Development of Program Visualization Systems [1]

Aulikki Hyrskykari

Department of Computer Science
University of Tampere
P.O. Box 607
SF-33101 Tampere
Finland
E-mail: ah@cs.uta.fi

The last decade has been a very active period of designing systems for program visualization. Without proper means for describing and evaluating both existing and new systems further develoment may be delayed. Recently, several researchers have been working for creating taxonomies for program visualization systems. During the active period of system development benefits of visualization were often praised without criticism, and scientific references were rarely presented. What do we really know about usefulness of program visualization?

We first present the state of the work with the terminology and taxonomy in the dicipline. Moreover, we review the existing empirical studies on the benefits of graphical presentation of programs. We also give a reference list to the existing program visualization systems.

Keywords: Program visualization, Algorithm animation, Software visualization, Graphical presentations

1 Development span of program visualization

The creation of various visualizations of programs is almost as old of an idea as programming itself [15]. Many different graphical methods have been developed. Along with the development of graphical terminals, computer aided methods for producing some of those visualizations appeared starting from the late fifties [2, 17]. The systems used computers to automatically produce static diagrams from programs. Most of those systems illustrated program code. Automation of the visualization of the program data is a more strenuous task,

[1] Presented as an invited talk at the 2nd Czech British Symposium of Visual Aspects of Man-Machine, Systems, March 27 1993, Praha

but a few attempts to that direction were made also quite early [18, 20]. By using the means of cinematography those program illustrations were brought to life. However, not until the late seventies was the technology mature enough for producing real-time graphical illustrations of program runs. One of the cinematisations [3] produced with the assistance of computer is generally considered to have launched serious research in the area.

The eighties was a very active decade (see the reference list for the program visualization systems in the end) for the design of program visualization systems, and the development of systems is still going strong. The possibility of bringing program executions into graphical multidimensional space caused enthusiasm to the extent that the benefits of visualizations were often uncritically praised. It was often stated that human senses are more suitable for adopting visual information (e.g.[24]), or that graphical representation makes better use of human brains because it taps brain's both hemispheres [30]. Graphics was considered to be a panacea for problems varying from programming education to software project management. During the recent years the superlativism has slowly settled down [16] and there has even been some quite sharp utterances in the other direction (e.g. [12], p. 1043]). To get a bit more profound picture of what is really known of the possible usefulness of program visualizations, we will review the related empirical studies.

Before that, we will make a summary of the taxonomies developed for evaluating program visualization systems, with which we want to give the reader an impression of the current state of the dicipline. The construction of new systems has this far been somewhat uncoordinated. The terminology is just getting stable and theories and taxonomies are under development.

2 Characterising program visualization

The enclosed reference list of visualization systems contains references to the systems that have gained considerable public attention. In addition to the ones listed there must be many noteworthy visualization systems and hundreds of more simple tools and procedure libraries created around the world to assist program visualization and animation. Considering the amount of systems, there has so far been remarkably few analytical studies of the principal ideas on which the systems are built.

Until the mid-eighties the terminology associated with program visualization was obscure. The terms *visual programming* and *program visualization* were used to describe nearly anything that had something to do with computer graphics together with programs. Myers pioneered in clarifying the basic terminology by defining first the separation between visual programming and program visualization [24]:

"Visual programming refers to any system that allows the user to specify a program in a two (or more) dimensional fashion", while *"... in program visualization, the program is specified in the conventional, textual manner, and the graphics is used to illustrate some aspects of the*

program or its run-time execution."

Furthermore, Myers divided program visualizations along two axes, depending whether they illustrated code or data of the program, and whether the illustration was static or dynamic. Later, in a review article [25] of visual programming and program visualization he updated the taxonomy by adding a new element, algorithm, to the object axis with code and data.

Along with program visualization the terms *software/algorithm visualization* or *animation* is also occasionally used. When the object of the visualization is software, one emphaszises the role of the whole programming process starting from the specification of the requirements. If the emphasis of the visualization is on high level abstractions, not regarding the implementation details, we are talking about algorithm visualization. As Myers defines, the visualizations may be either static or dynamic illustrations. Dynamic visualizations change in time reflecting the state of a program in execution. The changes on the view take place instantly from one state of a program to another. The authors of program visualization systems that are able to do dynamic visualizations often claim their system to be an animation system. However, we are justified in using the term 'animation' only if smooth transitions are used in graphical changes in the visualizations, giving the spectator time to internalize the connection between the two stages. As a matter of fact, if we use Myers' taxonomy for classifying the program visualization systems, we should add animation as an additional level of dynamics (see Figure 1).

	Static	*Dynamic*	*Animated*
Code			
Data			
Algorithm			

Figure 1: *Taxonomy of program visualization systems based on the taxonomies presented by Myers. Animation is added as a third level of dynamics [24]. [25]*

Even though Myers did a significant task in stabilizing the basic terms, his taxonomy is not powerful enough to describe the rich collection of potential characteristics of program visualization systems. Individual visualizations may still well be described by using the taxonomy, but for characterizing program visualization systems a more versatile taxonomy had to be developed.

As one of the pioneers in the field, Brown characterised program visualizations along three axes ([7] p.15):*Content, Transformation* and *Persistence* (see Figure 2). The *Content*-axis describes the tightness of connection between the implementation of the program and its visualization. In other words, at the synthetic end of the dimension, we have an algorithm visualization, and the further left we proceed the more the visualization resembles program monitoring. If the value of the *Transformation* -axis is high then the visualization may be considered animation. The third axis describes *Persistence* of the displayed information. The displays on the *current* side of the dimension contain less historical information of the

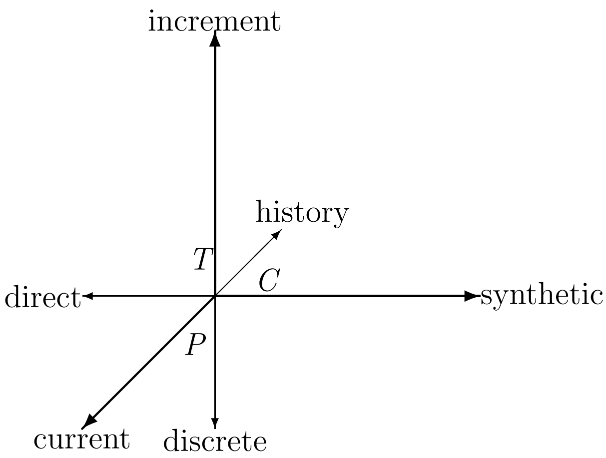

Figure 2: *Attributes of dynamic program visualization displays along three axes: T = Transformation, C = Content and P = Persistence ([7] p. 15).*

program or algorithm executed. At the other end displays constantly show the complete history of each change in the visualized object.

Two more recent papers [29, 34] suggest a characterisation scheme with four dimensions.

The four properties Roman and Cox [29] consider important when classifying program visualization systems are

- Scope,
- Abstraction,
- Specification method,
- Technique,

while the "four big As" considered by Stasko and Patterson [34] are

- Aspect,
- Abstractness,
- Animation,
- Automation.

The first two dimensions of both papers match quite well. With the first ones both describe which aspects of a program are visualised and the second ones are counterparts of Brown's content-axis.

The analysis of *Abstraction* is more profound and interesting in the paper of Roman and Cox. They have recognized five levels of abstractions in visualizing programs. Starting from the most concrete level they are: *direct, structural, synthesized, analytical* and *explanatory representations.* Surprisingly, Roman and Cox are the first ones to pay attention to a very fundamental property, namely the method of specifying the visualization (third dimension). The fact is that the flexibility and usability of a system depends to a great extent on the ease of creating new visualizations. The fourth dimension, *Technique*, characterises how the graphical representation is used to convey the information. It contains animation that is the fourth dimension of Stasko and Patterson [34]. Correspondingly, *Automation* of Stasko and Patterson falls in to *Specification method* of Roman and Cox [29] as a part of it.

The taxonomies referred above are steps to the right direction towards clear criteria for evaluation of program visualization systems, and they bring many points worth noticing up for discussion. However, they are still a bit scattered. The first work that can be considered comprehensive is the taxonomy created by Price, Baecker and Small (original ideas in [26] and revised version [27]). They have gathered and analyzed thirthy-four properties (leaf-level characteristics in the taxonomy) of program visualization systems, or software visualization (SV) systems, as they call them. The taxonomy contains many system properties that have almost totally been omitted in previous taxonomies. The characteristics have been organized into hierarchical branches of six broad categories as follows (see Figure 3):

- *Scope* contains characteristics that defining which programs a system is able to visualize.
- *Content* includes the characteristics that define what information about the software a system is able to visualize.
- *Form* defines attributes of the visualization itself.
- *Method* defines characteristics of a system facilities for specifying the visualization.
- *Interaction* includes characteristics of a system's interface.
- *Effectiveness* contains characteristics that describe how well does a system communicate information to the user.

The adaptibility of the taxonomy is tested by applying it to twelve systems. We believe that this taxonomy will evolve to be an important tool for the discipline, both for designing new systems and for evaluating the existing ones.

There are some points in the existing taxonomies that we don't find satisfying. Our point of view is, that a visualization system has two functions. On the other hand it acts as an execution environment for visualizations, and on the other hand as an specification environment for them. The characteristics of a program visualization system acting in the two roles should be recognized. This vision rises undeniably from the algorithm animation point of view, because in the systems that are able to visualize higher level abstractions, some kind of specification of the visualization is a necessity. The systems that aim at more to program monitoring type of visualization may not require any additional work for the sake of the visualization. Also the characteristics of the system and of the visualization (the

- **A: Scope**
  - a.1 Generality
    - a.1.1 Hardware
    - a.1.2 Operating system
    - a.1.3 Language
      - a.1.3.1 Concurrency
    - a.1.4 Applications
  - a.2 Scalability
    - a.2.1 Program
    - a.2.2 Applications
- **B: Content**
  - b.1 Program
    - b.1.1 Code
      - b.1.1.1 Control flow
    - b.1.2 Data
      - b.1.2.1 Data flow
  - b.2 Algorithm
    - b.2.1 Instructions
      - b.2.2.1 Control flow
    - b.2.2 Data
      - b.2.2.1 Data flow
  - b.3 Fidelity and completeness
    - b.3.1 Invasiveness
  - b.4 Data gathering time
    - b.4.1 Temporal control mapping
    - b.4.2 Visualization generation time
- **C: Form**
  - c.1 Medium
  - c.2 Presentation style
    - c.2.1 Graphical vocabulary
      - c.2.1.1 Colour
      - c.2.1.2 Dimensions
    - c.2.2 Animation
    - c.2.3.Sound
  - c.3 Granularity
  - c.4 Multiple views
  - c.5 Program synchronization
- **D: Method**
  - d.1 Visualization specification style
    - d.1.1 Intelligence
    - d.1.2 Tailorability
  - d.2 Connection technique
    - d.2.1 Code ignorance allowance
    - d.2.2 System - code coupling
- **E: Interaction**
  - e.1 Style
  - e.2 Navigation
    - e.2.1 Ellision control
    - e.2.2 Temporal control
      - e.2.2.1 Direction
      - e.2.2.2 Speed
  - e.3 Scripting facilities
- **F: Effective-ness**
  - f.1 Purpose
  - f.2 Appropriateness and clarity
  - f.3 Empirical evaluation
  - f.4 Production use

Figure 3: *Price-Baecker-Small -taxonomy of program visualization systems.*

output of the system) are often mixed. It is true that the characteristics of a visualization do also characterize the system, because the system is evaluated through the quality of the visualizations that can be produced by it. Nevertheless, this often leads to unclear situations, when you can not decide if the property refers to the system or to the visualization. That happens especially when dealing with attributes associated with the user interface. The Price-Baecker-Small -taxonomy does contain that separation explicitly, but the separation could be clearer.

3 Do visualizations really assist in understanding programs - empirical studies

Pictures do have many attributes that potentially make them an efficient means for conveying information. Attributes like the reduced need for indirect references, graphical means for expressing relationships, random access, multiple dimensions of a picture, possibility to use metaphors and possibility to change picture in time, to animate, are discussed in many papers.

Picture is more concrete than text. Visualizations make it possible to concretize abstract concepts. Concrete models may help a learner in the process of learning. They assist in connecting the new material to the old knowledge that already exists in memory ([4] p. 213-244). Mayers [22] deliberates widely upon the meaning of concrete models in learning, especially in learning programming. He concludes that there is *"clear and consistent evidence that a concrete model can have strong effect on the encoding and use of new technical information by novices"*, and that *"allowing the novices 'to see works' allows them to encode information in a more coherent and useful way".* Some of the experiments referred in [22] suggest that quality of learning is better when concrete models were used. For example, the students that were taught substraction algorithm with bundles of sticks succeeded better to apply the algorithm to more complicated tasks, than the students who were taught without concrete objects (experiment made by Brownell and Moser 1949, [22]). In addition to potential improvement of the quality of learning visualizations may make learning more appealing. In his dissertation on the attraction of computer games [21] Malone discovered that close coupling of a physical metaphor and the game structure is an important factor.

These general aspects can be used to defend graphical representations of programs, but what kind of evidence do we have? Are there any empirical studies concerning potential benefits of pictures and diagrams if used to illustrate specifically programs? There exists quite a lot of experiments which compare advantages and disadvantages arising when representing static programs either textually or graphically. The graphical representations illustrate program's structure and usually they are different forms of flowcharts. Some experiments do also test the effects of representing program's data structures with static pictures, but there are almost no tests on possible advantages gained from dynamic visualizations.

3.1 Graphical representation of the structure of a program

A well-known experiment investigating the utility of flowcharts [33] did not find evidence on the advantage of flowchart representations over program listings. The test was conducted by Shneiderman, Mayer, McKay and Heller, and they considered flowcharts useless as being only an alternative representation of the syntax of a program. They carried out five separate experiments. The number of the subjects in experiments varied from 43 to 70. They were divided into separate groups, that were provided with materials containing Fortran or/and flowchart presentations of small programs (from 24 to 147 lines). In order to assess the influence of the flowcharts each subject was asked to fill a questionnaire or assigned tasks that were designed to reveal the ability to either design, understand, debug or modify a program. None of the tests indicated statistically significant differences between the groups. Even though the experiments tested only the use of detailed flowcharts these results have been referred in textbooks on documentation and development of programs in order to judge the flowcharts complitely useless. As Scanlan points out [30] it should always be remembered that missing to find significant difference between things does not entitle to conclude that the difference does not exist. The results should only be considered inconclusive. One essential factor whose influence was ignored in the Shneiderman et al. experiment, was the use of time needed for the test. The subjects were given either all time they needed or equal amounts of time. Still, most of the potentially beneficial picture attributes are in connection with the increased human transfer rate of information, which suggests in these experiments the needed time should be measured. When using time as a measure we must remember to exclude the irrelevcant behaviour [6].

There are also some other empirical studies that support the above mentioned results that do not favour graphical representation of a program structure. Sheppard, Kruesi and Curtis [31] compared comprehension of several different forms of program descriptions including a program design language, flowcharts with different spatial arrangements and natural language. Ramsey, Atwood and Doren [28] tested the effects of the use of flowcharts during program's design phase on the quality of a program. Two groups of students were to design a two-pass assembler. The first group used a program design language (PDL), pseudocode, for the design of the first pass and flowcharts for the design of the second pass. The other group used the design methods other way round (the first pass with flowcharts and the second with PDL). The researhers judged the programs that were designed using PDL to be of better quality than the programs that were designed with the aid of flowcharts. Besides the admittedly subjective criteria used for assessing the quality of the designs, the reliability of the results suffer also from the inexplicable differences between the two groups. Quality differences between the PDL and flowchart designs were found only in the first group. The conclusions were drawn from that difference. The groups were reported to have a clear skill differencies for the benefit of the second group, which may have had some effects on the results as well. When the comprehension of a program was tested, no significant differences were found between the two forms of representations. The time given for the 20 subjects in comprehension test to study the program was 45 minutes for everyone.

However, there are several experiments that support the utility of flowcharts in programming [5, 10, 19, 30, 37]. Wright and Reid [37] studied flowchart as an aid to decision-making. They found out that flowcharts outperform textual representation when the complexity of the algorithm got high enough. With simple algorithms they did not find differences between comprehension of graphical and textual representations. Kamman [19] compared prose instructions and flowcharts for guiding telephone dialing. Speed and accuracy were increased when flowcharts were used. Brooke and Duncan [5] found the flowcharts more useful for tracing and debugging a program compared to using only listing of the program. The experiments conducted by Cunniff and Taylor [10] support the claim that time is an important factor when the usefulness of graphical and textual representation is compared. Cunniff and Taylor used a graphically formed FPL-language to test their hypothesis that comprehension of programs coded graphically would be faster and more accurate than comprehension of the same programs represented in a traditional textual language. FPL-language is functionally and instructionally a Pascal-like language, but its syntax is represented with graphical symbols and spatial arrangements. In the experiment 23 subjects were presented eight program segments in random order. The segments were presented both in Pascal and in FPL, i.e. there were sixteen program segments altogether. After each program segment the subjects were posed three questions, one about the structure of the segment, one testing comprehension of the control flow and one requiring evaluation of the value of a variable. The results strongly supported the hypothesis of the comprehension times. They indicated that graphically represented FPL program segments were comprehended more rapidly than their textually represented Pascal counterparts. Support for the other part of the hypothesis concerning accuracy also existed, but it was not so clear. An interesting hypothesis, that high visual aptitude of a subject would speed up comprehension of graphically represented segments was also tested. A moderate inverse correlation between visual aptitude and FPL reaction time was indeed found, but inverse correlation also existed between visual aptitude and Pascal reactions time, so the hypothesis was not confirmed. The findings suggested that comprehension of graphical representations was faster regardless of subject's visual aptitude.

Scanlan [30] carried out experiments comparing the use of structured flowcharts and pseudocode in understanding algorithms. He tried to avoid the flaws of the criticized Shneiderman et al. experiment [33]. The 83 subjects were provided with flowchart and pseudocod versions of three algorithms of different levels of complexity. The experiment showed that subjects made significantly fewer errors, had significantly more confidence, spent significantly less time answering questions and looked at the algorithms significantly fewer times when viewing structured flowcharts. The experiment also supported the finding of Wright and Reid [37] that the more complex the algorithm the more beneficial structured flowcharts are. Scanlan's test programs contained deep hierarchies of conditional statements, structures that we consider to be favourably disposed towards flowcharts. Despite of that we are still inclined to believe that graphical representation does in many cases have positive effects on the comprehension time.

After all these experiments reveal only little scientific information of the characteristics that may affect the usefulness of visualizations. In order to achieve more applicable result Curtis et al. [11] analyzed the properties of different software documentation formats that might have impact on the programmers' ability to adopt information. They recognized two primary dimensions for characterizing program documents: symbology and spatial arragement. The three types of symbology they found are natural language, constraint language and ideograms and the three types of spatial arrangement are sequential, branching and hierarchical arrangements. In the extensive and most carefully executed series of experiments Curtis et al. tested effectiveness of nine different documentation formats, one from each of these categories. The effecs of different formats were tested in four tasks that comprise a programmer's job: comprehension, composition, debugging, and modification of a program. The primary variable used to measure effectiveness in these tests was the response time. The most clear but also quite plausible result was that natural language was less effective form of symbology than the other two in every task. Clear difference between constraint languages and ideograms could not be found, but the perceivable tendency seemed rather favour constraint languages. However in the tasks where the control flow issues were important, the flowchart-like formats using branching arrangements appeared to be significantly better than other arrangements.

Even though the above described experiments on graphical representation of programs result into partially contradictory conclusions, it is important to gather empirical information to support the design decisions. However, the results should always be applied to the object context, unsubtle use of results should be avoided. If we want to get more generalized results the experiments should get more analytical in the direction pointed out by Curtis et al[11].

3.2 Graphical representation of the data structures of a program

Interesting experiments from the program visualization point of view are the Shneiderman's experiments on the effects of illustrated data structures on program comprehension [32]. He compared how the subjects succeeded in understanding, debugging and editing of a program if they only got a listing of a program or if they also were provided with either program's pseudocode presentation or graphical presentations of program's important data structures. The comprehension was tested with 15 questions. There were 51 subjects in the experiment. The program listing only' -group had the least right answers: their mean of correct answers was 5,06. The second group that was provided with listing and pseudocode scored 6,06: the difference to the first group was not significant. The third group that was provided with listing and data structure illustrations scored 8,47, that was significantly better result than those of the other two groups.

This experiment gives us permission to assume that the benefit was gained either from the graphical representation of data structures or maybe not from the graphical representation but generally from the description of data structures. In order to find out which was the case Shneiderman organized another experiment, in which 32 subjects were divided into four groups. All subjects got the listing of a program. The subjects from groups one and two got also descriptions of the program's structure, group one textually and group two graphically. The subjects from groups three and four got descriptions of the program's data structures, group three textually and group four graphically. Fifteen questions were asked in order to

MEAN SCORES (AND STANDARD DEVIATIONS) OF CORRECT ANSWERS	Format	
Content	Textual	Graphical
Data Structures	7.75 (4.49)	8.37 (4.40)
Control Flow	4.50 (3.50)	3.87 (4.05)

Figure 4: *Mean scores of correct answers out of 14 questions in four groups. Each group was provided with a program listing and one of the following supplements: a textual description of the program's data structures, diagram of the program's data structures, pseudocode of the program or flowchart of the program [32].*

test the subjects' capabilities for understanding, debugging and editing the program. As in his earlier experiments [33] Shneiderman ignored the time spent, and as before he did not find graphical representations beneficial (see Figure 4). However, data visualization is one part of program visualization, especially important in visualizations that are referred algorithm animation. That's why it is interesting that the differences between the groups using descriptions of a program's structure compared to the groups using descriptions of the data structures were strongly significant.

3.3 Dynamic visualization of a program

Since the early eighties dynamic visualizations have been used in teaching programming, data structures and algorithms. To give some examples, the Baecker's film [3] is still used some places in teaching sorting algorithms, in mid-eighties BALSA was systematically used in teaching algorithms [8] and the Amethyst visualization tool [23] that has been integrated to the MacGNOME programming environment [9] is being used in teaching programming in many universities and colleges in the United States. However, evaluations of the gained benefits have this far been mostly mere anecdotal observations.

The use of MacGNOME in teaching programming is assessed by Goldenson and Wang [14], but the evaluation contains just a short part that specifically evaluates the use of the visualization tool. They noticed that the usage of visualization tool depends in a great extend on the teacher's way to use it during lessons. That seems to be quite obvious. Outside the evaluation itself the teachers gave positive comments about experiences in using visualizations [14]:

"... some teachers have used the multiple program views and/or data visualizations heavily in their class demonstrations and/or assignments, and report that their students often rely

heavily on these tools as well."

One of the first reported attempts for an empirical experiment on benefits of real-time dynamic visualizations is the work of Badre et al. [1]. Eleven subjects participated the experiment and only three questions were made to test understanding of a Shellsort algorithm. The subjects were introduced with the algorithm either on a traditional lecture or they were given an animation of the algorithm. The results were inconclusive, which is not surprising considering the conciseness of the experiment.

To our knowledge the first serious attempt to empirically assess the possible advantages of algorithm animation is the experiment conducted by Stasko, Badre and Lewis in the Georgia Institute of Technology [35]. In this experiment there were 20 subjects, who were taught an implementation of the pairing heap algorithm [36]. They were divided into two groups, no one did not know the algorithm before. The first group received a textual description of the pairing heap algorithm, and the other received the same description supplemented with an opportunity to interact with an animation of the algorithm. After using 45 minutes for inspecting the material (and interacting with the animation) the subjects were asked 24 questions, which tested how they learned the algorithm. According to their nature the questions were grouped into six major sections. The hypothesis tested was that the benefit of the animation group would become more clear with questions requiring procedural knowledge, like how certain operations would affect to the data structure, than with the questions testing more declarative or factual understanding. Nor evidence for this hypothesis or any other expectations in favour of the animation group could be found. Even though the animation group succeeded better (see Figure 5) the differences were not statistically significant.

There are couple of points that diminish the value of the experiment. After analyzing the test material the researchers themselves found out that the information needed to complete the test was not easily procuceable from the given material. Another open question is the previous experience of the subjects. Were they familiar with using animations and the animation system? If not, the use of an animation not only demanded an additional effort from the subjects but they also did not have ability to benefit from the animation. Moreover the way of using an animation in the experiment is not recommendable. Usually animations are not used alone but sychronous with verbal instruction.

4 Concluding remarks

The results of the reviewed empirical experiments vary a lot, many of them are even contradictory. That arises from the fact that this kind of empirical experiments are very sensitive. The equality of test groups and test materials is very hard to obtain. It may become unbalanced with small changes in any of the three participants of the experiment: in object material used in the experiment, in characteristics of subjects used in the experiment or in the organization of the experiment.

Mean scores of correct answers and the time used for answering	Mean values	
Content	Score	Time
Text-only	11.2	41.0
Text + Animation	13.6	37.6

Figure 5: *Mean scores of correct answers out of 24 questions in two groups and of the times the subjects used answering the given questions. The first group was provided with written description of the pairing heap -algorithm, and the other group was given additionally possibility to interact with an animation of the algorithm. All subjects had 45 minutes to spend investigating the given material [35].*

A substantial share of the variations in the results derive from the nature of the visualized object. For example, if the program we choose for visualization contains deep hierarchies of conditional statements, we probably are able to create a clearer visual than a textual presentation of the program. On the other hand we certainly are able to find structures that are more easily and precisely presented with text than with graphics. From practical reasons students are often used as subjects in experiments studying programmer behaviour and it is often heavily critized [6, 11]. When assessing the effectiveness of learning this is not a problem. In spite of this relief there are many problems with choosing the subjects. Subjects' ability to utilize different presentations depends heavily on the prior education and experience. Usually the subjects have learned their original programming skills with a text based language and the effects of that can not be wiped out. Also the individual differencies are a problem in these experiments, because they easily obscure validation of the tested hypothesis. Brooks [6] notifies that the found ability differencies of subjects may imply a need for hundreds of subjects in order to obtain significant results. The experiment arrangements are usually very laborous even with small test groups, so this demand is often immpossible to fullfill. One possible solution to this problem is the use of within-subject experimental designs [11].

Most of the experiments reviewed above that did consider time as an evaluation criteria found graphical presentations better. However, Green, Petre and Bellamy [16] made also in that case an contradictory observation when they compared micro-structure comprehension of a visual dataflow language and of a text-based language. Obviously the usability of graphical representations of programs can not generally be certificated. As the matter of fact, graphical presentations as such are not valuable, but we should talk about the quality of the presentation. A poor animation of an algorithm is not beneficial; on the contrary, it may obscure understanding of the algorithm. On the other hand, with animations we can emphasize essential characteristics and thus undoubtedly create animations that significantly help in understanding the algorithm.

In any case, we think that merely the positive attitudes of students entitle the use of visualizations in teaching. A comprehensive (554 students in 42 classes) questioning executed by Scanlan [30] strongly supports the frequent informal observations of the favourable reception of program visualizations and animations. Even after the fascination of a new method has vanished animations give variety to lessons. If we consider the quality of learning the experiments suggest that concrete models help in meaningful learning, but on the other hand animations may lead to the possibility of too easy learning by imitation [**?**]. This is one of the qualitative things that should be remembered when a visualization, or animation is designed. Visualization systems are tools for designing visualizations. To give the system designers better knowledge of the desired potentials of systems they should be given information what kind of visualizations are useful. This leads us to a conclusion that we should change the question "are graphical representations beneficial?" to the question "what kind of graphical representations are beneficial?".

Acknowledgements

I wish to thank Kari-Jouko Räihä, Erkki Mäkinen and Pentti Hietala who provided valuable comments on a draft of this paper.

References

[1] Badre Albert, Beranekm Margaret, Morgan Morris J. & Stasko John T.: 'Assessing program visualization systems as instructional aids'. *Lecture Notes in Computer Science*, 602,Tomek I. (ed.), Springer, 1992, 87-99.

[2] Baecker Ronald M.: 'Experiments in on-line graphical debugging: The interrogation of complex data structures' (summary only). *Proceedings of the First Hawaii International Conference on the System Sciences.* 1968, 128-129.

[3] Baecker Ronald M.: 'Sorting out sorting'. 16 mm color, sound film, 25 minutes, presented at ACM SIGGRAPH '81, commercially available from Morgan Kaufman.

[4] Bransford John D.: *Human Cognition: Learning, Understanding and Remembering.* Wadsworth, 1979.

[5] Brooke J. B. & Duncan K. D.: 'Experimental studies of flowchart use at different stages of program debugging'. *Ergonomics.* 23(11), 1980, 1057-1091.

[6] Brooks Ruven E.: 'Studying programmer behaviour experimentally: the problem of proper methodology'. *Communications of ACM.* 23(4), 1980, 207-211.

[7] Brown Marc H.: 'Algorithm Animation'. Ph.D. Dissertation, Brown University, Department of Computer Science, Providence, RI, 1987, published also by MIT Press, 1988.

[8] Brown Marc H. & Sedgewick Robert: 'A system for algorithm animation'. *Computer Graphics.* 18(7), 1984, 177-186.

[9] Chandhok et.al.: 'Programming environments based on structure editing: The Gnome approach'.*Proceedings of the National Computer Conference 1985.* AFIPS.

[10] Cunniff Nancy & Robert Taylor: 'Graphical vs. Textual Representation: An empirical study of novices' program comprehension'. *Empirical Studies of Pprogrammers: Second Workshop.* Ablex, 1987, 114-131.

[11] Curtis Bill, Sheppard Sylvia B., Kruesi-Bailey Elisabeth, BaileyJohn & Boehm-Davis Deborah A.: 'Experimental evaluation of software documentaion formats'. *The Journal of Systems and Software.* 9, 1989, 167-207.

[12] Dijkstra Edsger W.: 'On the cruelty of really teaching computing science'. *Communications of ACM.* 32(12), 1989, 1398-1404.

[13] Gilmore D. J. & Smith H. T.: 'An investigation of the utility of flowcharts during computer program debugging'. *International Journal of Man-Machine Studies.* 20, 1984, 357-372.

[14] Goldenson Dennis R. & Wang Bing Jyun: 'Use of structure editing tools by novice programmers'. *Empirical Studies of Programmers: Fourth Workshop.* Ablex, 1991, 99-120.

[15] Goldstein H. H. & von Neumann J. : 'Planning and coding problems for an electronic computing instrument'. 1947, Reprinted in von Neumann, J., *Collected Works.* A.H. Traub (ed.), McMillian, 80-151.

[16] Green T.R, Petre M. & Bellamy R.K.E.: 'Comprehensibility of visual and textual programs: a test of superlativism against the match-mismatch conjecture'. *Empirical Studies of Programmers: Fourth Workshop.* Ablex, 1991, 121-146.

[17] Haibt L. M. : 'A program to draw multi-level flow charts'. *Proceedings of the Western Joint Computer Conference.* San Francisco, CA, 1959, 131-137.

[18] Hopgood F. R. A.: 'Computer animation used as a tool in teaching computer science'. *Proceedings of the 1974 IFIP Conference.* North-Holland, 889-892.

[19] Kammann J.: 'The comprehensibility of printed instructions and the flowchart alternative'. *Human Factors.* 17(2), 1975, 183-191.

[20] Knowlton K. C. : 'L6: Bell Telephone Laboratories low-level linked list languages'. Black and white sound films, Bell Laboratories, Murray Hill, 1966, NJ.

[21] Malone T. W.: *What makes things fun to learn? A study of intrinsically motivating computer games.* Ph.D Thesis, Department of Psychology, Standford University (Xerox PARC), Palo Alto, CA,1980.

[22] Mayer Richard E.: 'The psychology how novices learn computer programming'. *Computing Surveys.* 13 (1), 1981, 127-141.

[23] Myers Brad. A., 'Chandhok R. & Sareen A.: Automatic data visualization for novice Pascal programmers'. *Proceedings of the 1988 IEEE Workshop on Visual Languages.* 1988, 192-198.

[24] Myers Brad. A.: 'Visual programming, programming by example and program visualization: a taxonomy'. *Proceedings of the SIGCHI'86.* 1986, 59-66.

[25] Myers Brad. A.: 'Taxonomies of visual programming and program visualization'. *Journal of Visual Languages and Computing.* 1(1), 1990, 97-123.

[26] Price Blaine A., Baecker Ronald M. & Small Ian S.: 'A taxonomy of software visualization'. *Proceedings of the 25th International Conference on System Sciences.* Vol II, 1992, 597-606.

[27] Price Blaine A., Baecker Ronald M. & Small Ian S.: 'A principled taxonomy of software visualization'. *Journal of Visual Languages.* 4(3), 1993, 211-266.

[28] Ramsey H. Rudy, Atwood Michael E. & van Doren James R: 'Flowcharts versus program design languages: an experimental comparision'. *Communications of ACM.* 26(6), 1983, 445-449.

[29] Roman Gruia-Gatalin & Cox Kenneth C.: 'Program visualization: the art of mapping programs to pictures'. *Proceedings of the 14th International Conference on Software Engineering.* 1992, 412-419.

[30] Scanlan David A : 'Structured flowcharts outperform pseudocode: an experimental comparision'. *IEEE Software.* 6(5), 1989, 28-36.

[31] Sheppard Sylvia, Kruesi Elisabeth & Curtis Bill: 'The effect of symbology and spatial arrangement on the comprehension of software specifications'. *Proceedings of the 5th International Conference on Software Engineering.* 1981, 207-214.

[32] Shneiderman Ben: 'Control flow and data structure documentation: two experiments'. *Communications of ACM.* 25(1), 1982, 55-63.

[33] Shneiderman Ben, Mayer Richard, McKay Don & Heller Peter : 'Experimental investigations of the utility of detailed flowcharts in programming'. *Communications of ACM.* 20(5), 1977, 373-381.

[34] Stasko John T. & Patterson Charles:'Understanding and characterizing sotware visualization systems'. *Proceedings of the 1992 IEEE Workshop on Visual Languages.* 1992, 3-10.

[35] Stasko John, Badre Albert & Lewis Clayton: 'Do algorithm animations assist learning? An empirical study and analysis'. *Proceedings of INTERCHI'93.* 61-66.

[36] Stasko John T. & Vitter Jeffrey Scott: 'Pairing heaps: Experiments and analysis'. *Communications of ACM.* 30(3), 1987, 234-249.

[37] Wright P. & Reid F.: 'Written information: some alternatives to prose for expressing the outcomes of complex contingencies'. *Journal on Applied Psychology.* 57(2), 1973, 160-166.

Appendix A. Reference list for Program Visualization systems

ALADDIN

- Hyrskykari Aulikki & Räihä Kari-Jouko: 'Animation of algorithms without programming'. *Proceedings of the 1987 IEEE Workshop on Visual Languages.* 40-45.
- Helttula Esa, Hyrskykari Aulikki & Räihä Kari-Jouko: 'Graphical specification of algorithm animations with ALADDIN'. *Proceedings of the 22nd Annual Hawaii International Conference on System Science.* 1989, 892-900.
- Helttula Esa, Hyrskykari Aulikki & Räihä Kari-Jouko: 'Principles of ALADDIN and other animation systems'. *Visual Languages and Applications.* T. Ichikawa et.al. (eds.), Plenum Press, 1990, 175-187.

Amethyst

- Myers Brad. A., Chandhok R. & Sareen A.: 'Automatic data visualization for novice Pascal programmers'. *Proceedings of the 1988 IEEE Workshop on Visual Languages.* 192-198.

Anim

- Bentley Jon L. & Kernighan Brian W: 'A system for algorithm animation'. *Computing Systems.* 4(1), 1991, 5-30.
- Bentley Jon L. & Kernighan Brian W.: Anim. Available by anonymous ftp from research.att.com in /netlib/research, AT&T Bell Laboratories.

Animus

- London Ralph L. & Duisberg Robert A.: 'Animating programs in Smalltalk'. *IEEE Computer.* 18(8), 1985, 61-71.
- Duisberg Robert A.: 'Animated graphical interfaces'. *Proceedings of the SIGCHI'86.* 131-136.
- Duisberg Robert A.: 'Visual Programming of Program Visualizations'. *Proceedings of the 1987 IEEE Workshop on Visual Languages.* 55-66.

BALSA

- Brown Marc H. & Sedgewick Robert: 'A system for algorithm animation'. *Computer Graphics.* 18(7), 1984, 177-186.
- Brown Marc H.: 'Algorithm Animation'. Ph.D. Dissertation, Brown University, Department of Computer Science, Providence, RI, 1987, published also by MIT Press, 1988.
- Brown Marc H.: 'Exploring algorithms using BALSA-II'. *IEEE Computer.* 21(5), 1988, 14-36.

Field

- Reiss Steven P.: 'Interacting with the Field environment'. *Software - Practice and Experience.* 20(S1), 1990, 89-115.
- Reiss Steven P.:'Connecting tools using message passing in the FIELD environment'. *IEEE Software.* 7(4), 1990, 57-67.

LogoMotion

- Buchanan John W.: *LogoMotion: a visually enhanced programming environment.* M.Sc. Thesis, Department of Computer Science, University of Toronto, Canada, 1988.
- Backer Ronald M. & Buchanan John W.: 'A visually enhanced and animated programming environment'. *Proceedings of the 23rd Annual Hawaii International Conference on System Science.* 1990, 531-540.

LogoMedia

- DiGiano C. J.: *Visualizing program behavior using non-speech audio.* M.Sc. Thesis, Department of Computer Science, University of Toronto, Canada, 1992.
- DiGiano C. J.: LogoMedia. Available by anonymous ftp from hcrl.open.ac.uk in /pub/software/logomedia, Department of Computer Science, University of Toronto, 1992.

Movie and Stills

- Bentley J. L. & Kernighan Brian W.: 'A system for algorithm animation; Tutorial and user manual'. *AT&T Bell Laboratories Computing Science Tech.Rep.* No 132, 1987, Murray Hill, NJ.

Object-Oriented Diagramming

- Cunningham W. & Beck K.: 'A diagram for object-oriented programs'. *ACM SIGPLAN Notices.* 21(11), 1986, 361-367.

PASTIS

- Müller Heinrich, Winkler Jorg, Grzybek Stefan, Otte Matthias, Stoll Bertram, Equoy Frederic & Higelin Nicolas: 'The Program animation system PASTIS'. *The Journal of Visualization and Computer Animation.* 2(1), 1991, 26-33.

ParVis

- Linden L.B.: 'Parallel program visualization using ParVis'. in*Performance Instrumentation and Visualization.* M. Simmons & R. Koskela (eds.), ACM Press, 1990, pp. 157-188.

Pavane

- Roman Gruia-Catalin, Cox Kenneth C., Wilcox C. Donald & Plun Jerome Y.: 'Pavane: A system for declarative visualizations of concurrent programs'. *Journal of Visual Languages and Computing.* 3(2), 1992, 161-193.

PECAN

- Reiss Steven P.: 'PECAN: program development systems that support multiple views'. *IEEE Transactions on Software Engineering.* 11(3), 276-285.

PegaSys

- Moriconi Mark & Hare Dwight F.: 'PegaSys: A system for graphical explanation of program designs'. *Proceedings of the ACM SIGPLAN'85 Symposium on language issues in programming.* 148-160.
- Mark Moriconi & Dwight F. Hare: 'Visualizing program designs through PegaSys'. *IEEE Computer.* 18 (8), 1985, 72-85.

PIGS

- Pong M.C. and Ng N.: 'PIGS - A system for programming with interactive graphical support'. *Software - Practice and Experience.* 13(5),1983, 847-855.

PROVIDE

- T. G. Moher: 'PROVIDE: A Process Visualization and Debugging Environment'. *IEEE Transactions on Software Engineering.* 14(6), 1988, 849-857.

PV

- Christopher Herot Christopher F., Brown Gretchen P., Carling Richard T., Friedell Mark, Kramlich David & Baecker Ronald: 'An integrated environment for program visualization'. *Automated Tools for Information Systems Design.* Schneider & Wasserman (eds.), North-Holland, 1982, 237-259.
- Kramlich David, Brown Gretchen P., Carling Richard T. & Herot Christopher F.: 'Program visualization: graphics support for software development'. *Proceedings of the ACM/IEEE 20th Design Automation Conference* 1983, 143-149.
- Brown Gretchen. P., Carling Richard.T., Herot Christopher F., Kramlich David. A. and Souza Paul:'Program visualization: Graphical support for software development'. *IEEE Computer.* 18(8), 1985, 27-35.

PVS

- Foley J. D. & McMath C. F.: 'Dynamic program visualization'. *IEEE Computer Graphics and Applications.* 6(2), 1986, 16-25.

SEE Program Visualizer

- Baecker Ronald & Marcus Aaron: 'Design principles for the enhanced presentation of computer program source text'. *Proceedings of the SIGCHI'86.* 51-58.
- Baecker Ronald & Marcus Aaron: *Human Factors and Typography for More Readable Programs.* Addison Wesley, 1990.

Software Oscilloscope, Software through Pictures

- Wasserman Anthony I. & Pircher Perer A.: 'A graphical, extensible integrated environment for software development'. *SIGPLAN Notices.* 22(1), 1987, 131-142.

TANGO

- John T. Stasko: *Tango: a framework and system for algorithm animation.* Ph.D. Dissertation, Brown University, Department of Computer Science, Providence, RI, 1989.
- John T. Stasko: Tango: 'A framework and system for algorithm animation'. *IEEE Computer.* 23(9), 1990, 27-38.
- John T. Stasko: 'Using direct manipulation to build algorithm animations by demonstration'. *Proceedings of the SIGCHI'91.* 307-314.

TPM

- Eisenstadt Mark & Brayshaw Mike: 'The transparent Prolog machine: an execution model and graphical debugger for logic programming,. *Journal of Logic Programming.* 5(4), 1988, 277-342.
- Kwakkel Fred: 'TPM for Macintosh', version 1.1, Human Cognition Research Laboratory, UK, 1991.

UWPI

- Henry Robert R., Whaley Kenneth M. & Forstall Bruce: 'The University of Washington illustrating compiler'. *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation.* 223-233.

VIPS

- Isoda Sadahiro, Shimomura Takao & Ono Yuji: 'VIPS: A Visual Debugger'. *IEEE Software.* 4(5), 1987, 8-19.

Zeus

- Brown Marc H.: 'Zeus: A System for Algorithm Animation and Multi-View Editing'. *Proceedings of the 1991 IEEE Workshop on Visual Languages.* 4-9.