Abstract

Computer programming is a difficult skill for many students and visualizations may be used to foster learning. This paper presents a program animation system, PlanAni, that is based on the concept of the roles of variables. Roles represent schematic uses of variables that occur in programs over and over again, and a set of nine roles covers practically all variables in novice-level programs. PlanAni has been tested in a teaching experiment comparing traditional teaching with role-based teaching and animation. The results of a semi-structured interview with the teacher indicate that students like to work with the animator and that the system clarifies many concepts in programming.

CR Categories: K.3.2 [Computers and Education]: Computer and Information Science Education—Computer science education; D.m [Software]: Miscellaneous—Software psychology

Keywords: program animation, roles of variables, teaching experiment

1 Introduction

Computer programming is a difficult skill for many students to learn. One reason is that programs deal with abstract entities—formal looping constructs, pointers going through arrays etc.—that have little in common with everyday issues. These entities concern both the programming language in general and the way programming language constructs are combined to produce meaningful combinations of actions in individual programs. New methods and techniques that help students to understand these issues are needed to enhance learning elementary programming.

Visualizations may be used to make both programming language constructs and program constructs more comprehensible [Hundhausen et al. 2002; Mulholland 1998]. From the programming point of view, the programming language is just a tool used to build the more important artefacts, programs. 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. Hence visualization of higher-level program constructs should be preferred to visualization of language-level constructs in teaching to 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 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 succession of values and finds the moment when the succession 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—which is uninformative.

The dissimilarity of the comparisons cannot be observed by looking at the comparisons themselves or by looking at their locations within the syntactic constructs of the program. The distinctive issue is the nature of the variables involved; to detect this difference a deeper analysis of the program is needed. In the first comparison, the variable is an input value holder while in the second the variable is a descending counter. In order to describe the true meaning of the comparisons, we must take into account this difference. Sajaniemi [2002] has recently introduced the term role to describe this aspect of variables that characterizes their behavior.

A role is defined to be the dynamic character of a variable embodied by the sequence of its successive values as related to other

Figure 1: A short Pascal program.
variables and external events. For example, the variable data obtains its new values from an unpredictable external source while the new values of the variable count can be predicted as soon as the variable is initialized. Thus roles characterize higher-level program information of variables. Sajaniemi [2002] has found that only nine roles are needed to cover 99% of variables in novice-level programs.

Teaching elementary programming may be augmented by introducing roles to students. Traditionally, students have had to acquire this kind of higher-level knowledge from example programs and program fragments. Introducing the nine roles explicitly gives students a comprehensive set of concepts and a vocabulary that they can use in studying example programs and in authoring new programs. Roles provide also a basis for program animation that, e.g., reveals the difference between the two comparisons in the program of Figure 1. Thus role-based animation can escape the within-paradigm level and has the potential of being informative for students.

Current visualization systems pay practically no attention to the roles of variables. Semi-automatic visualization systems (e.g., DISCOVER [Ramadhan 2000], DYNALAB [Birch et al. 1995], Eliot [Lahtinen et al. 1998], Jeliot [Haajanen et al. 1997]) provide a set of ready-made representations for variables from which users can choose whatever representation they like. On the other hand, hand-crafted visualization systems (e.g., ANIMAL [Rösslung and Freisleben 2002], BALSA-II [Brown 1988], HalVis [Hansen et al. 2002], 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.

Semi-automatic visualization systems have a fixed set of optional visualizations for variables, usually with a larger set of alternatives for scalar variables than for arrays. Visualizations may be purely textual or involve a simple graphical form, like a rectangle, with the name and value of the variable in textual form. Operations are visualized automatically using smooth transition of values from one variable to another or to the side of a comparison operator. In order to capture users’ attention the operands may change color or size just before the movement starts. Figure 2 [Ben-Bassat Levy et al. 2003] showing the user interface of Jeliot 2000 is a representative example of this approach.

In this paper we present a new program animation system, PlanAni, that uses variable roles as a basis for visualization. PlanAni provides automatic program animation and it is intended to be used in teaching basic programming constructs to novices. We will also report experiences of using the system in teaching an introductory programming course.

The rest of this paper is organized as follows: Section 2 is an overview of variable visualization in current animation systems. Section 3 discusses the role concept for variables and presents roles with their characteristic behaviors. Section 4 describes how roles are utilized to produce higher-level visualizations in the PlanAni system and other features of PlanAni. Section 5 analyses our experiences in using role-based animation in teaching, and Section 6 contains the conclusion.

2 Traditional Visualization of Variables

Current program visualization systems can be divided into two categories depending on their approach to visualization of variables. Semi-automatic visualization systems (e.g., DISCOVER [Ramadhan 2000], DYNALAB [Birch et al. 1995], Eliot [Lahtinen et al. 1998], Jeliot [Haajanen et al. 1997]) provide a set of ready-made representations for variables from which users can choose whatever representation they like. On the other hand, hand-crafted visualization systems (e.g., ANIMAL [Rösslung and Freisleben 2002], BALSA-II [Brown 1988], HalVis [Hansen et al. 2002], 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.

Semi-automatic visualization systems have a fixed set of optional visualizations for variables, usually with a larger set of alternatives for scalar variables than for arrays. Visualizations may be purely textual or involve a simple graphical form, like a rectangle, with the name and value of the variable in textual form. Operations are visualized automatically using smooth transition of values from one variable to another or to the side of a comparison operator. In order to capture users’ attention the operands may change color or size just before the movement starts. Figure 2 [Ben-Bassat Levy et al. 2003] showing the user interface of Jeliot 2000 is a representative example of this approach.

All these visualization techniques operate solely on the programming language level. Some aspects of variables can be stressed by a carefully considered selection of graphical forms, but it is questionable how much information about the nature of various variables is brought to the novice user by, e.g., the fact that the corners of the rectangle containing the value are rounded.
3 Roles of Variables

This section introduces the role concept, gives a comprehensive list of roles in novice-level procedural programming, and explains role changes.

3.1 The Role Concept

The role of a variable [Sajaniemi 2002] characterizes the dynamic nature of the variable embodied by the sequence of its successive values as related to other variables and external events. A role is not a unique task in some specific program but a more general concept occurring in programs over and over again. The way the value of a variable is used has no effect on the role, e.g., a variable whose value does not change is considered to be a fixed value whether it is used to limit the number of rounds in a loop or as a divisor in a single assignment.

The role concept has been studied by Sajaniemi [2002] who has found that only nine roles are needed to cover 99 % of variables in novice-level, procedural programs. The following list gives for each role both an exact definition and an informal definition suitable to be used in teaching:

- **Fixed value**: 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 its absolute value if it is negative).

  Informal definition: A variable whose value does not change after initialization.

- **Stepper**: A variable going through a succession of values depending on its own previous value and possibly on other stepper, stepper followers, and fixed values (e.g., a counter of input values, a variable that doubles its value every time it is updated, a variable that alternates between two values, or an index to an array that sweeps through the array using varying densities) even though the selection of possibly alternative update assignments may depend on other variables (e.g., the search index in binary search).

  Informal definition: A variable stepping through a systematic, predictable succession of values.

- **Follower**: A variable which, apart from initialization, goes through a succession of values depending on the value of a single variable that is updated immediately after being used for updating the follower, and possibly on fixed values (e.g., the “previous” pointer when going through a linked list, or the “low” index in a binary search).

  Informal definition: A variable that gets its new value always from the old value of some other variable.

- **Most-recent holder**: A variable holding the latest value encountered in going through a succession 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).

  Informal definition: A variable holding the latest value encountered in going through a succession of values, or simply the latest value obtained as input.

- **Most-wanted holder**: A variable holding the best value encountered so far in going through a succession 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).

  Informal definition: A variable holding the best or otherwise most appropriate value encountered so far.

- **Gatherer**: A variable accumulating the effect of individual values in going through a succession 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).

  Informal definition: A variable accumulating the effect of individual values.
• **One-way flag:** A two-valued 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).

  Informal definition: A two-valued variable that cannot get its initial value once its value has been changed.

• **Temporary:** A variable holding the value of some other variable or input value for a very short time only (e.g., in a swap operation).

  Informal definition: A variable holding some value for a very short time only.

In this classification, an array is considered to be a **fixed value** (or stepper, follower, most-recent holder, most-wanted holder, gatherer, one-way flag) if all of its elements are fixed values (…). For example, an array is a **gatherer** if it contains 12 gatherers 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).

  Informal definition: An array used for rearranging its elements.

Sajaniemi [2002] has analyzed all programs in two Pascal programming textbooks that have had continuous reprints [Sajaniemi and Karjalainen 1985; Foley 1991], and in a more advanced textbook intended to be used in a second course on programming [Jones 1982]. His analysis covers all variables in all 109 entire programs in these books. Table 1 gives the distribution of roles in the three books.

Ben-Ari and Sajaniemi [Submitted] have shown that in one hour’s work, computer science teachers can learn roles and assign them successfully in normal cases. In their experiment, the roles one-way flag, temporary and organizer were not included in the materials—nevertheless, one variable to be recognized was a one-way flag. Subjects’ responses to this variable were totally different from other responses, with a large variety in suggested roles. These results provide evidence that roles are intuitive for computer science educators and that all the roles are really needed.

It should be noted that roles are cognitive—rather than technical—concepts. As an example, consider the Fibonacci sequence 1, 1, 2, 3, 5, 8, 13, … where each number is the sum of the previous two numbers. A mathematician who knows the sequence well can probably see the sequence as clearly as anybody sees the sequence 1, 2, 3, 4, 5, … i.e., the continuum of natural numbers. On the other hand, for a novice who has never heard of the Fibonacci sequence before and who has just learned the way to compute it, each new number in this sequence is a surprise. Hence, the mathematician may consider the variable as stepping through a known succession of values (i.e., a stepper) while the novice considers it as a gatherer accumulating previous values to obtain the next one.

Another example of the cognitive nature of the roles is the character of role descriptions above. They 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 the 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. It might be hard to try to define such a property formally but for human cognition this is not a problem.

In teaching elementary programming, one can introduce roles to students by presenting the above informal definitions and accompanying them with simple example programs. For a more advanced reader the rest of this subsection covers some special cases that can (and should) be skipped in novice teaching.

In deciding the role of a variable, pointers are treated just like other variables. In particular, the role of a pointer describes how the pointer behaves and not how the pointed variable behaves. Dynamic variables can in principle be treated similarly to normal, static variables.

Fields of record variables must be treated individually because they may have different roles. For example, a record may consist of two fields: one for identifying the item that the record describes (presumably a fixed value) and one for accumulating data concerning that item (presumably a gatherer).

In procedures and functions, formal value parameters and local variables can be considered to be normal variables. In contrast, formal variable parameters must be unified with the actual parameter because every change made to the parameter (within the procedure or function) contributes to the succession of values that the actual variable goes through.

Finally, file variables behave totally differently from other variables: the file variable itself is usually a fixed value, i.e., it denotes the same file throughout the program, but the contents of successive file elements convey a file element role not comparable to other roles.

### Table 1: Roles of variables in three textbooks

<table>
<thead>
<tr>
<th>Role</th>
<th>[Sajaniemi and Karjalainen 1985]</th>
<th>[Foley 1991]</th>
<th>[Jones 1982]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed value</td>
<td>51.1 %</td>
<td>36.8 %</td>
<td>25.1 %</td>
</tr>
<tr>
<td>Stepper</td>
<td>20.7 %</td>
<td>26.4 %</td>
<td>34.5 %</td>
</tr>
<tr>
<td>Follower</td>
<td>0.0 %</td>
<td>1.3 %</td>
<td>3.0 %</td>
</tr>
<tr>
<td>Most-recent holder</td>
<td>18.5 %</td>
<td>19.4 %</td>
<td>22.1 %</td>
</tr>
<tr>
<td>Most-wanted holder</td>
<td>2.2 %</td>
<td>1.9 %</td>
<td>0.7 %</td>
</tr>
<tr>
<td>Gatherer</td>
<td>4.4 %</td>
<td>5.2 %</td>
<td>8.2 %</td>
</tr>
<tr>
<td>One-way flag</td>
<td>1.5 %</td>
<td>0.0 %</td>
<td>1.5 %</td>
</tr>
<tr>
<td>Temporary</td>
<td>0.8 %</td>
<td>4.5 %</td>
<td>0.7 %</td>
</tr>
<tr>
<td>Organizer</td>
<td>0.8 %</td>
<td>4.5 %</td>
<td>1.9 %</td>
</tr>
<tr>
<td>Other</td>
<td>0.0 %</td>
<td>0.0 %</td>
<td>2.3 %</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>100.0 %</strong></td>
<td><strong>100.0 %</strong></td>
<td><strong>100.0 %</strong></td>
</tr>
</tbody>
</table>
Role changes are of two basic types: either the final value of the variable in the first role is used as the initial value for the next role, or the variable is re-initialized with a totally new value at the beginning of the new role phase. In the former case the role change is called proper and in the latter case it is said to be sporadic. This distinction is not only technical but it is also meaningful for a human trying to comprehend a program. In case of a proper role change it is important to know which value ends the succession of values of the first role and how it affects the new series of values. In contrast, sporadic role changes have little to do with the meaning of the variable—rather, they tell about a will to keep the number of variables low for, e.g., memory conservation reasons.

In his analysis of novice-level procedural programs, Sajaniemi [2002] found that only 5% of the variables were affected by role changes and that 85% of role changes were sporadic. However, in most sporadic role changes the variable was used for a similar purpose in both roles, e.g., as a row index for a two-dimensional array. Figure 4 depicts role change types in these programs. The most frequent change types were sporadic role changes in both ways between most-recent holder and stepper.

The PlanAni Program Animator

We will now turn to PlanAni—a program animation system intended to be used in teaching basic programming constructs to novices. We start by describing how PlanAni utilizes roles in program visualization and animation, and continue with other features and an overview of its implementation.

Role-Based Visualization

In PlanAni, each role has a visualization—role image—that is used for all variables of the role. Role images give clues on how the successive values of the variable relate to each other and to other variables. For example, a fixed value is depicted by a stone giving the impression of a value that is not easy to change, and a most-wanted holder by flowers of different colors: a bright one for the current value, i.e., the best one found so far, and a gray one for the previous, i.e., the best next value. A most-recent holder shows its current and previous values, also, but this time they are known to be unrelated and this fact is depicted by using two squares of a neutral color.

A stepper is depicted by footprints and shows the current value and some of the values the variable has had or may have in the future, together with an arrow giving the current direction of stepping. A follower is visualized as a dog and it is positioned next to the variable whose values it follows. A gatherer is depicted as a box holding the current and the previous value. A one-way flag is a light bulb which will break when the flag goes off, or a toaster that pops up the toasted bread when the flag goes on. 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 disappears.

Finally, arrays are depicted using copies of the associated role image, one copy for each array element. The elements of an organizer are similar to the fixed value stone but now the stones have small wheels to give the impression of easy movability.

Graphic visualizations are to some extent related to cultural environment [Gershon et al. 1998]. In PlanAni, role visualizations are implemented using GIF images and separate files defining the coordinates of the various values within the image. Thus role images can be changed without editing the source code. Table 2 lists basic properties of roles that should be considered when devising alternative role images.

Role-Based Animation

In addition to role images, PlanAni utilizes role information for role-based animation of operations, also. As the deep meaning of operations is different for different roles, PlanAni uses different animations. For example, Figure 5 (also on color plate) gives visualizations for the two syntactically similar comparisons "some_variable > 0" of the program in Figure 1. 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, allowed values with a green background and disallowed values with red. The arrow that points to that part of the values where the current value of 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.

In Figure 5(b) the variable is a stepper and, again, the allowed and disallowed values are colored. However, these values are now part of the variable visualization and no new values do appear. The values flash and the user can see the result by the color of the current value. 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.

Just as comparisons are animated differently for different roles, the animation of assignment statements depends on roles. Figure 6 (also on color plate) shows the animation of the assignment fib := fib+temp where fib is a gatherer. The old value of the variable moves to the lower left corner of the role image and becomes gray, the expression and a short arrow appear, the result of the expression appears inside the box, and finally the expression and the arrow disappear. Thus the animation stresses the typical behavior of gatherers: the new value is obtained as a combination of the old value and some other data.

Assignment to a stepper looks radically different: the numbers inside the role image scroll smoothly until the new value is between the middle footprints. Thus the animation stresses the fact...
Table 2: Properties of roles for visualization

<table>
<thead>
<tr>
<th>Role</th>
<th>Inherent properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed value</td>
<td>Impossible to change after initialization; may have two distinct values during its lifetime</td>
</tr>
<tr>
<td>Stepper</td>
<td>Most often the step between successive values is constant; future values can be predicted if past values are known; usually there is a direction for successive values: either upwards or downwards; the validity of a stepper’s value is usually repeatedly tested against the same limit</td>
</tr>
<tr>
<td>Follower</td>
<td>Tightly connected to another variable; usually its previous value</td>
</tr>
<tr>
<td>Most-recent holder</td>
<td>Successive values are obtained from some data succession but they have no fixed relationship; each logical value may consist of two distinct values: the original and the scaled or corrected</td>
</tr>
<tr>
<td>Most-wanted holder</td>
<td>The current value is better than any previous value; successive values are obtained from some data succession but new values are discarded if they are worse than any of those seen before</td>
</tr>
<tr>
<td>Gatherer</td>
<td>The new value is obtained by combining some new data with the previous value</td>
</tr>
<tr>
<td>One-way flag</td>
<td>Only two possible values; impossible to regain the initial value once changed</td>
</tr>
<tr>
<td>Temporary</td>
<td>Exists for short time periods only; tightly connected to the previous value of another variable</td>
</tr>
<tr>
<td>Organizer</td>
<td>Individual elements cannot be changed but they can be moved around</td>
</tr>
<tr>
<td>Other</td>
<td>No fixed properties</td>
</tr>
</tbody>
</table>

that the new values of steppers are typically known as soon as the succession of values starts.

Role change animations depend on the type of the change. A proper role change is animated as a smooth transformation of the role image. e.g., in a proper role change from a most-recent holder to a stepper the two boxes start to get wider and then smoothly transform to footprints. Sporadic role changes are animated as a fade-out of the old role image followed by a fade-in of the new role image. Thus role change visualizations show the strength of the relation between the old and new roles.

Smooth transformations are also used for changes in the values of one-way flags. For example, if a one-way flag that is originally true, depicted by a glowing light bulb, is set to false, depicted by a broken one, the light bulb bursts into small pieces in a smooth animation. The animation makes it clear that there is no return back to the original value.

4.3 Other Features

Figure 7 (also on color plate) is an actual screenshot of the PlanAni 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 variables, and below it there is the input/output area consisting of a paper for output and a plate for input. The currently active action in the program pane on the left is connected with an arrow to the corresponding variables on the right. Whenever the color enhancement is moved to a new location in the program, the new enhancement flashes to attract users’ attention.

It is impossible for humans to divide their visual attention into two or more targets—especially if the task is demanding for the person in question [Lavie 1995]. As a consequence, it is hard for novices to follow simultaneous actions in various parts of the user interface. In order to minimize users’ needs to jump back and forth between the program code and the variables, the system uses frequent pop-ups that explain what is going on in the program. This includes variable creation (e.g., “creating a gatherer called sum”), role changes (“the variable count acts henceforth as a stepper”), operations (“comparing count with zero”), and control constructs.
Figure 7: Visualization of an array element comparison in the PlanAni system.

<table>
<thead>
<tr>
<th>Automatic role analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program−level animation</td>
</tr>
<tr>
<td>Statement−level animation</td>
</tr>
<tr>
<td>Graphic primitives and animation</td>
</tr>
</tbody>
</table>

Figure 8: Architectural levels of PlanAni.

(“entering a loop”). Our intention is to use a speech synthesizer in place of pop-ups in future versions of PlanAni.

To avoid unnecessary details PlanAni does not animate the evaluation of expressions: only the resulting value—accompanied by the expression itself—is shown and its effect in a comparison or assignment is animated.

Users can change animation speed and the font used in the panes. Animation can proceed continuously (with pauses because the frequent pop-ups require clicking “Ok” button, or pressing “Enter”) or stepwise. Animation can be restarted at any time but backward animation is not possible.

4.4 Implementation

PlanAni is implemented using Tcl/Tk and it has been tested both on Linux/Unix and Windows NT. The architecture consists of four levels as depicted in Figure 8. The lowest level takes care of primitive graphics and animation, and implements the user interface. The next level knows how to animate smallest actions that are meaningful to viewers of the animation. This level is language independent in the sense that it can be used to animate programs written in various languages, e.g., Pascal, C, and Java.

The next level takes as input a program to be animated, annotated with the roles of variables and possible role changes, and animates the program automatically. Finally, the uppermost level does not need role information because it finds roles automatically.

Currently, the two uppermost levels are not implemented. As a consequence, animation commands must be authored by hand for each program to be animated. Typically 5 animation lines are required for each line in the animated program. Figure 9 gives Tcl/Tk commands needed to animate the variable declarations and the first loop of the program in Figure 1. The first fourteen \texttt{AN\_create\_program\_line} commands (of which only the first and last are given in Figure 9) bring up on the left pane of the user interface the program code to be animated. The \texttt{AN\_start\_animation} command then waits for the user to start running the animation.

The three \texttt{AN\_create\_var} commands create role images for the three variables in the animated program. These commands get as parameters the name of the variable, and its role and initial value. The last parameter (in curly braces) gives the program line and character indexes of the variable declaration in the program code so that the animator can enhance this part of the program when animating the creation of the variable. Optional parameters can be used to declare the type of the variable (if not \texttt{integer}), to give the number of elements (if the variable is an array), and to guide the animator to position the role image with respect to other role images on the right pane of the user interface. The \texttt{AN\_create\_var} command returns a handle to the new role image. For example, the handle of the variable \texttt{data} of the animated Pascal program is assigned to the Tcl/Tk variable \texttt{d\_H}. 
In the current version of PlanAni, an \texttt{AN\_notice} command brings up a pop-up explaining actions of the animated program. In future, this command will use a speech synthesizer for the same task. Its parameters are the explanation to be given and a reference to the part of the animated program that is to be enhanced during the explanation.

Next, the animation of the first loop begins with a request for input (\texttt{AN\_write\_line}). The \texttt{AN\_var\_read\_value} command reads a value and animates its transfer to the appropriate role image, i.e., to the Pascal variable data denoted by the handle \texttt{d\_H}. PlanAni keeps track of values of animated variables, and therefore there is no need for the \texttt{AN\_var\_read\_value} command to return the actual user input. If needed, the values of animated variables can be obtained with an \texttt{AN\_var\_get\_attribute} command. In the example, the value of the animated variable data is utilized in the comparison ("data > 0") that tests whether the loop ends. This comparison is animated using the \texttt{AN\_compare} command that obtains a handle of the variable as parameter, and finds the value of the variable through the handle. This command animates the comparison (cf. Figure 5(a)) and returns the result of the comparison. Thus, there is no need for the animating program in Figure 9 to know the values given by a user as input for the Pascal variable data.

The Tcl/Tk variable \texttt{ok} is used to store the result of the comparison, and the animation loop continues until the \texttt{AN\_compare} command returns \texttt{I} (true) as its result. Then, a valid input has been entered by the user and the animation proceeds to the second loop (not shown in Figure 9).

We are currently implementing two prototypes for the third level of PlanAni; one for generating animation commands for simple Pascal programs, and the other for simple C programs. The input for both prototypes consists of programs that are annotated with roles and role changes in the style of Figure 10 where both variables are initially \texttt{most-recent holders}. When the second loop begins, the role of the variable \texttt{count} changes to \texttt{stepper}. Since the last value in the old role is used as the first value in the new role, the role change is marked as proper. With these annotations, the prototypes will be able to automatically produce the animation commands of Figure 9.

Finally, the uppermost level of PlanAni will accept programs without role annotations. It makes a data flow analysis and assigns roles based on this analysis. PlanAni can be downloaded through the Roles of Variables Home Page at http://www.cs.joensuu.fi/~saja/var\_roles. There are ready-made animations for five Pascal programs—four that are suitable for teaching elementary programming, and one that demonstrates the animation of various roles. These programs can be easily turned into, e.g., C or Java, and new programs can be added with a modest effort.

\section{5 Experiences on Using Role-Based Animation in Teaching}

We have conducted a teaching experiment during an introductory Pascal programming course at university level. This section starts by describing the way the experiment was organized and the methods used in teaching. Then, we will discuss experiences in using PlanAni and present a short overview of other results of the experiment.

\subsection{5.1 Procedure}

The subjects of the experiment—ninety-one Finnish undergraduate students studying computer science for the first semester—were divided into three groups that were instructed differently: in the traditional way in which the course had been given several times before, i.e., with no specific treatment of roles (the traditional group); using roles throughout the course (the roles group); and using roles together with the use of the animator in exercises (the animation group). The course lasted five weeks, with four hours of lectures and two hours of exercises each week. There were three exercise groups for each experimental group.

During exercises, all groups animated four programs; one program in each exercise session except the first session. For animation, the animation group used PlanAni and the other groups used a visual debugger (Turbo Pascal v. 7.0). All exercise sessions of the roles group and the animation group were supervised by the same teacher.

Roles were introduced in the lectures for the roles and animation groups systematically throughout the course. Each time, when
a new role appeared the teacher presented the informal definition and
mentioned special cases covered by the exact definition. Later,
whenever a role reappeared in some example program the teacher
repeated the informal definition and explained the life cycle of the
variable. In example programs, the declaration of each variable was
followed by a comment stating the role of the variable and its use in
the program. Students were also given a printed list describing
all roles (4 pages).

In exercises, students first presented their solutions to home as-
signments. Animations, lasting between 20 and 40 minutes, were
always used at the end of the sessions. When students saw the sys-
tem used for animation (whether PlanAni or debugger) for the first
time, the teacher explained the user interface and gave instructions
for using the system.

In each session, the teacher first presented the animation step by
step using her computer and a video projector. As each role ap-
peared for the first time, she explained what the role image was and
how it tried to visualize the most important properties of the role.
Students were then instructed to run the animation using given data,
carefully selected by the teacher. Thereafter, students animated the
program with their own input data. Finally, the teacher discussed
with students about complicated issues or other problems students
had in understanding the program. All the time, students were en-
couraged to proceed slowly with the animation and predict the ef-
fect of the next statement on the values of variables and other as-
pacts of the program.

Novices have little grounds to interpret visualizations in the an-
ticipated way [Mulholland and Eisenstadt 1998]. We were there-
fore afraid that seeing past and future values in role images might
give students the impression that all these values are available and
could be accessed using some special syntax. Therefore, the teacher
stressed constantly during animation sessions that only one of the
values does exist in reality and that the others are shown for vi-
ualization purposes only. Students’ comments did not reveal any
misunderstanding in this respect.

Subjects attended a post-test after the course. Both in the mid-
dle and at the end of the course, some students were, moreover,
given program comprehension and program creation tasks which
were videotaped. The detailed analysis of this experiment will be
published elsewhere, and here we concentrate on our experiences
in using PlanAni, and compare it to the debugger. This analysis is
based on a semi-structured interview with the teacher who super-
vised all animation sessions for the roles group (using debugger)
and the animation group (using PlanAni).

5.2 Results
Working with PlanAni proved to be a positive experience to the
students: discussions were more lively than in the sessions us-
using the debugger, and students worked longer with PlanAni; other
activities—like web surfing—were more frequent with debugger
users. Even though there were some technical problems with
PlanAni, its users found the system easy to use while debugger stu-
dents complained that the debugger was hard to use. The spacious
layout of variables in PlanAni helped students to locate variables,
and role images increased distinction between variables as com-
pared with the debugger. Some students gave positive feedback on
PlanAni and stated that role images are “really good and clarify-
ing”: nobody criticized role images.

PlanAni users concentrated more on variables while debugger
users spent most of their time following the program code. As a
consequence, debugger users got a better understanding of the ac-
tions in the program code but PlanAni users got a better under-
standing of the total effect of the program and how each variable
contributed to this. Even though PlanAni flashes each code frag-
ment before animating its effect, students appeared not to follow
program code. We hope that using speech synthesizer during code
enhancement (as opposed to the current solution based on pop-ups
that hide part of the program behind them) will direct users’ atten-
tion more on the program code.

Due to role-based animation of operations, PlanAni uses several
ways to animate assignment and comparisons. Students made no
comment about possible confusion raised by this variability. The
animations are selected carefully to correspond to the properties of
the role, and this selection thus seems to be successful.

One program contained a one-way flag that was originally \texttt{true}
and set permanently to \texttt{false} when a certain condition was en-
countered for the first time (see the variable \texttt{pail} in Figure 7). The
role image for this role is a glowing light bulb that bursts into small
pieces when the value of the variable changes. In the debugger
user group, students had difficulties in finding out that the variable
could not regain the value \texttt{true} any more and, moreover, they had
problems in understanding how the expression in the assignment caused
this effect. In the PlanAni group, the animation made it clear to the
students that the value had changed permanently, but even these stu-
dents had problems in understanding the expression in the program
code. It remains an open question how the execution of Boolean
operators could be visualized to improve understandability.

One of the programs contains a fixed value whose value is cor-
rected (by taking its absolute value) just after initialization. This
raised vivid discussions as the informal definition does not cover
this case—albeit implications of the exact definition had been re-
peated in the lectures several times. These discussions were not,
however, frustrating but clarified the role concept and various uses
of variables in programs.

Students found the four programs instructive and interesting,
though some students questioned the Fibonacci sequence program
as they could not figure out any use for the Fibonacci sequence. In
the teacher’s opinion, the example programs were appropriate for
students attending an introductory programming course: programs
were simple—the palindrome program in Figure 7 being the most
complicated one—but more complex programs would have been
too hard for students at this level.

Finally, the teacher found PlanAni a better tool than the debugger
for explaining how programs work. In particular, it was much easier
for her to use roles in her speech—thus strengthening learning of the
roles—because role images aided her to remember the role of
each variable and there was no fear of mixing up the variables.

The first results of the post-test suggest that the introduction of
roles improves program comprehension and program writing skills
[Kuittinen and Sajaniemi 2003]. Moreover, the use of PlanAni af-
facts the way students describe programs: they stress data-related
issues concerning the deep structure of a program, as opposed

to directly visible operations and control structures. Previous re-
search has shown that summarizing programs using deep struc-
tures is an indication of superior programming skill [Détienne
1987; Pennington 1987]. For example, Clancy and Linn [1999] cite a
study demonstrating that code reuse—which demonstrates expert-
like programming skill—was substantially more common for stu-
dents who gave data level program summaries. Thus, the use of
PlanAni in exercises seems to foster the adoption of expert-like pro-
gramming strategies.

6 Conclusion
Learning to write computer programs is a hard task for many stu-
dents and research into techniques that could be used to help them
is needed. We have suggested that the role concept may be used to
foster learning and presented a program animation system, PlanAni,
that uses role images for variables and role-based animation for op-
erations. Even though the role concept applies to programming in
general, PlanAni is intended to be used in teaching introductory
programming to novices, only. We also presented experiences on using PlanAni in an experiment comparing traditional teaching with role-based teaching and animation. The results indicate that the introduction of roles improves program comprehension and program writing skills, that students like to work with the animator and that the system clarifies many concepts in programming.

Future research on roles will include analysis of roles in both correct and incorrect novice programs, roles in experts’ programs and in other programming paradigms (e.g., object-oriented programming), and cognitive studies to investigate if roles are truly part of experts’ knowledge. We will also study the effects of role-based animation vs traditional program visualization in more detail.

**Acknowledgments**

The authors would like to thank Elina Rüüsänen for supervising the exercises, Mikko Korhonen and Ville Rapa for programming PlanAni, and Anu Nevalainen and Aki Rissanen for graphic design of the role images.

**References**


