CAD systems as a case study. *Le Travail Humain* , 55(1), 47-70

Petre, M. , and Green, T. R. G. (1993) Learning to read graphics: some evidence that "seeing" an information display is an acquired skill. *Journal of Visual Languages and Computing*, 4, 55-70. .

Rist, R. S. (1996) System structure and design. In W. D. Gray and D. A. Boehm-Davis (Eds), *Empirical Studies of Programmers: sixth workshop*. Newark, N. J. : Ablex.

Rist, R. S. and Bevemyr, J. (1991) PARE: a cognitively-based program analyzer. Unpublished MS, Dept. of Computer Science, University of Technology, Sydney, Australia.

Robertson, S. P. , Davis, E. F. , Okabe, K. and Fitz-Randolf, D. (1990) Program comprehension beyond the line. In D. Diaper, D. Gilmore, G. Cockton and B. Shackel (Eds.) *Human-Computer Interaction – INTERACT '90*. Elsevier.

Rohr, G. (1986) Using visual concepts. In: S. -K. Chang, T. Ichikawa and P. A. Ligomenides (Eds.), *Visual Languages*. Plenum Press.

Scaife, M. & Rogers, Y. (In press.) External cognition: how do graphical representations work? To appear in *Int. J. Human Computer Studies* .

Shah, P. & Carpenter, P. A. (1995). Conceptual limitations in comprehending line graphs. *Journal of Experimental Psychology: General*, **124**(1), 43-61.

Soloway, E. (1985) From problems to programs via plans: the content and structure of knowledge for introductory Lisp programming. *J. Educational Computing Research*, 1(2), 157-172.

Soloway, E. and Ehrlich, K. (1984) Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE-10, 595-609.

Stenning, K. and Oberlander, J. (1995) A cognitive theory of graphical and linguistic reasoning: logic and implementation. *Cognitive Science*, 19, 97--140.

Stenning, K. , Cox, R. , & Oberlander, J. (1995) Contrasting the cognitive effects of graphical and sentential logic teaching: reasoning, representation and individual differences. *Language and Cognitive Processes*, 10, ??--??.

Tufte, E. R. (1990) *Envisioning Information*. Connecticut: Graphics Press.

Tversky, B. , Kugelmass, S. & Winter, A. (1991). Cross-cultural and developmental trends in graphic productions. *Cognitive Psychology*, **23**, 515-557.

Ullman, S. (1984). Visual routines. *Cognition*, 18, 97-159.

Vessey, I. (1990) Cognitive fit: a theory-based analysis of the graphs versus tables literature. *Decision Sciences* 22, 219-240.

Wickens, C. D. & Carswell, C. M. (1995). The proximity compatibility principle: Its psychological foundation and relevance to display design. *Human Factors*, **37**(3), 473-494.