Visualizing Roles of Variables to Novice Programmers

Jorma Sajaniemi
Department of Computer Science
University of Joensuu, Finland
Jorma.Sajaniemi@joensuu.fi

Keywords: POP-I.A. learning to program, POP-II.A. novices, POP-III.A. variables, POP-III.D. visualisation, POP-V.B. case studies, POP-VI.D. computer science education research

Abstract

Many students learning to write computer programs encounter considerable difficulties. For novices, one of the key problems is in understanding how the very basic programming constructs work. In this paper, we concentrate on visualizing the role of a variable, i.e., the dynamic character of a variable embodied by the sequence of its successive values as related to other variables. We present a classification of roles and introduce an animation system, PlanAni, that uses this approach.

Introduction

Many students learning to write computer programs encounter considerable difficulties. One obvious reason is that programs deal with abstract entities – formal looping constructs, pointers going through arrays etc. – that have little in common with students' past experience. These abstract entities concern either the programming language in general or the way programming language constructs are combined to produce meaningful combinations of actions in individual programs.

Visualizations may be used to make both programming language constructs and program constructs more comprehensible (Stasko et al., 1998). A programming language is, however, just a tool to build the more important artefacts, programs, and hence visualization of higher-level program constructs is more important than visualization of language-level constructs. For example, Petre and Blackwell (1999) note that visualizations should not work in the programming language level because within-paradigm visualizations, i.e., those dealing with programming language constructs, are uninformative.

```
program doubles;
var input, count, value: integer;
begin
    repeat
        write('Enter count: '); readln(input)
    until input > 0;
    count := input;
    while count > 0 do begin
        write('Enter value: '); readln(value);
        writeln('Two times ', value, ' is ',
                2*value);
        count := count - 1
    end
end.
```

Figure 1: A short Pascal program.

As an example, consider the Pascal program in Figure 1. In this program the comparison "`some_variable > 0`" occurs twice – first to test whether the input value is valid and then to test whether there are still new values to be processed. In *programming language terms*, these two comparisons are equal: they both yield `true` if the value of the variable is greater than zero; otherwise they both yield `false`. In *program terms*, the meanings of these two comparisons are radically different: the first is a guard that looks at whatever the user has given as input and either accepts or rejects it, while the second is looking at a descending series of values and finds the moment when the series reaches its bottom. If these two tests are visualized equally, these visualizations do not provide students information about the program but about the programming language.

The dissimilarity of the tests does not stem from the programming language level but from the nature of the values involved. In the first test, the variable is an input value holder while in the second test the variable is a descending counter. In order to describe the deep meaning of the tests, we must take into account this difference in the nature of various variables. In the following, we will use the term *role* for the dynamic character of a variable embodied by the sequence of its successive values as related to other variables. For a variable, its name and type (e.g., `integer`) characterize lower-level programming language information while the role characterizes higher-level program information.

Current visualization systems pay practically no attention to the roles of variables. Semi-automatic program visualization systems (e.g., DYNALAB (Birch et al., 1995), Eliot (Lahtinen et al., 1998), Jeliot (Haajanen et al., 1997), ProgramVisualization (LaFollette et al., 2000)) provide a set of ready-made representations for variables. These representations are based on text and simple geometric forms, and operate basically on the programming language level which does little to help students to understand how variables are used to build up meaningful constructs. Hand-crafted algorithm visualization systems (e.g., BALSA-II (Brown, 1988), LogoMedia (DiGiano et al., 1993), Pavane (Roman et al., 1992), POLKA (Stasko and Kraemer, 1993), TANGO (Stasko, 1990), Zeus (Brown, 1991)) give more freedom for variable visualization but they are used to demonstrate algorithms (e.g., various sorting methods) to intermediate students. Such visualizations assume that students already know how to write programs and that they master basic constructs. Therefore, such visualizations are too difficult for novices trying to learn basic programming skills. Rather, novices need visualizations that make clear that there are different roles for variables and that these roles occur in programs again and again.

In this paper we propose a mechanism to visualize variable roles to novice programmers and show how it is implemented in a prototype system we are currently developing. The rest of this paper is organized as follows: We will start with a discussion about the roles of variables and presents a set of roles with their characteristic behaviors. We will then describe how roles can be visualized, and describe the new animation system that uses this approach. Finally, there are the conclusions.

Roles of Variables

The role of a variable characterizes the dynamic nature of the variable, e.g., a *counter* starts from zero and is continuously increased by 1. A role is not supposed to be a unique task in some specific program (e.g., the counter that counts lines in a text processing program) but a more general concept characterizing many variables in various programs.

A role is characterized by the sequence of successive values of a variable and how it depends on other variables but not by the way the variable is used. For example, the value of some variable, say `i` may be used in a single loop as an index to an array, as a limit value to guard loop termination, and as an output value for debugging purposes. Consider then another variable, say `j`, which gets the same series of values but is not used for debugging and whose final value is later used to start a new sequence of values going now backwards. These two variables will encounter the same series of values and they are considered to have the same role in spite of differences in their use.

To find an appropriate set of roles we started with roles listed by Green and Cornah (1985) in their proposal for a tool, Programmer's Torch, intended to clarify the mental processes of maintenance programmers. Among other features, the tool is supposed to reveal roles of variables listed tentatively as follows:

- Constant: a variable with an unchanged value
- Counter: a variable stepping through a series of values each depending on its own old values and possibly some other variables
- Loop counter: a counter used to control the execution of a loop

- "Most-recent" holder: a variable holding the last value encountered in going through a series of values (with no restriction on the nature of the series)
- "Best-of" holder: a variable holding the best value encountered so far (with no restriction on how to measure the goodness of a value)
- Control variable: a variable controlling the execution path based on a condition evaluated earlier in the program
- Subroutine variable: a variable used for communication between the caller and the called function; parameter, result, `var` parameter, or local variable

The last role, *subroutine variable*, is actually another attribute of variables and not a role in the sense of the previous ones. Furthermore, *counter* and *loop counter* do differ in the way the variable is used but not in the nature of the value series that the variable represents. Therefore, they will collapse into a single role in our role list below.

To make a comprehensive list of roles we started with the above role list and analyzed a set of novice-level programs. We took all programs that appear in two Pascal programming textbooks that have had continuous reprints, and in a more advanced textbook intended to be used in a second course on programming. The first book (Sajaniemi and Karjalainen, 1985) is a brief introduction to Pascal covering only a part of the language, e.g., records, pointers, files, procedures, and functions are excluded. The second book (Foley, 1991) is an introduction to programming and it covers all the above features of Pascal excluding pointers. The third book (Jones, 1982) gives a comprehensive treatment of Pascal, covers basic algorithms for searching and sorting, and describes basic data structuring methods like linked lists and hash-tables. All the books contain numerous entire programs making a total of 109 programs that were analyzed.

Table 1 contains basic data about the number of programs and variables in the analyzed books.

Book	Number of programs	Number of variables
(Sajaniemi and Karjalainen, 1985)	38	129
(Foley, 1991)	26	153
(Jones, 1982)	45	246

Table 1 – Basic data about the analyzed books.

By analyzing the behavior of each variable we created the following classification of variable roles. The role descriptions are meant for human classifiers who are able to use their understanding of a program to capture the data flow through a variable and identify main phases of this flow. For example, a *one-way flag* may be reset to its initial value at the beginning of the main loop (and changed within a nested loop) yielding the one-way behavior within the main loop even though the flag goes both ways during the entire program execution.

The identified roles of simple variables (i.e., other than arrays) are the following:

- *Constant:* a variable whose value does not change after initialization (e.g., an input value stored in a variable that is not changed later) possibly done in several alternative assignment statements (e.g., a variable that is set to `true` if the program is executed during a leap year, and `false` otherwise) and possibly corrected immediately after initialization (e.g., an input value that is replaced by zero if it is negative)
- *Stepper:* a variable going through a series of values not depending on values of other non-*constant* variables (e.g., a counter of input values, a variable that doubles its value every time it is updated, or a variable that alternates between two values) even though the selection of

possibly alternative update assignments may depend on other variables (e.g., the search index in binary search)

- *Follower:* a variable going through a series of values depending on the values of a *stepper* or another *follower* but not on other non-*constant* variables (e.g., the "previous" pointer when going through a linked list, or the "low" index in a binary search)
- *Most-recent holder:* a variable holding the latest value encountered in going through a series of values (e.g., the latest input read, or a copy of an array element last referenced using a *stepper*) and possibly corrected immediately after obtaining a new value (e.g., to scale into internal data representation format)
- *Most-wanted holder:* a variable holding the best value encountered so far in going through a series of values with no restriction on how to measure the goodness of a value (e.g., largest input seen so far, or an index to the smallest array element processed so far)
- *Gatherer:* a variable accumulating the effect of individual values in going through a series of values (e.g., a running total, or the total number of cards in hand when the player may draw several cards at a time)
- *One-way flag:* a Boolean variable that can be effectively changed only once (e.g., a variable stating whether the end of input has been encountered) even though the new value may be re-assigned several times (e.g., a variable initialized to `false` and set to `true` each time an error occurs during a long succession of operations)
- *Temporary:* a variable holding some value for a very short time only (e.g., in a swap operation)
- *Other:* all other variables

In our classification, an array is considered to be a *constant* (or *stepper*, *follower*, *most-recent holder*, *most-wanted holder*, *gatherer*, *one-way flag*) if all of its elements are *constants* (...). For example, an array is a *gatherer* if it contains 12 *gatherer* elements to calculate the total sales of each month from daily sales given as input. Moreover, there is a special role for arrays:

- *Organizer:* an array which is only used for rearranging its elements after initialization (e.g., an array used for sorting input values)

As a special case a pointer may have any of the above roles except *gatherer*, *one-way flag* or *organizer*.

For example, in the program of Figure 1 the variables `input` and `value` are *most-recent holders*, and the variable `count` is a *stepper*.

Our *constant* is a generalization of the constant role proposed by Green and Cornah (1985) because we allow a correction immediately after initialization. Their counter and loop counter are jointly covered by our *stepper* and *follower*. Their most-recent holder and best-of holder are similar to our *most-recent holder* and *most-wanted holder*, respectively. Their control variable and subroutine variable have no counterparts in our roles. Finally, our *gatherer*, *one-way flag*, *temporary*, and *organizer* cannot be found in their classification.

Ehrlich and Soloway (1984) have introduced variable plans consisting of such aspects as the variable's role in the program, the manner the variable is initialized and updated, and a guard that may protect the variable against invalid updates. As examples of roles they give "counter variable", "running total variable", and "new value variable" (that holds the newest number given as input in a loop). Our roles are more general than Ehrlich and Soloway's roles: their "counter variable" is a special case of our *stepper*, "running total variable" a special case of *gatherer*, and "new value variable" is an example of a *most-recent holder*.

Table 2 gives the distribution of roles in the three books analyzed. Variables having more than one role have been counted as many times as the number of different roles - hence the number of roles is slightly larger than the number of variables in Table 1. The three most frequent roles cover 84.0 % of all the roles, and only 1.1 % of variables were classified as *other* with none occurring in the two totally novice-level books. Consequently, we may deduce that the above set of roles is sufficient to characterize variables in novice programming.

Role	(Sajaniemi and Karjalainen, 1985) *n* = 135	(Foley, 1991) *n* = 155	(Jones, 1982) *n* = 267
Constant	51.1 %	36.8 %	25.1 %
Stepper	20.7 %	26.4 %	34.5 %
Follower	0.0 %	1.3 %	3.0 %
Most-recent holder	18.5 %	19.4 %	22.1 %
Most-wanted holder	2.2 %	1.9 %	0.7 %
Gatherer	4.4 %	5.2 %	8.2 %
One-way flag	1.5 %	0.0 %	1.5 %
Temporary	0.8 %	4.5 %	0.7 %
Organizer	0.8 %	4.5 %	1.9 %
Other	0.0 %	0.0 %	2.3 %
Total	100.0 %	100.0 %	100.0 %

Table 2 – Roles of variables in the analyzed books.

Visualization of Roles

The visualization of a role must be both general in the sense that the same form will be used for all variables of that role, and specific in the sense that it should make clear how the successive values relate to each other, and to other variables. For example, a *constant* should give the impression of a value that is not easy to change. Table 3 lists properties of the roles that should be reflected by their visualizations.

Graphic visualizations are to some extent related to cultural environment. For example, a variable that is always increased by 1 could be visualized by a concrete tally counter, a small hand-held device used for counting purposes, but such a device is not widely known in all countries. For example in Finland, practically nobody knows the existence of such a device. As a consequence, role visualizations must be tailored to the cultural background of users, and the following sketches may not appear natural to all readers

Role	Inherent properties
Constant	Impossible to change
Stepper	Future values can be predicted if past values are known; usually there is a direction for successive values: either upwards or downwards
Follower	Tightly connected to another variable; usually its previous value
Most-recent holder	Successive values are obtained from some data series but they have no fixed relationship
Most-wanted holder	Current value is better than any of the previous values
Gatherer	A new value is obtained by combining some new data and the previous value
One-way flag	Only two possible values; impossible to regain the initial value once changed
Temporary	Exists for short time periods only
Organizer	Individual parts cannot be changed but they can be moved around
Other	No fixed properties

Table 3 – Role properties that should be visualized.

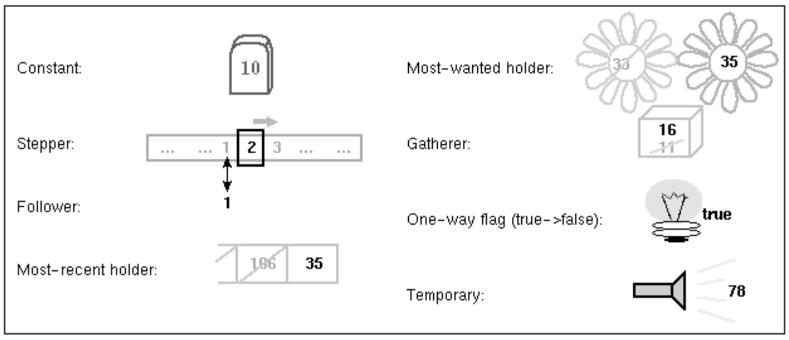


Figure 2: Example visualizations for various roles.

Figure 2 gives examples of role visualizations designed to reveal the properties listed in Table 3. In our implementation, we use changeable role images so that the actual outlook of role visualizations can be tailored to suit user backgrounds. In Figure 2, a *constant* is shown as a stone that gives the impression of being not so easy to change. A *stepper* shows its current value and two lists of some values it has had or may have in the future, together with an arrow giving the direction that the stepper is currently going. A *follower* is attached to the variable it follows. Both a *most-recent holder* and *most-wanted holder* show the current and previous values, a *most-recent holder* depicted by squares in a neutral color and a *most-wanted holder* by flowers of different colors: green for the current value, i.e. the best found so far, and yellow for the previous, i.e., the next best.

Both a *constant* and a *most-recent holder* may be adjusted immediately a value has been assigned, for example to convert input into upper case letters. In such a case, the original value can be shown with a small font in the bottom-left corner of the stone or square holding the adjusted value because the original value is not a previous value in the "normal" sense.

A *gatherer* is depicted as a box holding the current and the previous value with new values coming into it from above through its lid. A *one-way flag* is a light bulb which will break when the flag goes off. Finally, a *temporary* is shown as a flashlight that is on just as long as the value is used. When the value has no meaning any more, the flashlight goes off and the value itself turns into grey.

The role has an effect on the way the variable is used, and this can be taken care of by employing *role-sensitive visualizations of operations*, i.e., the visualization of an operation depends on the roles of the variables participating in that operation. For example, consider the two syntactically similar comparisons "`some_variable > 0`" of the program in Figure 1. Figure 3 gives two visualizations revealing the difference in the deep meanings of these comparisons. In case (a), the variable is a *most-recent holder* and the comparison just checks whether the value is in the allowed range. In the visualization, the set of possible values emerges, with allowed values colored in green and disallowed values in red. The arrow that points to that part of the values where the variable lays, appears as green or red depending on the values it points to. The arrow flashes to indicate the result of the comparison.

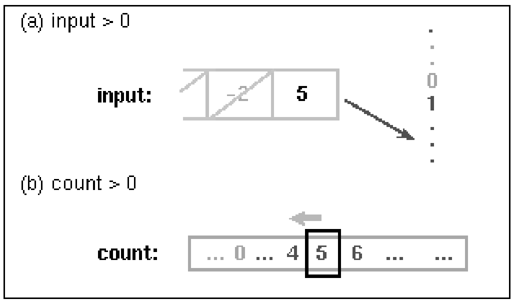


Figure 3: Visualizations of the same operation for different roles.

In Figure 3 (b) the variable is a *stepper* and, again, the allowed and disallowed values are colored. However, these ranges are now part of the variable visualization and no new values do appear. The current value of the variable flashes and the user can see the result by its color. In both visualizations, if the border value used in the comparison is an expression (as opposed to a single constant), the expression is shown next to the value.

The selection of the appropriate method for comparison visualization can be selected automatically based on the roles of the participating variables. If two variables are involved in a comparison, say "`a > b`", the visualization could be built around either variable, but the roles can be used to determine which one should be used. For example, if one variable is a *stepper* and the other is a *most-recent holder*, then it is reasonable to assume that the *stepper* is the central variable that is compared to a *most-recent holder* border.

As another example, Figure 4 depicts variables in a program that finds the average and maximum of its inputs. In spite of the nonsense names given to the variables the meaning of each variable can be immediately recognized by their roles. Even the method used to detect the end of input is easy to deduce from the visualization of the comparison. In real use, the variable names should of course be meaningful, and program code should be available to users. This example just shows that the visualization of variable roles contains more information than provided by a visualization based on mere values of variables.

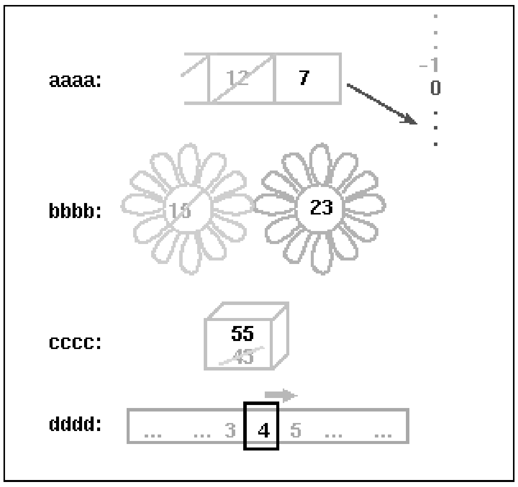


Figure 4: Variables for finding the average and maximum of inputs.

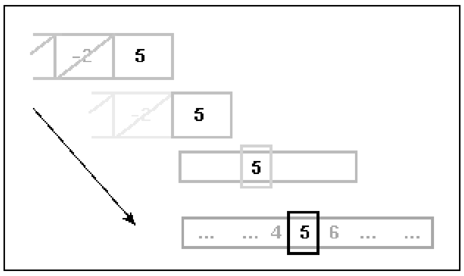
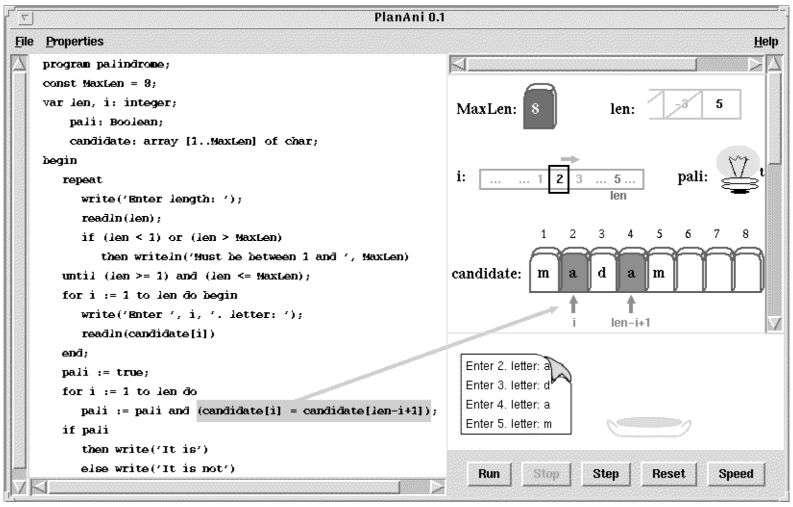


Figure 5: Animation of a most-recent holder *changing its role to a* stepper.

The role of a variable may change during the execution of a program and this happens usually somewhere between two loops. For example, in the program of Figure 1, the two variables `input` and `count` could be combined to a single variable, say `count` (making the assignment "`count := input;`" unnecessary). The role of this variable would first be a *most-recent holder* and then, in the second loop, a *stepper*. This can be animated by a smooth deformation of the visualization as shown in Figure 5.

Role-Based Program Animation

We are currently building a prototype animation system, PlanAni, that animates semi-automatically small programs using the role-based approach. The first version of the prototype works with a fixed set of pre-defined programs but we have plans to extend it to work with any program. Figure 6 depicts the user interface when the system is animating a simple program that checks whether its input is a palindrome. The left pane shows the animated program with a color enhancement showing the current action. The upper part of the right pane is reserved for the variables, and below it there is the input/output area consisting of a paper for output and a plate where the user can enter input. The currently active action is connected with an arrow to the corresponding variables on the right.



```
program palindrome;
const MaxLen = 8;
var len, i: integer;
    pali: Boolean;
    candidate: array [1..MaxLen] of char;
begin
  repeat
    write('Enter length: ');
    readln(len);
    if (len < 1) or (len > MaxLen)
      then writeln('Must be between 1 and ', MaxLen)
  until (len >= 1) and (len <= MaxLen);
  for i := 1 to len do begin
    write('Enter ', i, '. letter: ');
    readln(candidate[i])
  end;
  pali := true;
  for i := 1 to len do
    pali := pali and (candidate[i] = candidate[len-i+1]);
  if pali
    then write('It is')
    else write('It is not')
```

```
Enter 2. letter: a
Enter 3. letter: d
Enter 4. letter: a
Enter 5. letter: m
```

Figure 6: Visualization of a comparison in the PlanAni system.

In the program, the array `candidate` is a collection of *constants* because the elements of the array do not change after they have been initialized. To avoid unnecessary details PlanAni does not animate the evaluation of expressions: the resulting value – accompanied by the expression itself – is shown and its effect in a comparison or assignment is animated only. In Figure 6, the program is comparing two elements of the array. The elements appear as colored and their indexes are denoted by small arrows marked with the appropriate expression. As the elements are equal, they are colored green.

It is practically impossible for humans to divide their visual attention to two or more targets especially if the task is demanding as compared to the abilities of the person (Lavie, 1995). In program animation, users are novices in program constructs and it is therefore hard for them to follow what happens in various parts of the user interface. In order to minimize users' need to jump back and forth between the program code pane and the variable pane, we use a speech synthesizer that explains what happens in the program. For example, in the situation of Figure 6 the voice says "*comparing whether the element* i *of the array* candidate *is the same as the element* len-i+1." Voice is also used in connection to variable creation (e.g., "*creating a gatherer called sum*"), role changes ("*the variable count acts henceforth as a stepper*"), and control constructs ("*entering a loop*"). The use of the speech synthesizer divides perceptual load between the viewer's visuo-spatial and auditory systems and hence gives her more capacity as opposed to pure visual stimulus used in many animation systems.

Conclusions

We have shown that there is a small set of roles of variables covering the vast majority of variables in novice-level programs. Roles capture higher-level program information of variables that can be utilized in program visualization for novices. We have presented example visualizations of roles and role-specific operations, and introduced a prototype program animation system that uses this approach.

Empirical studies have shown that visualization is problematic for very early novices (Mulholland and Eisenstadt, 1998) implying that learners at different skill levels need different visualizations. Roles represent very basic programming skills and, in accordance with this, variable role visualization is intended for novice use. More experienced programmers may find these visualizations too space-demanding.

Current program visualization systems work on the programming language level and provide no specific program level information to students. Our approach captures one part of this higher-level information and presents it to viewers. However, we do not expect that the visualizations used for roles could be immediately understandable to all novices. Therefore, animation sessions are supposed to be preceded by an explanation of the various roles, how they are visualized, and why the visualizations look the way they look. Moreover, first animations shown should utilize a few roles only. Other features of the animation should be more obvious to PlanAni users and no special guidance should be needed. We are planning an experiment to empirically validate these effects once the first version of the prototype system is ready.

Acknowledgments

The author would like to thank Raija Sarajärvi for acting as the second judge in classifying variables in the analyzed programs and for programming a demonstration version of the PlanAni system.

References

Birch, M.R., Boroni, C.M., Goosey, F.W., Patton, S.D., Poole, D.K., Pratt, C.M. and Ross, R.J. (1995) DYNALAB A Dynamic Computer Science Laboratory Infrastructure Featuring Program Animation. *ACM SIGCSE Bulletin,* **27**(1): 29-33.

Brown, M.H. (1988) Exploring Algorithms Using Balsa-II. *IEEE Computer,* **21**(5): 14-36.

Brown, M.H. (1991) ZEUS: A System for Algorithm Animation and Multi-View Editing. In *Proceedings of the 1991 IEEE Workshop on Visual Languages,* Kobe, Japan, 4-9.

DiGiano, C.J., Baecker, R.M. and Owen, R.N. (1993) LogoMedia: A Sound-Enhanced Programming Environment for Monitoring Program Behaviour. In *INTERCHI'93,* 301-302.

Ehrlich, K. and Soloway, E. (1984) An Empirical Investigation of the Tacit Plan Knowledge in Programming. In *Human Factors in Computer Systems*, (Thomas, J. C. and Schneider, V. L. (eds.)), Norwood: Ablex Publishing Co., 113-133.

Foley, R.W. (1991) *Introduction to Programming Principles Using Turbo Pascal.* London: Chapman&Hall.

Green, T.R.G. and Cornah, A.J. (1985) The Programmer's Torch. In *Human-Computer Interaction - INTERACT'84,* 397-402.

Haajanen, J., Pesonius, M., Sutinen, E., Tarhio, J., Teräsvirta, T. and Vanninen, P. (1997) Animation of User Algorithms on the Web. In *VL'97, IEEE Symposium on Visual Languages,* 360-367.

Jones, W.B. (1982) *Programming Concepts – A Second Course.* Englewood Cliffs: Prentice-Hall.

LaFollette, P., Korsh, J. and Sangwan R. (2000) A Visual Interface for Effortless Animation of C/C++ Programs. *Journal of Visual Languages and Computing,* **11**: 27-48.

Lahtinen, S.-P, Sutinen, E. and Tarhio, J. (1998) Automated Animation of Algorithms with Eliot. *Journal of Visual Languages and Computing,* **9**: 337-349.

Lavie, N. (1995) Perceptual Load as a Necessary Condition for Selective Attention. *Journal of Experimental Psychology: Human Perception and Performance,* **21**: 451-468.

Mulholland, P. and Eisenstadt, M. (1998) Using Software to Teach Programming: Past, Present and Future. In (Stasko et al., 1998), 399-408.

Petre, M. and Blackwell, A.F. (1999) Mental Imagery in Program Design and Visual Programming. *International Journal of Human-Computer Studies,* **51**(1): 7-30.

Roman, G.-C., Cox, K., Wilcox, C. and Plun, J. (1992) Pavane: A System for Declarative Visualization of Concurrent Computations. *Journal of Visual Languages and Computing,* **3**(1): 161-193.

Sajaniemi, J. and Karjalainen, M. (1985) *Suppea johdatus Pascal-ohjelmointiin (A Brief Introduction to Programming in Pascal).* Joensuu: Epsilon ry.

Stasko, J.T. (1990) Tango: A Framework and System for Algorithm Animation. *IEEE Computer,* **23**(9): 27-39.

Stasko, J., Domingue, J., Brown, M.H. and Price, B.A. (eds.) (1998) *Software Visualization – Programming as a Multimedia Experience.* Cambridge: The MIT Press.

Stasko, J. and Kraemer, E. (1993) A Methodology for Building Application-Specific Visualizations of Parallel Programs. *Journal of Parallel and Distributed Computing,* **18**(2): 258-264.