Designing Effective Program Visualization Tools for Reducing User's Cognitive Effort

M. Eduard Tudoreanu[*]
University of Arkansas at Little Rock, USA

Abstract

Program visualization holds great potential for conveying information about the state and behavior of a running program. However, barriers exist to the realization of this potential, and the limited knowledge about the factors that affect program visualization makes the identification of these barriers difficult. We present arguments that the economy of information and tasks related to the visualization environment has a significant impact on the user's performance in solving algorithmic problems. We apply this knowledge to develop an approach for creating application-specific visualizations solely through interactions with program visualizations and textual views of the computation, thus promoting economy of interaction. The approach consists of a multiple technical contributions that are surveyed in the paper.

CR Categories: D.2.5 [Software Engineering]: Testing and Debugging—Debugging aids; H.1.2 [Models and Principles]: User/Machine Systems—Human factors, Human information processing, Software psychology; H.5.1 [Information Interfaces and Presentation]: Multimedia Information Systems—Animations, Evaluation/methodology; I.6.8 [Simulation and Modeling]: Types of Simulation—Animation, Visual

Keywords: Program visualization, cognitive economy, automatic presentation, user studies

1 Introduction

Program visualization, also termed algorithm animation, is a form of presenting the execution of a running computation through the use of animated graphical displays. Such animations visually encode and present how data is processed inside a running program. Current trends in software engineering and visualization suggest that a demand for program understanding techniques exists and that graphics have the potential to supply significant help. Current and future software packages demand new and improved techniques, beyond traditional software engineering and formal analysis approaches, to cope with the sheer complexity and size of code while maintaining an acceptable level of reliability. Program visualization can play a role in bridging the gap between human reasoning and computational processes from the early stages of training when graphics can be used to give students extra insight, to production settings when visual aids can be used for programmers and administrators who seek to manage large software, and even to end-users who must become more and more aware of the high-level mechanics of the software they employ. Computer graphics has proven to be an effective conveyor of information in the form of scientific and information visualization.

Despite program visualization's expected potential and the need for richer tools to connect people with computations, animation is largely underutilized. Since the introduction of program visualization tools, program visualization technology has undergone significant advances, yet our knowledge of the barriers that need to be overcome to make animation effective for users is still limited. The extent and details of the program visualization's problems are not entirely clear and by no means simple. This might explain the mixed results of empirical studies into the practical importance of algorithm animation [Byrne et al. 1999; Hansen et al. 1998; Kehoe et al. 2001; Lawrence et al. 1994; Stasko et al. 1993]. The limited understanding of the barriers to effective animations may lead to program visualization environments that include only a few elements to enhance the performance of animation users.

This paper identifies the management of user's mental load as the underlying theme for a class of barriers to increased visualization effectiveness, and describes a collection of technical contributions that are glued together into a visualization environment capable to address these barriers. Rather than present the details of individual contributions, which can be found elsewhere [Tudoreanu 2002], the merits of this writing reside in painting a comprehensive view of an approach to program visualization centered on increasing the role of animation as a problem solving tool.

The problems of program visualization that we concentrate on stem from a poor *cognitive economy* that is induced by the entire settings, based on algorithm animation, in which a user solves a problem. *Cognitive economy* seeks to minimize the load on the cognitive system and includes two competing directions. First, cognitive economy aims at reducing both the amount of information handled by the user and the user tasks that do not pertain to the observed computation but solely to the animation and environment. Making sense of, building, and refining the animation are tasks that generally do not help the user make progress toward solving the problem; the result of these tasks are beneficial. Second, cognitive economy's goal is to maximize the information pertinent to the user problem that the visualization conveys, which may be opposite to the goal of reducing the complexity of the user's tasks related solely to manipulating visual displays (consider searching, navigating, or refining a picture that contains all available data on a subject). Ideally, the picture will depict all and only the information the user's changing tasks require, and do so without requiring the user to search, refine, or decode the subset of data that is currently needed.

Our approach to improving cognitive economy consists of three directions for reducing the mental load due to the introduction of animation in the user's environment. The first direction focuses on removing indirect representations, such as programs for visualizations themselves, that tend to be a part of the visualization process. In our approach, users build and refine visualizations through

[*]e-mail: metudoreanu@ualr.edu

interactions performed directly on the graphical representations of the monitored computation. The second direction includes an automatic presentation system capable of determining the values for the graphical attributes of the visualization. The automatic presentation relieves the user from defining the functions that take data and calculate graphical values. It suffices for the user to specify the information and properties to be observed, and the system can create a view customized to that information. The third direction consists of enriching the visualization with an explicit representation of the *visualization syntax*, the mapping between changing entities of the computation and the dynamic graphical elements. The syntax is displayed via legends, which are interactive and allow the user to continuously adjust the appearance of the program views.

Cognitive economy is also achieved by extending the range of visualizations that can be constructed by the user. We augmented our environment with a model of interaction that permits users to derive information from multiple states of the computation and even from other views of the computation. Flexible temporal behavior of animations in response to changes in the state of the computation can be specified by the user. As such, the user can keep interesting past information in the visualization instead of attempting to remember that information.

Next section details the barriers that are related to cognitive economy and presents empirical findings that seem to support the critical role of such economy. Section 3 goes through a simple scenario for building an animation. The interaction model that allows the creation and refinement of animations without the need to rely on indirect structures is the subject of Section 4.1. Temporal behavior and automatic presentation techniques are described in Section 4.2 and Section 4.3. Interactive legends are briefly presented in Section 4.4, followed by related work.

2 Cognitive Economy Critical for Effective Program Visualizations

2.1 Problems Related to Poor Cognitive Economy

The importance of cognitive economy can be deduced from the theoretical advantages of graphical representations. The benefits of graphics as a presentation medium are two-fold, as Bertin notes [Bertin 1983] in his book "Semiology of Graphics", as a tool for conveying large amounts of information and as an environment for solving logical problems. Scaife and Rogers [Scaife and Rogers 1996] view graphical representations as a form of external cognition, in which mental internal representations are offloaded onto an external medium to relieve the cognitive burden and speed up processing. Visualization extends the cognitive resources of the user. Note that these considerations are applicable to graphics in general, and it is unclear whether users of program visualization take advantage of animation in this manner.

One immediate observation is that visualization may be of little use if packaged with an environment that increases cognitive load because the reduction in cognitive resources provided by the picture is offset by the environment. In other words, a situation in which both graphics and high cognitive load exist resembles a situation when neither is present. Hence, a visualization environment that requires users to handle additional information and tasks, which increases cognitive load, offers similar performance advantages to that of a user who has no visualization at all.

Poor cognitive economy for users trying to solve a problem may occur both when the visualization requires them to perform tasks and handle information not related to the problem at hand (see **A.1.** and **A.2.** below) and when the visualization is not customized for the problem (see **B.**).

A.1. The creation and refinement of an application-specific visualization often relies on indirect structures, such as specialized languages with their compilers, or dedicated graphics packages and editors [Brown 1988; Roman et al. 1992; Stasko and Kraemer 1993]. These indirect structures create a barrier by consuming the cognitive resources that the user devotes to understanding and handling the structures and tools. A schematic of such an interactive visualization process is shown in Figure 1. Compare this with our approach summarized in Figure 2.

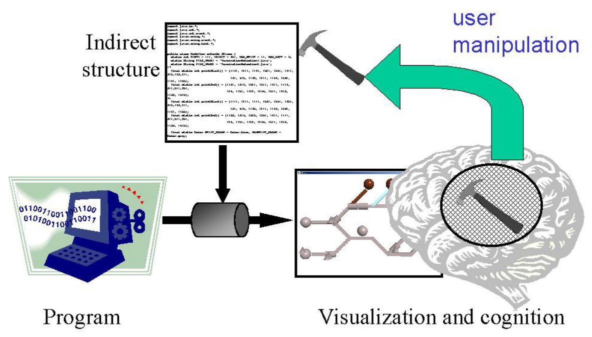


Figure 1: The cognitive capacity is enhanced by the visualization, but cognitive resources must be diverted (hashed area of the brain) from understanding the program toward handling indirect structures. These structures control how the program is depicted.

A.2. Algorithm animations might employ numerous concepts and representations to present the abstract data and mechanics of a computation. To accomplish any task, the user must expend additional effort for understanding what the animation is presenting during its execution. If the visualization syntax is unknown, the visualization is perceived by the user as a set of randomly changing shapes and colors. Unfortunately, in classical program animation, no provisions are made for reducing the time and cognitive resources diverted toward learning the syntax.

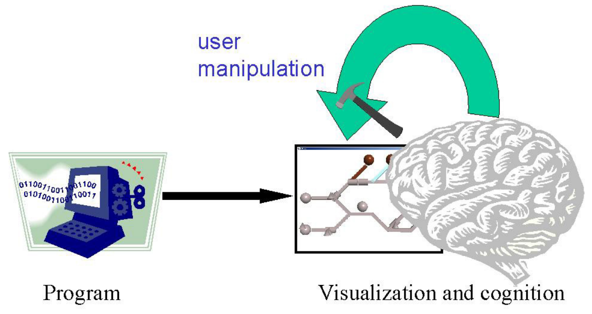


Figure 2: The running program can be observed through a visualization. The visualization can be customized directly via user interactions, leaving most cognitive resources available for solving program related problems.

B. Pre-defined animations are limited in the range of tasks with which they can efficiently assist the user. Although such views are easy to obtain and quite helpful for their intended tasks, they can quickly become a burden as the task shifts. An attempt to apply a pre-defined visualization to a different task, even if derived from the original task, might force the user to devote substantial cognitive effort to associating the information in the visualization to the information required by the task. This might have the undesirable effect of increasing the data that has to be mentally stored and might result in an increase in the time and cognitive burden to perform the task. In such a situation, the task of understanding the program is shifted to understanding and applying the relations between the available visualization and the problem to be solved. The mere existence of an animation does not lead to reduced cognitive load.

2.2 Empirical Evidence

Cognitive economy is quite possible the cause of different results registered by two similar user studies. We briefly describe the studies, compare them, and list the differences between the animation environments employed in the experiments.

The studies were designed to answer the question "Does the addition of a visualization in the user's environment promote understanding of distributed computations?" One study failed to show any significant benefit when visualizations were used to answer questions about distributed computations. The other study, in which both the animation and the testing environment were designed in light of the concept of cognitive economy, showed a clear benefit for visualization users. Performance of the users who did not have access to visualizations was similar across the studies, providing us with the opportunity to compare these studies and to postulate possible causes of improved performance of visualization users.

The two studies are similar in that they employed the same computation (Dijkstra-Scholten's termination detection [Dijkstra and C.S.Scholten 1980]), user profile and type of tasks to be performed by the participants. Moreover, about half of the tasks, which consisted of answering questions, were identical in the two studies. Both studies presented users with a one page description of an algorithm, an interactive environment for answering questions, and the code of the algorithm. The participants were able to run the algorithm and examine its execution via a textual output produced by print statements and, in about half of the cases, via an animation. The methodology of both experiments allowed users to examine the visualization and any other learning material while completing their task (answering questions). Most previous experiments focused on learning, and the participants underwent a training session to examine the learning materials (including visualization), followed by a test session to answer the questions.

In experiment $\mathscr{A}$, the program animation was based on a three-dimensional world. The meaning of the graphical elements employed in the visualization was explained on a piece of paper handed to the participants at the beginning of the visualization session. One characteristic of the visualization was that the point of view in the three-dimensional world could be modified by the user via mouse controls built into the Java 3D distribution. Changing the point of view helped users in determining the parent-child relationship, which was encoded on the z-axis; the position of the parent was below that of its children (see Figure 3).

In experiment $\mathscr{B}$, the program animation was also based on a three-dimensional world. The main difference between this view and traditional program visualization is the addition of a legend. Instead of talking to users or having them read a description of what the graphical attributes encode, we added the legend to the visualization. We hoped that the legend would reduce the amount of information the user must remember. One difference from experiment $\mathscr{A}$ was that the point of view in the three-dimensional world was fixed and the users could not change it, which reduced the number of tasks to be performed by the user (see Figure 4).

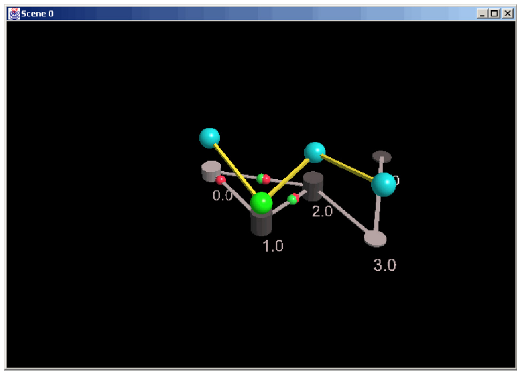


Figure 3: A snapshot of the termination detection visualization from experiment $\mathscr{A}$.

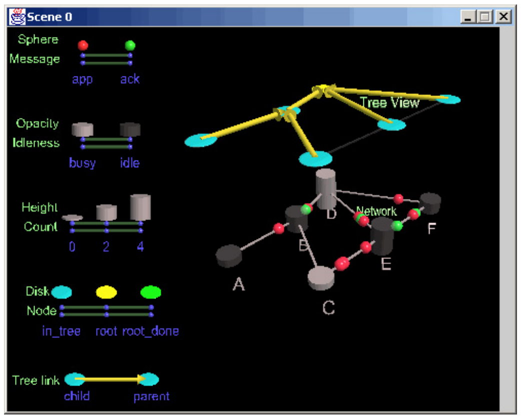


Figure 4: A snapshot of the termination detection visualization in the second study. The tree that spans the distributed network is presented above the network topology. A legend is displayed on the left.

Analysis of the Common Questions in the Two Studies Although experiment $\mathscr{A}$ failed to find an improvement for the situations in which visualization was employed, the collected data can be analyzed to determine whether the effect of visualizations was different between the two studies. The data to be analyzed was restricted to the questions that were common to both studies, five in number. The rate of correct answers was computed for each such question in one experiment and then in the other. This rate was further used in an analysis of variation.

A difference was found for the visualization situations, while the text-only sessions were comparable across the experiments. We found that there is a significant, though not very significant, increase in performance (number of correct answers) for the visualization situations in experiment $\mathscr{B}$ ($F_{1,4} = 7.87, p = 0.0485$). This suggests that the factors listed in the next paragraph are important for animation users. For the non-vis groups, no statistical difference was registered between the two studies ($F_{1,4} = 0.62, p = 0.4745$), although in study $\mathscr{A}$, the performance of the text-only situations was slightly higher than in study $\mathscr{B}$.

Differences Between the Experiments Only differences that relate to cognitive economy and perceptual issues are given here. Additional information about factors that reduced noise in the experiment design, and about those that were most likely irrelevant can be found in [Tudoreanu 2002]. Perceptual and cognitive economy factors are listed next. 1) Legends were used in $\mathscr{B}$, while a hard copy, textual description of the visualization was used in $\mathscr{A}$. 2) Navigation in $\mathscr{B}$ was improved by pre-arranging windows on the screen, using two monitors for the visualization group, and adding navigation buttons that allowed jumping to the desired window. 3) In $\mathscr{B}$, arrows presented the parent-child relationship as opposed to increased height on the z-axis and capability to change the 3D point of view in $\mathscr{A}$. 4) *execute*-type questions were only presented in study $\mathscr{B}$. Such questions inquired about a specific *execution* of the program, and it is likely that the user had to run and observe that execution to gather all the information referred to in the question. The other questions were more *general*, in the sense that they were referring to properties common to more than one execution of the program, and could have been answered merely by reading the question. Given the space constraints, examples of the two types of questions are left for other publications [Tudoreanu 2002; Tudoreanu et al. 2002]. 5) In study $\mathscr{B}$, an interactive code view allowed limited filtering of the output. 6) The continuous play of the animation depicted only one step at a time in $\mathscr{B}$ as opposed to multiple steps being shown at the same time to convey the parallelism in the computation in $\mathscr{A}$.

Factors Related to Cognitive Economy Factors one through five can be regarded as promoting cognitive economy. Legends free the users from learning and remembering the mapping between abstract concepts in the program and the graphical features of the animation. Navigation, although typically a trivial task, becomes distracting and consumes cognitive resources because it is performed frequently (every time the user is interested in another perspective on the computation or problem). Eliminating the need, and the possibility, to modify the point of view eliminates a task that has little to do with the user's problem.

Factors four and five contribute to reducing the mental processing of the information learned from the animation and code view. First, *execute*-type questions do not require the user to observe a number of computations, infer the general properties of the algorithm, and then apply them to the particular question. Instead, the user observes exactly the execution referred to in the question. Second, filtering tools reduce the task of searching through the information in the code view and of mentally putting together relevant data that is cluttered with irrelevant information. Although this feature did not seem to improve the text-only group, it might have played a role in the success of visualization groups. Participants using visualizations could take advantage of the searching capability and further reduce the utilization of mental resources.

Perceptual Factors The continuous mode of showing the animation (factor number six) exposed the viewers to a number of simultaneous changes in the program view. It is possible that the users could not track all these changes, especially in random executions with a high degree of parallelism. The importance of this factor is attenuated by the existence of a common step by step mode of playing the animation in both studies. It gave users an option to observe the computation at their own pace when they realized that too many changes occur at the same time.

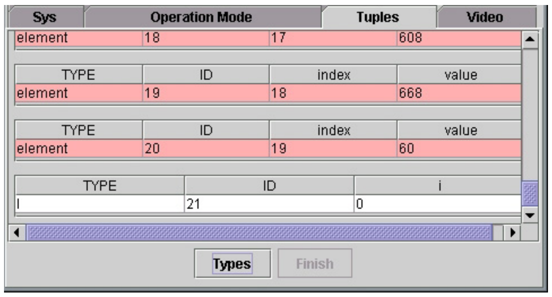
In conclusion, cognitive economy is the most likely cause of significantly increased user performance. Thus, it is important to design program animation tools and environments that reduce the amount of data and the tasks required in a visualization session. In other words, care needs to be taken to avoid increasing the cognitive load through the mere introduction of animations in the user's environment, and to allow the user to concentrate on the intrinsic data and complexity of the problem and algorithm at hand.

3 Building a Visualization under Cognitive Economy

The sample visualization session presented next demonstrates that animations can be created with no coding and virtually no computation of graphical attributes. The user specifies only the data of interested and the relations among the data that are to appear in the animation. Query-like operations provide users with a powerful and efficient manner of selecting information and determining relations. Finally, the user operations have a temporal dimension, which is very important for creating expressive animations that explicitly relate information from multiple program states. Views of multiple points in time relieve the user from having to remember the past features of the running computation.

We elaborate next on the manner in which a simple visualization can be constructed. The fundamental features that may improve cognitive economy are pointed out throughout the example. A movie captured while the example was being performed can be downloaded from `http://ifsc.ualr.edu/metudoreanu/demo.mpg`. The interface of our implementation is kept to a minimum in order to emphasize more fundamental aspects of algorithm animation creation and refinement. Finally, our implementation happens to use a three-dimensional world for visualizations, but similar functionality can be provided in a two-dimensional environment. In the example below, the third dimension is never used.

From the user's perspective, the computation appears as a set of variables whose values change as the program runs. The values of the variables, the state of the computation, are assumed to be collected and fed into the visualization tool by a monitoring system such as PathFinder [Hart et al. 1999]. The user can observe the state and select the variables of interest in a pre-defined textual view that presents each variable as a tuple (Figure 5). The tuple format was chosen because it is flexible enough to encode both basic data, such as scalar variables or machine registers, and composite variables such as structures and objects with multiple fields.



element	18	17	608

TYPE	ID	index	value
element	19	18	668

TYPE	ID	index	value
element	20	19	60

TYPE	ID	i
i	21	0

Figure 5: The state of the computation is shown as a set of tuples.

The computation, as it can be seen in Figure 5, has an array of size twenty named 'element' and an integer named 'i'. The user selects all array elements and then adds them to an empty visualization. Thus, each graphical object stands for an array element. The user can click on an element and examine more closely the tuples for which a visual object stands for. Simple queries, such as the selection of all elements at once, can be performed by the user to select all items with a common property through one action. Queries can reduce the number of tasks performed by the user.

The user was not required to calculate graphical values, which reduces the amount of data and the complexity of operations that are related exclusively to graphics and not to the observed computation. Figure 6 shows that the computer automatically generated a picture to display all elements of the array as specified by the user. The layout is random because the user did not instruct the visualization to show any relation or property of the elements. The random layout is one solution for avoiding conveying false relations. The automatic presentation algorithm also ensures that graphical objects do not overlap, which also prevents conveying false relations.

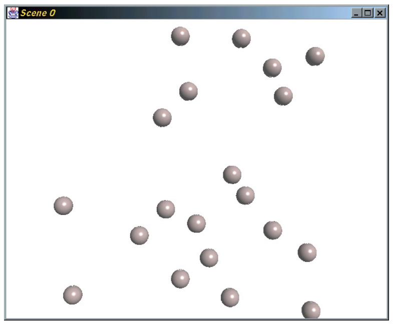
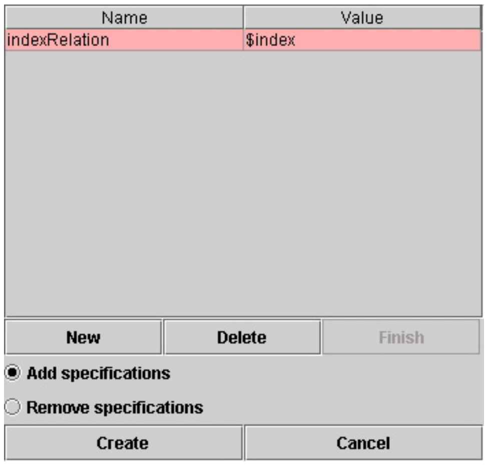


Figure 6: The elements of an array are depicted in a random layout. Each element is conveyed through a separate graphical object. No relations are shown among elements.

This automatic presentation algorithm is able to augment an animation with additional relations between elements as well as with additional objects. The image is constructed by composing simple graphical objects whose graphical attributes are set in such a way to present the relations requested by the user. Thus, the final animation is not dependent on a pre-defined pattern, but is customized to the current computation and user specifications. Cognitive economy is improved because no information that is deemed unnecessary by the user clutters the display.

Next, the user requests that the elements be ordered according to their index. This can be done by specifying a relation among array elements as shown in Figure 7. Again, the automatic presentation algorithm ensures that objects do not overlap Figure 8. Likewise, the relation defined by the values of the elements is added to the visualization. The user names this relation ‘height’, and the automatic presentation sub-system chooses the scale on the y-axis to visually convey the relation.



Name	Value
indexRelation	$index

Figure 7: A new relation named ‘indexRelation’ is entered by the user.

The user can always override the system’s graphical decisions. For aesthetic reasons, cylinders are better at presenting the elements of the array. Figure 9 shows the visualization where cylinders replace the system-chosen spheres. The interactive legends of Section 4.4 provide a direct interaction manner of overriding the automatic presentation algorithm and of customizing the animation (for a demonstration of interactive legends, visit `http://ifsc.ualr.edu/metudoreanu/legend.wmv`). The user can now start the computation and observe its execution. Note that only the current state is shown at any given time.

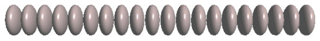


Figure 8: The elements of an array are shown in the order in which they appear in the array.

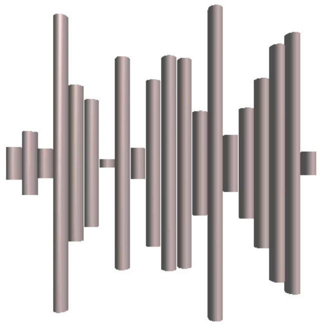


Figure 9: The elements of an array are depicted as cylinders. Two relations are shown: the order in which elements are stored in the array by x-position, and the value of each element by height.

The hypothetical user is interested in observing how the elements that now have small values will perform in the future. The user selects and highlights the elements that have values smaller than 300. This action is performed in a different temporal mode, termed absolute, than all other actions. The absolute mode will always depend on the same state, and therefore the same elements will be highlighted in the future, regardless of their future values. The display will keep emphasizing those objects that are selected now.

4 Reshapeable Visualizations

In the program visualization environment (Figure 10), users can build graphical representations of a computation through direct manipulation of animations as demonstrated in the previous section. The information and structure of a visualization is defined by user actions on *scenes*. A scene is the underlying data presented in a window. Visualizations corresponding to those scenes are automatically drawn by the automatic presentation sub-system. Users can intervene and determine entirely or partially the graphical elements to be used.

The architecture, depicted in Figure 10, consists of two modules, implementing a two-step mapping of program state to graphics. The first module generates scenes based on the user actions, which also control the temporal behavior, and the state of the computation. The second is a rendering system capable of realizing each scene as an actual image. The user can adjust the graphical features chosen by the automatic presentation algorithm.

The functionality of this architecture is twofold. First, it enables the encoding of the changing program state into a visualization. As the computation runs, the scenes are updated to reflect the new program state. A scene modification prompts the automatic presentation sub-system to redraw the image of that scene. Second, the architecture also allows users to interact with the visualization by performing operations on scenes and to control the automatic rendering.

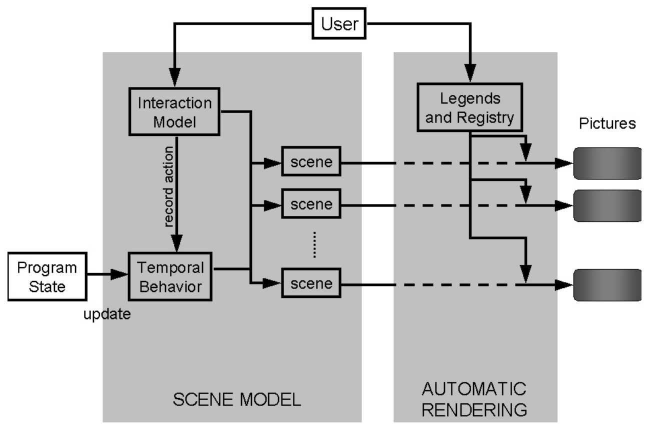


Figure 10: The framework of the program visualization environment.

4.1 Interaction Model

We designed an interaction model that allows uniform operations regardless of the interaction devices and graphical features of the visualization environment. The goal is to circumvent the use of graphical attributes in the creation and modification process and to base the interaction on the information content of a program view. We have developed an abstract structure termed a *scene* that captures the data and relations portrayed in a picture, but does not concentrate on graphical features. Any program view can be captured as a scene, and a scene can be transparently transformed into a visualization with the techniques in Section 4.3. As such, the user can create or refine a scene via interactions with a visualization.

We introduce the scenes from the perspective of a system designer rather than a visualization user because the former supports a more precise definition. The role of a scene is to provide an exact description of the information in a static program visualization. Consider Figure 9 which presents a number of objects, in this case cylinders. These objects have certain properties, and some of the properties suggest relations between objects. Moreover, each graphical object has a meaning in that it represents some entity or entities in a computation (an element of an array). The same picture can be used, in one instance, to present an array, or in another instance, to present a matrix by depicting each row as a cylinder. The particular meaning of the objects is the distinguishing feature of a program visualization compared to other types of visualizations.

Based on the previous observations, a scene is defined as a set of *scene elements*. Each of the elements is a pair of *structural specifications* and *data specifications* because it both has a number of properties and represents some subset of the state of the observed computation. The properties of the elements define a structure among scene elements, and we termed that part *structural specification*. The subset of the state that each scene element depicts is termed *data specification*. Typically, a one-to-one correspondence exists between scene elements and graphical objects, and each structural specification is depicted through a graphical feature such as color, width or adjacency.

4.2 Temporal Behavior of Visualizations

The interaction model is designed to allow users to define animations with a rich behavior in relation to the evolving program. More precisely, animations are sometimes richer if they behave slightly differently than the monitored computation. It may be advantageous to maintain some representation of the previous states of the computation and to show the current state in the context of the previous states. This may visually reveal causal relationships and patterns that would otherwise be lost unless the user memorized old states. Reducing the amount of information to be memorized promotes cognitive economy.

Users have the flexibility to determine how the visualization responds to changes in the program state by specifying the behavior of individual operations over time. As such, two users observing the same computation and building the same initial view can define different animation behaviors. These animations will present the program differently in the future without the need for further user intervention.

In a static context, for one state of the computation, a user operation on a scene can be given precise semantics in terms of what scene elements and specifications are modified and what their new values are. These values are specified as a function of the current program state and of the current content of the scene. An interesting observation is that, in the context of an evolving computation, an operation can be re-applied in multiple ways based on what set of states, current or past, the operation is considered to depend on. Thus, the user can have the flexibility to choose the states upon which the operation will depend on in the future.

We propose and develop three basic modes of re-applying an operation: absolute, relative and cumulative. The mode is given by the user at the time the operation is first executed. For simplicity, only operations that perform selection, i.e., choose operands for future processing, are allowed to have any mode, all other operations work in relative mode.

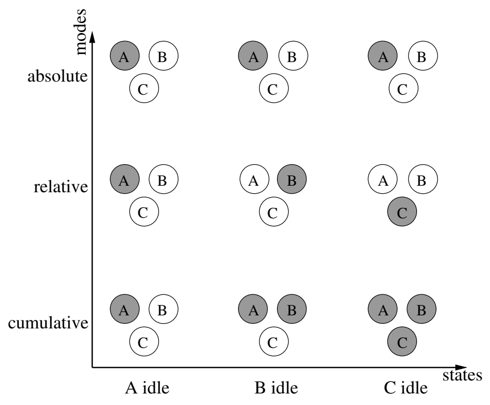


Figure 11: A row presents a visualization during three states of a program. Initially, only A is idle, then only B and then only C. The shading of the idle nodes is applied in a different mode in each row.

For example, suppose that the user wants to highlight idle nodes in a computation that has three processes, and they become idle in the order A, B, C with only one idle at a time. The first time the operation is executed, the marked node is A in all modes (see Figure 11, first column, which corresponds to the static behavior). After the program state has changed, one possible behavior of interest to the user might be to keep marking process A as a reference point that A is the one that was idle. The animation must display the same process that was initially idle, A, even if it is not idle anymore (first row of the figure). Another possibility would be to ignore the past and show the process that is idle now, node B (second row). There is yet another option: to present both processes that were idle and the ones that are idle. More precisely, processes that were idle in any state between the initial and the current one. This mode allow easy tracing of momentary properties. Of course, the user can input the same operation twice in different modes, such as absolute and relative in order to observe both the node that was initially idle and the current idle ones.

4.3 Automatic Presentation

This section presents the technique for creating graphical representations of programs based on data that streams from a running program. Our approach is customized to handle data that consists of a number of heterogeneous program entities. These entities are data objects or variables, such as integers or queues, that are manipulated by the monitored program during its execution.

The constraints that must be satisfied by the automatic presentation algorithm can be described informally as a set of three goals for scene depiction. These goals appear as common features of classical examples of program visualization, such as Stasko's POLKA and TANGO visualizations of sorting and bin-packing algorithms [Stasko 1990; Stasko 1998], Pavane visualizations of trains, ATM Networks, and elevator controls [Roman et al. 1992], Brown and Najork's animation of a shortest-path algorithm [Brown and Najor n. d.], and Baecker's depiction of sorting algorithms in his "Sorting Out Sorting" video [Baecker and Sherman 1981]. We note that these program visualizations have a different flavor from the ones created for program optimization.

The goals are: **Expressiveness** Display all the entities and relations of the scene and nothing else. Consider the monitoring of a generic distributed computation. For a given state, a scene describes three computers and one message as entities as well as their x and y position as relations. According to this goal only the four entities are to be drawn and they should be placed according to their position. **Visual Classification** Distinguish between different kinds of entities. In the previous example, messages and computers would be drawn differently to permit a viewer to easily infer the type of each object. The visual appearance of one data category should be significantly different from the appearance of another's. A graphical attribute like color or length is typically insufficient. Consider the case of a visualization that presents every entity, computers and messages, as spheres with a different color for each category. Such a visualization fails to emphasize the category of each object and makes message passing subject to misinterpretation as computer movement. A better visualization might depict the two categories as two different types of graphical objects, such as small spheres for messages and cubes for computers. **Continuity** Ensure that consecutive depictions of the same scene present similar data using common graphical features and unrelated data using different features. For example, once a message has been presented as a sphere it will continue to be presented as such at a later time. This also prevents a mobile computer that enters the environment of interest from being depicted as a sphere even in a state when no messages, and consequently no spheres, are displayed.

Visual presentations of a scene are obtained from a combination of graphical elements. The available types of elements and a description of their function are located in a *style gallery*. The gallery contains definitions for graphical objects whose appearance was previously designed either as built-in objects of the system or as custom representations for a monitored computation. The gallery provides a characterization of each object type and of the graphical attributes of the objects. A graphical attribute is a variable visual feature of an object.

Scenes are translated into graphics following a top-down, three-step framework as shown in Figure 12. In the first step, graphical objects are chosen to present the elements of the scene. The objects are based on the types that exist in the style gallery. In the second step, graphical attributes of the allocated objects are assigned to the structural specifications of the scene. Across the entire scene, all structural specifications with the same name must be depicted by similar attributes, e.g., color or length. This step is designed to ensure that all relations are shown distinctly, as required by **Expressiveness**, and to choose different types of objects for different types of program entities (**Visual Classification**). In the third step, values are generated for the attributes of the graphical objects. The values of the attributes that present a structural specification must make the relation defined by the specification visible in the appearance of the corresponding graphical objects. In some cases, values of the unused attributes, which present no structural specifications, must also be computed in order to satisfy the presentation goals, specifically to avoid hiding graphical objects (scene elements) and presenting false relations via object intersection (required by **Expressiveness**). A decision made in one step may lead, in a later step, to a graphical design that does not satisfy the presentation goals, in which case the process backtracks and an alternate decision is tried.

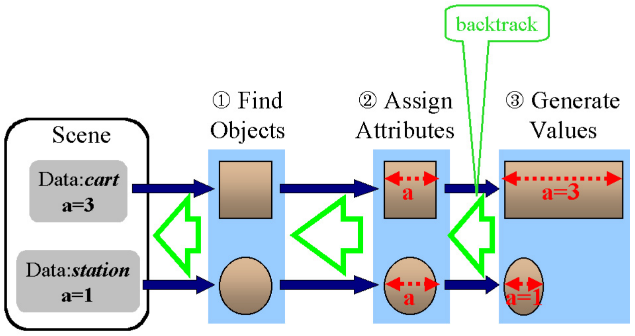


Figure 12: An example of a cart and station scene elements as they go through the three rendering steps. They have one structural specification named `a` and different data specifications.

The framework can produce graphical representations by analyzing the type of data and relations described by the structural specifications of a scene and by finding the graphical attribute that is most fit to present those relations, as employed in other automatic tools [Casner 1991; Mackinlay 1986; Roth and Mattis 1991]. Our approach, however, also makes use of the names present in the scene to build a graphical representation that is more likely to reflect the viewer's intentions. The naming of the program objects and their fields can aid the system in choosing the graphical object for a scene element. The names given to the structural specifications typically show the viewer's mental model for the final visualization. A structural specification that refers to a graphical property, such as `width=4`, is most likely intended to be depicted through that attribute.

A *registry* is employed to record previous assignments of graphical elements to scene components. When similar components appear in the future, the system chooses the visual elements recorded in the registry to display those components. This is the mechanism that maintains consistency over time (**Continuity**). It also speeds up the re-drawing of a slightly modified scene for which most of the graphical elements have already been decided.

The technique, although fit for presenting relations, has limitations in capturing the semantics of those relations. Generated visualizations might not show the representation expected by a viewer. The final picture is accurate in presenting all relations, but due to lack of knowledge or graphical attributes, an unintuitive visual attribute might be used to encode a property. The interactive legends presented in the next section provide a direct interaction device for users to adjust the visual representations chosen by the system.

4.4 Interactive Legends

Our visualization framework relies on legends similar to those commonly employed in cartography as a means of explicitly conveying the visualization syntax. A legend key is itself a graphical representation that allows a viewer to find the relations between information and graphics at a glance by simply examining the picture. Legend keys are familiar, easy to interpret by most people, and offer a compact representation of a mapping via linear interpolation. By varying the coarseness of the interpolation, legends can be naturally scaled to fit into the available space of a visualization.

In our approach, legends are regarded as an integral part of visualizations. As such, legends have a dual role, as both a presentation mechanism and an interaction element that can be modified, included or removed as a result of user actions and program execution. As a presentation instrument, legends have the potential to improve cognitive economy as in the experiments presented in Section 2. An interactive widget that is based on traditional legend keys is described next. The widget is expressive enough to convey the choices of the automatic presentation technique.

Basic Legends consists of a number of *keys*, each presenting a relation between data and graphical domains. Basic legend keys, such as the one in Figure 13, consist of a data thread and a graphics thread. Threads encode a domain of values and are generally depicted as lines. Threads convey either discrete points or continuous domains. For threads encoding discrete points, each value of the domain has a corresponding tick, while for continuous domains, the ticks sample a range in the domain.

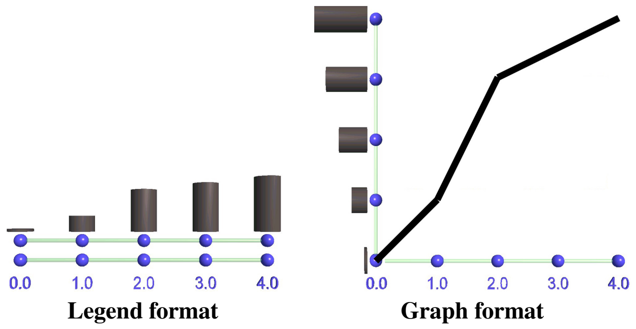


Figure 13: A segmented linear function is shown in both legend and graph formats.

A legend key is a brief representation of a mapping. Mappings are also customarily illustrated with a graph, but the legends may be better because they reduce estimation error and occupy a smaller space than the corresponding graph format, as seen in Figure 13. For a visualization user, it is important to establish how certain data is visually encoded, or conversely to estimate the data value that a graphical object is representing. For this kind of tasks, legends may reduce the estimation error because of the placement of the two domains next to each other.

Automatic Presentation of the Visualization Syntax The automatic presentation algorithm of the previous chapter can present the generated encoding via basic keys and coordinate axes. The algorithm maintains a registry in which three types of mappings are recorded: tuple types to graphical objects, structural specifications to graphical attributes, and values of the specifications to values of the graphical attributes. The first mapping is discrete and is depicted as a basic key with textual labels on the data thread and with sample graphical objects on the graphic thread. For the second mapping both threads have textual labels. Finally, the third mapping is presented through multiple keys, one for each graphical object–attribute pair.

Interaction with Legend Keys Keys can be manipulated through discrete operations, such as adding and removing ticks and threads, and through continuous adjustment of the mapping. The continuous manipulation is designed to update both the animation and the elements of the keys affected by a user gesture while the gesture is performed, to provide cues as to all elements of a mapping that are being changed, and to maintain the consistency of key, which include preserving the ordering of a continuous domain. Figure 14 shows an example of interaction.

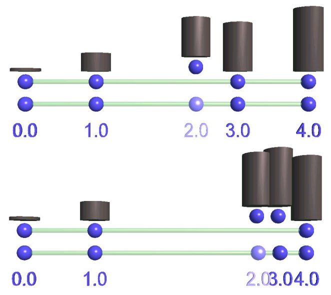


Figure 14: Snapshots of an interaction with the legend of Figure 13 by moving data tick 2.0 to the right.

5 Background

The notion of cognitive economy is a derivation of Scaife and Roger's external cognition [Scaife and Rogers 1996] for the particular area of algorithm animation, whose main task is to facilitate insight. This paper is not intended to create a model or theory for cognitive psychology, but a simple way, based on the load of cognitive resources, of improving the visualization environments. There is important research in cognitive psychology that explores more detail about how people handle programming, such as Green's cognitive dimensions [Green and Petre 1996] and Blackwell and Green's [Blackwell and Green 1999] attention investment. Cognitive dimensions, however, are focused and developed based on programming tasks, which are broadly the translation of a known solution (to a problem) into code to be executed. Programming seems to be different than the task of animation users: gaining insight. Attention investment is orthogonal, and perhaps complementary, to our approach being based on the user's focus of attention and risk assessment rather than on the economy of cognitive resources.

A wide range of tools, each built on different principles and with its own strengths and weaknesses, can be employed by a user who seeks to observe an animation. A number of comprehensive taxonomies of program visualization have been compiled by Myers [Myers 1990], Price *et al.* [Price et al. 1993], and Roman and Cox [Roman and Cox 1993]. However, from the perspective of our research, the primary feature of interest in classifying program visualization systems is their reliance on the user manipulation of indirect structures. The *indirect group* contains systems in which the visualization is defined by writing a piece of code or by constructing a visual program [Brown 1988; Crescenzi et al. 2000; Hibbard et al. 1992; Roman et al. 1992; Stasko and Kraemer 1993]. The *direct group* allows the users to obtain a view of the observed computation by manipulating the animation directly. For completeness, we consider that pre-defined program visualizations are part of the direct group because such animations do not require the construction or manipulation of any structure, and that *animation authoring* belongs to the indirect group. In animation authoring systems [Hamilton-Taylor and Kraemer 2002; Hundhausen and Douglas 2001; Roessling and Freisleben 2000], there is no running computation to be monitored, and the user focuses on the animation itself rather than on an existing program. Authored animations may or may not present an exact computational process. The main reason for classifying these approaches under indirect structures is the different paradigm the animation authoring entails. Moreover, it happens that most authoring tools expose users to coding.

Next, we focus on the direct group because the results in the indirect group differ fundamentally from our approach. Our empirical investigations suggest that under some conditions indirect structures may incur the risk of degrading the performance of the animation users in understanding a computation. The direct group is closer to our work, but the range of animations that can be obtained with the help of those systems is more limited than the range offered by our approach.

Direct manipulation of visualizations was explored as a means of eliminating the coding requirement. Mukherjea and Stasko integrated into a debugger a program visualization tool named Lens [Mukherjea and Stasko 1994]. The user can click on statements of the observed program and add visualization requests mainly by filling out various forms. A graphics editor is integrated in Lens to produce the graphical objects of the visualization. The values of graphical attributes can be smoothly animated by choosing among pre-defined transitions. Lens is not designed to construct a large range of animations, but instead is designed to permit easy creation of most used types of animations. Lens is further restricted by its dependency on the control structures of the program such as `if` or `while` blocks. Steve Reiss' BEE/HIVE [Reiss 2001] is another system for creating program visualizations in a direct manipulation environment. It consists of a visual query interface that can be used to process the data collected from a running program, and a number of visualization templates to convey the processed information. Although the query mechanism can help explore relations in the program state, the final appearance of the animation must rely on a pre-existing template.

Automatic presentation techniques appeared in the context of structured, static information such as database systems. An early approach, APT [Mackinlay 1986] by Mackinlay, was able to create a picture to present relations from a database. The algorithm ensures that all and only the data chosen by the user is depicted, and that the relations between values are shown by an appropriate visual feature such as color, shape or position. SAGE [Roth and Mattis 1991], created by Roth *et al.*, relies on a richer characterization of information than APT and is capable of creating visualizations that can present data more clearly. The University of Washington illustrating compiler (UWPI) [Henry et al. 1990] can automatically create visualizations for simple programs. UWPI recognizes a set of abstract data structures and has a built-in representation for these data structures.

Automatic presentation techniques, although requiring little effort from the user, are subject to a number of limitations which include the following. 1) Continuity of animations. Some of these systems (APT [Mackinlay 1986], SAGE [Roth and Mattis 1991], and Boz [Casner 1991]) were designed for static information, and for dynamic information may produce animations that are hard to understand. 2) Animation behavior. The tools that handle dynamic data (VisAD [Hibbard et al. 1992], and UWPI [Henry et al. 1990]) support only a limited range of animation behaviors. Namely, only the information from the current state of the computation is displayed. 3) Types of program entities. No automatic presentation technique, except UWPI, distinguishes between the type of program entities and properties of those entities.

Pre-designed visualizations have the main advantage that they are simple to use and quick to employ. Additionally, such visualizations typically benefit from the insight of an expert into the problem for which the animation was created. Pre-defined visualizations tend to be created for common tasks that occur in a certain programming language, type of computation, or class of problems [Eisenstadt and Brayshaw 1988; Jin n. d.; Pearl and Lalanne 2000; Topol et al. 1998]. Such views might increase cognitive load when applied to solve more specific problems or tasks outside the intended scope of the visualization because they add to the user's problem the task of mentally relating the user's problem to the visualization's problem.

6 Conclusions

This paper presents an approach to the creation, refinement, and use of program visualizations that centers on improving cognitive economy. The results of our empirical studies suggest that such improvements in cognitive economy will result in increased user understanding of the computations portrayed. The approach supports cognitive economy by allowing the production of animations customized to the user's task, which maximizes the amount of information that can be offloaded from the working memory onto the visualization, and by applying solutions for reducing the mental effort allocated to managing and interacting with the visualization.

The main characteristic of this approach is the building and modification of animations via interactions with graphical and textual views of the observed computation, without the need to learn and manage indirect structures. The solutions employed for alleviating cognitive effort and increasing the effectiveness of program animations include a data-driven manipulation of visualizations complemented by an automatic presentation algorithm and interactive legends for conveying the visualization syntax and for manual adjustment of the appearance of animations. The user specifies and has access to the information in the visualization, which is transformed into animated graphics by an algorithm customized for presentation of running computations. The choices of the system can be overridden by the user via continuous interaction with legends.

7 Acknowledgments

This work is a summary of my thesis, which would have not been possible without the help of Eileen Kraemer, Catalin Roman, and Del Hart. Also many thanks to my dissertation committee: Cindy Grimm, Ron Loui, and John Stasko. This work was supported by the National Science Foundation.

References

BAECKER, R. M., AND SHERMAN, D., 1981. Sorting out sorting. 16mm color sound film. Shown at SIGGRAPH '81.

BERTIN, J. 1983. *Semiology of Graphics*. The University of Wisconsin Press.

BLACKWELL, A. F., AND GREEN, T. R. G. 1999. Investment of attention as an analytic approach to cognitive dimensions. In *Collected Papers of the 11th Annual Workshop of the Psychology of Programming Interest Group (PPIG-11)*, 24–35.

BROWN, M. H., AND NAJOR, M. A. Algorithm animation using interactive 3d graphics. In *Software Visualization*, J. Stasko, J. Domingue, M. Brown, and B. Price, Eds. The MIT Press, Cambridge, ch. 9.

BROWN, M. H. 1988. Exploring Algorithms using Balsa-II. *IEEE Computer 21*, 5, 14–36.

BYRNE, M., CATRAMBONE, R., AND STASKO, J. 1999. Evaluating animations as student aids in learning computer algorithms. *Computers & Education 33*, 4, 253–278.

CASNER, S. M. 1991. A task-analytic approach to the automated design of graphic presentations. *ACM Transactions on Graphics 10*, 2, 111–151.

CRESCENZI, P., DEMETRESCU, C., FINOCCHI, I., AND PETRESCHI, R. 2000. Reversible execution and visualization of programs with leonardo. *Journal of Visual Languages and Computing 11*, 2, 125–150.

DIJKSTRA, E. W., AND C.S.SCHOLTEN. 1980. Termination detection for diffusing computations. *Inf. Proc. Letters 11*, 1, 1–4.

EISENSTADT, M., AND BRAYSHAW, M. 1988. The transparent prolog machine (TPM): An execution model and graphical debugger for logic programming. *Journal of Logic Programming 5*, 4, 1–66.

GREEN, T. R. G., AND PETRE, M. 1996. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing 7*, 131–174.

HAMILTON-TAYLOR, A., AND KRAEMER, E. 2002. SKA: Supporting algorithm and data structure discussion. In *Proceedings of the Thirty-third SIGCSE Technical Symposium on Computer Science Education (SIGCSE-02)*, ACM Press, New York, J. Impagliazzo, Ed., vol. 34, 1 of *SIGCSE Bulletin*, 58–62.

HANSEN, S., SCHRIMPSHER, D., AND NARAYANAN, N. 1998. Learning algorithms by visualization: A novel approach using animation-embedded hypermedia. In *Proc. Third International Conference on The Learning Sciences, Atlanta, GA*.

HART, D., KRAEMER, E., AND ROMAN, G.-C. 1999. Consistency considerations in the interactive steering of computations. *International Journal of Parallel and Distributed Systems and Networks 2*, 3, 171–179.

HENRY, R. R., WHALEY, K. M., AND FORSTALL, B. 1990. The University of Washington illustrating compiler. *ACM SIGPLAN Notices 25*, 6 (June), 223–233.

HIBBARD, W., DYER, C. R., AND PAUL, B. 1992. Display of scientific data structures for algorithm visualization. In *Proc. IEEE Visualization*, 139–146.

HUNDHAUSEN, C. D., AND DOUGLAS, S. A. 2001. Low fidelity algorithm visualization. *Journal of Visual Languages and Computing*. Under review.

Jinsight - Visualisation tools for Java. http://www.research.ibm.com/jinsight/.

KEHOE, C., STASKO, J., AND TAYLOR, A. 2001. Rethinking the evaluation of algorithm animations as learning aids: An observational study. *International Journal of Human-Computer Studies 54*, 2, 265–284.

LAWRENCE, A., BADRE, A., AND STASKO, J. T. 1994. Empirically evaluating the use of animations to teach algorithms. In *Proceedings of the 1994 IEEE Symposium on Visual Languages, St. Louis, MO*, 48–54.

MACKINLAY, J. D. 1986. Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics 5*, 2, 110–141.

MUKHERJEA, S., AND STASKO, J. T. 1994. Toward visual debugging: Integrating algorithm animation capabilities within a source level debugger. *ACM Transactions on Computer-Human Interaction 1*, 3 (Sept.), 215–244.

MYERS, B. A. 1990. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing 1*, 1 (Mar.), 97–123.

PEARL, P., AND LALANNE, D. 2000. Interactive problem solving via algorithm visualization. In *Proceedings of IEEE Information Visualization*, 145–153.

PRICE, B. A., BAECKER, R. M., AND SMALL, I. S. 1993. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing 4*, 3 (Sept.), 211–266.

REISS, S. P. 2001. Bee/hive: A software visualization back end. In *Workshop on Software Visualization, International Conference on Software Engineering ICSE 2001*.

ROESSLING, G., AND FREISLEBEN, B. 2000. The animal algorithm animation tool. In *ACM 5th Annual Conference on Innovation and T echnology in Computer Science Education (ITiCSE 2000)*, ACMPress, New York, 37–40.

ROMAN, G., AND COX, K. 1993. A Taxonomy of Program Visualization Systems. *IEEE Computer 26*, 12, 11–24.

ROMAN, G.-C., COX, K. C., WILCOX, D., AND PLUN, J. Y. 1992. Pavane: a system for declarative visualization of concurrent computations. *Journal of Visual Languages and Computing 3*, 2, 161–193.

ROTH, S. F., AND MATTIS, J. 1991. Automating the presentation of information. 90–97.

SCAIFE, M., AND ROGERS, Y. 1996. External cognition: How do graphical representations work? *International Journal of Human-Computer Studies 45*, 185–213.

STASKO, J. T., AND KRAEMER, E. 1993. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing 18*, 2, 258–264.

STASKO, J., BADRE, A., AND LEWIS, C. 1993. Do algorithm animations assist learning? an empirical study and analysis. In *Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems*, Understanding Programming, 61–66.

STASKO, J. T. 1990. The Path-Transition Paradigm: A practical methodology for adding animation to program interfaces. *Journal of Visual Languages and Computing 1*, 3, 213–236.

STASKO, J. 1998. Smooth continuous animation for portraying algorithms and processes. In *Software Visualization*, J. Stasko, J. Domingue, M. Brown, and B. Price, Eds. The MIT Press, Cambridge, ch. 8, 103–118.

TOPOL, B., STASKO, J. T., AND SUNDERAM, V. S. 1998. Pvanim: a tool for visualization in network computing environments. *Concurrency - Practice and Experience 10*, 14, 1197–1222.

TUDOREANU, M. E., WU, R., HAMILTON-TAYLOR, A., AND KRAEMER, E. 2002. Empirical evidence that algorithm animation promotes understanding of distributed algorithms. In *IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)*, 236–244.

TUDOREANU, M. E. 2002. *Economy of Interaction in Program Visualization: Designing Effective Visualization Tools for Reducing User's Cognitive Effort*. PhD thesis, Washington University in St. Louis. Also appears as technical report WUCS-02-30.

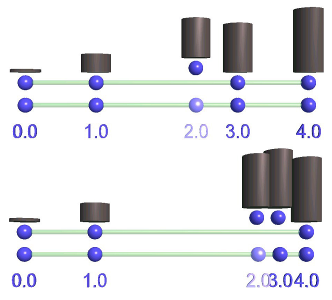


Figure 14: Snapshots of an interaction with the legend of Figure 13 by moving data tick 2.0 to the right.

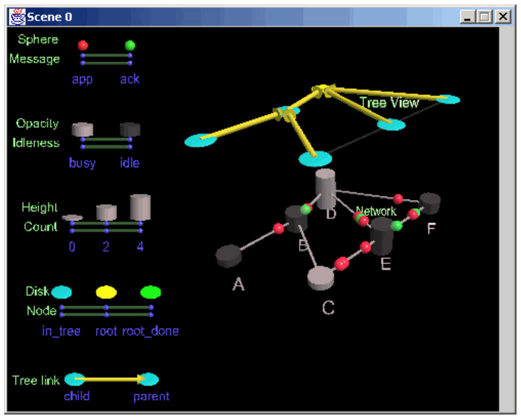
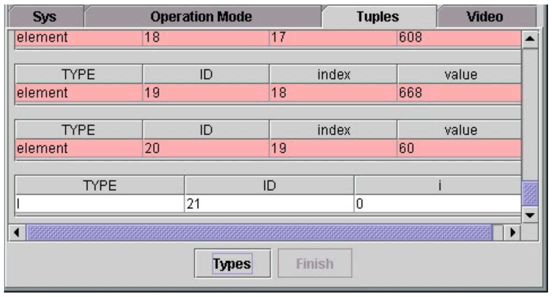


Figure 4: A snapshot of the termination detection visualization in the second study. The tree that spans the distributed network is presented above the network topology. A legend is displayed on the left.



element	18	17	608

TYPE	ID	index	value
element	19	18	668

TYPE	ID	index	value
element	20	19	60

TYPE	ID	i
I	21	0

Figure 5: The state of the computation is shown as a set of tuples.

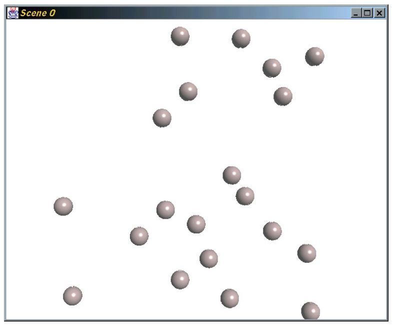
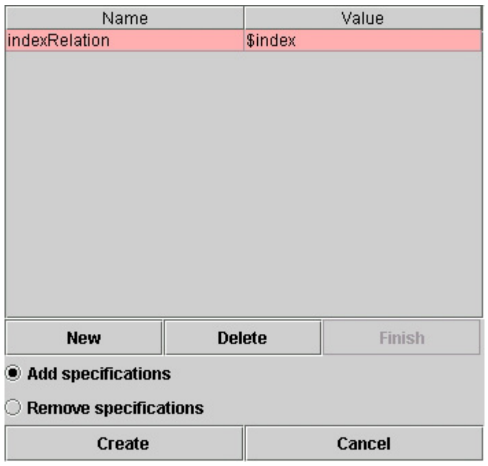


Figure 6: The elements of an array are depicted in a random layout. Each element is conveyed through a separate graphical object. No relations are shown among elements.



Name	Value
indexRelation	$index

(●) Add specifications

( ) Remove specifications

Figure 7: A new relation named 'indexRelation' is entered by the user.

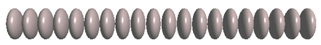


Figure 8: The elements of an array are shown in the order in which they appear in the array.

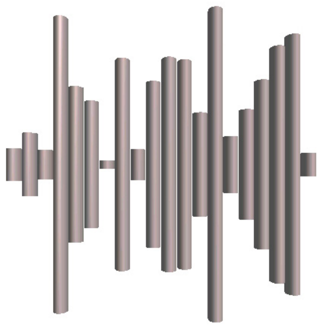


Figure 9: The elements of an array are depicted as cylinders. Two relations are shown: the order in which elements are stored in the array by x-position, and the value of each element by height.