Washington University in St. Louis

Washington University Open Scholarship

All Computer Science and Engineering Research

Computer Science and Engineering

Report Number: WUCS-93-22

1993-01-01

A Taxonomy of Program Visualization Systems

Gruia-Catalin Roman and Kenneth C. Cox

Program visualization may be viewed as a mapping from programs to graphical representations. This simple idea provides a formal framework for a new taxonomy of program visualization systems. The taxonomy is compared briefly against previous attempts to organize the program visualization field. The taxonomic principles and their motivation are explained in detail with reference to a number of existing systems, especially Balsa, Tango, and Pavane.

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation

Roman, Gruia-Catalin and Cox, Kenneth C., "A Taxonomy of Program Visualization Systems" Report Number: WUCS-93-22 (1993). *All Computer Science and Engineering Research.* https://openscholarship.wustl.edu/cse_research/310

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

School of Engineering & Applied Science

A Taxonomy of Program Visualization Systems

Gruia-Catalin Roman
Kenneth C. Cox

WUCS-93-22

February 1993

To appear in December, 1993
IEEE *Computer*

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

Printed June 30, 1993

Abstract

Program visualization may be viewed as a mapping from programs to graphical representations. This simple idea provides a formal framework for a new taxonomy of program visualization systems. The taxonomy is compared briefly against previous attempts to organize the program visualization field. The taxonomic principles and their motivation are explained in detail with reference to a number of existing systems, especially Balsa, Tango, and Pavane.

Keywords: software visualization, program visualization, algorithm animation.

Correspondence: All communications regarding this paper should be addressed to

Dr. Catalin Roman
Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

office: (314) 935-6190
secretary: (314) 935-6160
fax: (314) 935-7302

roman@cs.wustl.edu

1. Introduction

Pictorial representations of program structure, flow of control, and data have always been part of the programmer's repertoire of tools and techniques. The main purpose of such representations is to simplify the explanation of a particular aspect of the program. Their effectiveness is derived from a properly-chosen level of abstraction and a clean visual presentation. With the advent of modern workstations equipped with powerful graphical capabilities, interest has shifted beyond simple static presentations to exploring the dynamic behavior of programs. Graphical representations of programs have made their way into the debugging, monitoring, and teaching arenas. Program visualization has become recognized as a scientific endeavor concerned with the generation and use of graphical representations of programs. Today, program visualization is attracting considerable attention among researchers and is raising the hope that research results will soon be put to meaningful practical use in areas as diverse as software design, performance monitoring, and software training.

However, a significant gap exists between program visualization research and its practical application to programming environments and design tools. This paper attempts to remedy this situation by presenting a taxonomy for program visualization (excluding related work in visual programming and scientific visualization). The authors hope that this taxonomy will increase awareness of the status of program visualization and prompt consideration of the applicability of program visualization to the activities of the larger computing community. We illustrate the taxonomy using three program visualization systems: Balsa [1] and its successor Zeus [2]; Tango [3]; and Pavane [4]. Additional examples of program visualization systems may be found in a two-volume collection of papers recently published by the IEEE Computer Society Press [5]. The three systems mentioned above, however, are representative of research trends in the area.

A formal definition of program visualization as a mapping from programs to graphical representations (Section 2) supplies an objective technical justification for the taxonomy proposed in this paper. The remaining sections follow the organization of this taxonomy and describe each of the classification axes in greater detail; illustrative examples are presented in sidebars. Section 3 considers what aspects of a program one might want to visualize. Section 4 organizes the types of mappings encountered in program visualization from the point of view of their power of abstraction. Section 5 focuses on methods used to specify or construct the mapping from programs to graphical representations. Section 6 explores the graphical and interactive capabilities of visualization systems. Section 7 considers various presentation techniques used in the development of visualizations. Except as noted, all color images were generated using Pavane.

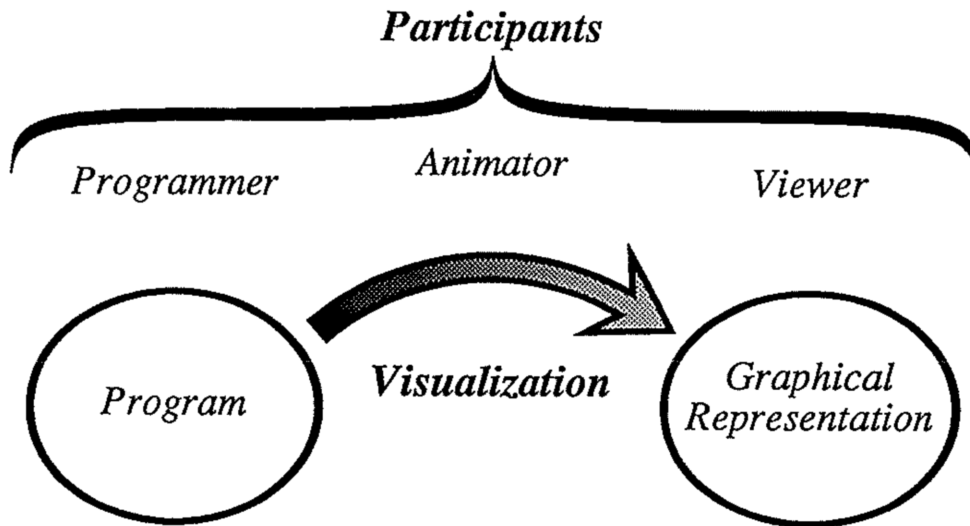


Figure 1. Visualization as a mapping from programs to graphical representations.

2. A Taxonomy of Program Visualization

The authors of a taxonomy are challenged to select classification principles that differentiate cleanly among recognized contributions to the field, offer meaningful insights into the workings of the various systems, and help identify likely prospects for future developments. Each published survey to date employs a different taxonomy. Shu [6] focuses on increasing degrees of sophistication exhibited by program visualization systems, from pretty-printing to complex algorithm animations. Myers [7] uses a classification along two axes: the *aspect* of the program that is illustrated (code, data, algorithm) and the *display style* (static or dynamic). Brown [8] proposes classifying algorithm animations along three axes: *content* (direct or synthetic representation of information about the program), *transformation* (discretely or smoothly changing images), and *persistence* (representations of the current state or of the entire execution history). Finally, Stasko and Patterson [9] advocate a categorization scheme involving four dimensions: *aspect*, *abstractness*, *animation*, and *automation* (a new criterion not discussed by others). Although each of these taxonomies has its own rationale and merits, we find them less than satisfactory because they are not based on a well-formulated model or theory of the field. Our search for an adequate (if not necessarily definitive) model of program visualization starts with an attempt to clarify what we mean by the term.

We see program visualization as a mapping from programs to graphical representations (Figure 1). In this light, the meaning of the term program, the available graphical vocabulary, the kinds of mappings being contemplated, and the means to construct these mappings become some of the defining features of a program visualization system. This definition suggests that the process of visualization is the result of an interplay among

three participants: the *programmer* who develops the original program; the *animator* who defines and constructs the mapping; and the *viewer* who observes the graphical representation. While these are only stylized roles meant to help us organize and present the material, the specialized expertise required of each role may actually lead in practice to a natural division of labor among distinct individuals.

This particular perspective on visualization leads to a classification of program visualization systems centered around characterizations of the domain, the range, and the nature and limitations of the mapping they support. We identify below five axes along which we classify program visualization systems. The first and the fourth relate to the domain and the range of the mapping, respectively, while the second and third deal with the mapping alone. The fifth axis is concerned with the communication from the animator to the viewer.

(1) *Scope*—what aspect of the program is visualized? The domain of an individual visualization is a program. Formally, a program can be characterized by its code, by its data and control states, and by its execution behavior. Visualization systems often limit their scope to a subset of these program aspects. A similar criterion was suggested by Myers.

(2) *Abstraction*—what kind of information is conveyed by the visualization? This criterion relates both to Shu's classification with respect to degree of sophistication and to Brown's notion of content. The issue is the level of abstraction associated with the concepts presented in a graphical form by the particular system. Highlighted code, for instance, offers a very low-level representation of the control state, while a module interconnection diagram hides the details of the code and depicts the flow of control at a more abstract level.

(3) *Specification Method*—how is the visualization constructed? This issue is essential to understanding the power and flexibility of a particular system. Some systems provide "hard-wired" mappings, while others may allow for arbitrary definition of the mapping; some focus on mapping program states while others focus on events; some require modification of the code while others do not. The automation axis proposed by Stasko and Patterson is related to this criterion.

(4) *Interface*—what facilities are provided for the visual presentation of information? This category addresses the issues of the visual vocabulary that the system provides, as well as the interactions that the viewer is provided to assist in exploring and understanding the information that is displayed. The focus is on what the viewer sees and how the viewer may control what the system shows.

(5) *Presentation*—how is the system used to convey information? This criterion is concerned with the effective use of the entire system in visual communication. It includes the mechanisms and heuristics that the animator uses in constructing visualizations. The main issue here is cognition rather than

mere visual perception, emphasizing the means by which the animator helps the viewer to gain an understanding of the program being visualized.

Our suggested taxonomy is quite compatible with those proposed by other researchers—indeed, their empirically-established criteria fit very naturally in our classification which, at least at the top level, is derived directly from a formal model of program visualization. We consider this to be evidence that the model captures certain properties intrinsic to this field. In the next five sections we further define each of these classification criteria.

3. Scope

The purpose of program visualization is to extract information about some aspect of a program and present it in a graphical form. Precisely which aspects of the program are examined serves as a major defining characteristic of visualization systems. Formally, the state of a program is characterized by its *code* (e.g., statements), *data state* (e.g., value assignment to variables), and *control state* (e.g., program counter). Dynamic properties of the program are captured by its execution history or *behavior*, which may be defined as a sequence of states with each pair of consecutive states representing an atomic event such as the execution of a statement. The next several paragraphs further examine each of these four aspects of the scope.

Code. Pretty-printers represent the most primitive form of code-oriented program visualization. They are concerned with increasing code readability by transforming syntactic patterns into page layouts. Programs to produce graphical presentation of code have also been available for years—the earliest systems produced flowcharts. Initially these programs were rather limited, producing statement-level diagrams of the program's structure. Following a trend toward greater abstractive power, modern systems permit a single program to be viewed in several different ways, from tree structures of the expressions within a statement to block diagrams of the interconnections between program modules. CASE tools make the most extensive use of code visualization, but it is also used to good effect in program visualization systems. For example, Balsa allows the animator to display the program code and highlight the current statement.

Data state. Early data-visualization systems were hampered by the relatively slow processors available, necessitating somewhat oblique methods. Some early algorithm animations (e.g., *Sorting Out Sorting* [10]) were recorded on film frame-by-frame, with each frame representing the results of a single state snapshot. The visualizations in these films use a relatively low level of abstraction, typically extracting only scalar variables such as the values held in an array. Increasing processor speeds permitted correspondingly more powerful monitoring and abstraction techniques. These have been exploited in debuggers designed to examine complex data structures and in database systems that permit graphical representation of both contents and queries.

Control state. In a sense, the program counter of the underlying processor represents the control state of a sequential program, but few systems use such a low level of abstraction. Instead, higher-level constructs such as statements and procedures are presented, as well as information about program flow and procedure invocation sequences. A typical approach is to relate control information to the overall structure of the program, for example by displaying the code structure as a module-interconnection diagram and highlighting the module currently having control. As noted above, Balsa provides such a mechanism for viewing the control state.

Behavior. A program may be viewed as performing a series of atomic transformations of its state. Observation of these transformations, formally known as events, permits development of an understanding of how the program works. The granularity of events can vary from primitive machine instructions to large operations performed by entire blocks of statements. The events of interest may be changes in the values of specific variables, entry and exit from procedures, or communication activities. Events are obtained by monitoring a computation. The results of the monitoring may be displayed "live", or an execution trace may be saved for later visualization, with the possible advantage of being able to display it forward and backward at different speeds.

4. Abstraction

In this section we consider the information that is conveyed via the graphical representation and discriminate among systems based on the level of abstraction they support. Abstract representations of the program are needed to control complexity during debugging and monitoring and to facilitate program understanding in a pedagogical setting. Display size limitations are another factor requiring the use of compact abstract representations. Our taxonomy distinguishes among three levels of abstraction. We consider each in turn from the least to the most abstract representation of the program. The boundaries between the various levels are imprecise, and in practice a visualization system is likely to support multiple levels of abstraction, separately and in combinations.

Direct representation. The most primitive graphical representations are obtained by mapping some aspect of the program directly to a picture. Because only very limited abstraction mechanisms are employed, it is often the case that the original information can be easily reconstructed from the graphical representation. Gauges set to indicate the values of variables, two-dimensional representations of binary trees, and color encodings of values stored in an array are some examples of direct representations. Other forms are encountered primarily in CASE and debugging systems: flowcharting and similar graphical representations of code structure, monitoring of control flow through display of the current statement, and tracking of procedure invocations by presenting the call stack (possibly as overlapping windows, one per invocation). Direct visualizations are appealing because they are easy to relate to the program and can often be constructed mechanically, without knowledge about the programmer's intent.

Unfortunately, intent must often be conveyed in a visualization, particularly when sophisticated algorithms are involved.

Structural representation. More abstract representations may be obtained by focusing attention on particular aspects of the program or its execution. This can be done by concealing or encapsulating information that is not of interest and using a direct representation of the remaining information. Alternatively, the important portions of the information can be emphasized in the final image by techniques such as highlighting or positioning. Many examples of this type of abstraction can be cited. Two- or three-dimensional diagrams and graphs are typically used to depict program structures (e.g., structure charts), network connectivity, and data access capabilities; in each case a complex object composed of a number of subcomponents is treated as a single simple object, with its internal structure hidden. Histograms may encode the relative frequency of message occurrences by type, concealing other message details. In an operating system, proportionally-sized colored blocks may indicate memory allocation and usage without attempting to represent the contents of memory. In all these cases, the information presented to the viewer is present in the program, although possibly obscured by details. The representation simply conveys the information in a more economical way by suppressing aspects not relevant to the viewer.

Synthesized representation. Representations in this category are distinct from structural representations in that the information of interest is not directly represented in the program but can be derived from the program data. This shift of perspective often occurs when the data representations selected by the programmer come into conflict with the needs of the animator. The animator may prefer to emphasize other aspects of the algorithm that, although logically present in the program, have no explicit representation. The animator must then construct and maintain a representation that is more convenient for purposes of visualization. Many systems provide for forms of synthesized representations. In Balsa, for instance, the animator can construct synthesized representations by defining data structures that are shared by the graphical procedures.

5. Specification Method

The taxonomic criterion of *specification method* encompasses the means whereby the animator specifies which aspects of a program are to be extracted and how they are to be displayed. The ease and efficiency with which both these operations can be done is a major factor in the utility of a visualization system. The exploratory activities involved in testing and debugging demand high degrees of flexibility in defining and redefining the mapping, while execution monitoring requires minimal intrusion in the computation process. In this section we discuss the ways in which different program visualization systems approach the task of specifying visualizations.

Predefinition. Application-specific visualizations often employ a fixed or highly constrained mapping. The animator has little or no control over what is visualized and in what way the information is presented. Most

CASE systems opt for this approach because their intent is to support and enforce a particular methodology and graphical notation. Although the expressiveness of these systems is often limited, they have the advantage of speed; the constraints permit the use of specialized visualization algorithms that have been optimized for performance. The extraction of program data can also be automated, for example by incorporating the necessary monitoring code at compile-time.

Annotation. This approach was pioneered by Balsa and gained widespread acceptance in the algorithm animation field. The animator writes procedures which construct and modify images, then augments the program with calls to these procedures. The placement of these calls (annotations) corresponds to the occurrence within the code of events deemed significant to the operation of the algorithm. Information about the program state is passed via the parameters of the invoked animation procedures. In this manner computational events are mapped to arbitrary sequences of visual events, i.e., changes in the graphical representation. Tango uses a similar approach but emphasizes the production of animations in which object locations and other attributes change smoothly following specific trajectories. Tango visualizations are treated as a mapping from program events to animation actions. The occurrence of an event of interest may trigger, for instance, coordinated movements of one or more graphical objects in the image.

The annotative approach has several advantages, chief of which is the animator's ability to define events at suitable levels. For example, a sorting algorithm might be animated by displaying each comparison and swap; however, if the same sorting algorithm were used as a subroutine in another algorithm, the animator might choose to present only the final result. The ability to write application-specific animation procedures to handle each event is both an advantage and a disadvantage, in that it permits versatility but requires additional work; this can be somewhat mitigated through the use of libraries of routines. The most significant shortcoming of the method is the need to access and modify the code of the program.

Declaration. In the declarative approach the animator specifies a mapping between the program state and the final image, and changes in the state are immediately reflected in the image. Simple versions of the declarative approach allow the animator to directly map the values of variables to attributes of graphical objects in the final image. This approach (in which declaration is limited to direct mappings) might be called *associative* , since it requires that the animator use only one-to-one associations between variables and values. Its advantages and disadvantages are similar to those of specification by predefinition.

The more general declarative approach permits mappings with any level of complexity, allowing the capture and visual expression of arbitrary predicates over the state. Pavane is the first system to make extensive use of the declarative approach to visualization. The decoupling of the visualization from events in the underlying computation has a number of advantages. In particular, computations in which events are difficult to define or to isolate can be

visualized more readily using declaration than with annotation, and the specifications are often quite compact. However, declaration does have disadvantages. One problem typically arises when several primitive state changes should be considered as a single logical change and the image should not reflect the intermediate results. This can be avoided in several ways, such as the approach used by Aladdin [11] in which the relationship between program variables and graphical objects is defined declaratively, while code annotations indicate where the image should be updated. The Animus system [12] uses the declarative approach in two ways. Each object can have a graphical representation that is automatically updated in response to changes in the object. More significantly, the animator can declaratively specify *constraints* on the relations between objects, and the system will ensure that these constraints are maintained (e.g., by moving the representation of an object).

Manipulation. Systems that use manipulation, also called *animation by demonstration*, specify visualizations through the use of examples. These systems attempt to capture the gestures used by the animator in manipulating objects present in an image and to tie these gestures to specific program events—in effect, defining a mapping between program and visual events. One such system [13] was illustrated on a sorting algorithm, where the exchanges of array elements were animated by defining a "swap gesture" that exchanged the positions of two rectangles. This gesture was then tied to the exchange event in the sorting algorithm by selecting the appropriate portion of the program code. Specification by manipulation suffers from the difficulty of specifying the exact relationship between the gesture and the program data or events. This can be accomplished for specific cases, but a general approach has yet to be defined. Most of the existing systems use a mixed approach, where motions are specified gesturally but the binding between events and some attribute of the motion is specified by annotation or declaration. Tango has experimented with such a gesture-capturing system.

6. Interface

Under this heading we discuss the range space of the mapping (see Figure 1), i.e., the graphical representation that is presented to the viewer. We restrict this topic to the concrete entities and operations that the system provides, in other words to the *syntactic* constructions that the animator can use—the *semantics* attached to these constructs is the subject of the next section. We can divide the interface criterion into the two subcategories of *graphical vocabulary* and *interaction*. The first refers to the objects which the animator can use to present information to the viewer, while the second is concerned with the tools that allow the viewer to modify the presentation.

6.1. Graphical Vocabulary

The graphical vocabulary of a system specifies the kinds of graphical objects that can be used by the animator and the ways in which these objects can be combined to construct the images being presented to the viewer. We list some of the common components of the graphical vocabulary below.

Simple objects. The simplest type of object is an abstract geometric entity such as a point, line, rectangle, circle, or sphere. We can also include icons and alphanumeric text in this category. Virtually all program visualization systems provide a large vocabulary of geometric objects. In practice, these are treated as *classes* of objects which are instantiated by providing the values of the object's attributes (e.g., the endpoints and color of a line, or the center, radius, and color of a sphere).

Composite objects. Many systems provide means whereby new types of objects can be created from collections of simpler objects. These new, user-defined object types can be used exactly as if they were system-supplied types, with the attributes of the component objects inherited or derived from those of the complex object. Tango provides such a type-definition mechanism. Many visualization systems provide a variety of pre-defined composite objects; in this context, the objects are often called *widgets*.

Visual events. We define a visual event as the creation, destruction, or modification of a graphical object. The resulting transformation can be instantaneous, but most systems provide mechanisms whereby the change can be animated. For example, if the size of an object changes from two units to ten units, this may be rendered as a sequence of images in which the object is depicted with a size of two, three, four, and so on up to ten units; if the images are displayed rapidly, the result appears to be a smoothly-growing object. Tango has very sophisticated facilities for the specification of such smooth animations, using a mechanism whereby a *path* (a definition of a motion in some space, such as object coordinates, color, or visibility) is applied to an object. Pavane also provides mechanisms for generating visual events by assigning time-variant values to the attributes of objects.

Worlds. Graphical objects may be combined by establishing geometric relationships between their positions in a "world" or space of two or more dimensions. Most program visualization systems use two dimensions, although Pavane uses three, and certain specialized systems for scientific visualization use spaces of more than three dimensions which are projected onto a three-dimensional space for display. The contents of the world are then "viewed" from some point to generate the image.

Two main approaches to the positioning of objects are used. Objects can be placed according to absolute coordinates, with each object independent of all the others. In a second approach, generally called *constraint-based*, object positions can be specified relative to other positions (this approach is often used in the construction of composite objects, with the positions of the component objects specified relative to those of the composite object). Balsa, Pavane and Tango all permit limited forms of constraint-based positioning, but the Animus system has

probably the most sophisticated capabilities. It should be noted that constraint-based systems are computationally expensive, since the determination of attribute values may involve the solution of large systems of equations. However, our expectation is that constraint-based systems will take a dominant position in the future as processor speeds increase.

Multiple worlds. An additional component of the graphical vocabulary is provided by the use of multiple worlds, each with a separate collection of graphical objects, and each presented in a separate window on the screen. These multiple worlds are often used to display several different visual representations of a computation. This ability to compare alternative visualizations is thought to lead to greater insight into the behavior of the programs being visualized and can act as a complexity-control mechanism by limiting the amount of information that must be shown in each window.

6.2. Interaction

The second major subdivision of the interface category deals with the mechanisms whereby the viewer of a visualization can interact with the presentation. Most traditional methods of presenting information, for example books, permit only a few limited forms of interaction such as selecting which pages to examine. The reader cannot change the contents of the book or alter the formatting of the pages. In contrast, computer-aided presentations of information such as visualizations permit a wide variety of interactions. In addition to selecting portions of the information to examine, the viewer can often act to modify the information or the manner in which it is presented. We distinguish two basic types of interaction as described below.

Interaction through controls. A common method of handling interactions is through pre-defined controls which interface with the system in predictable ways. Such controls are visible in the Zeus, Tango, and Pavane screen images presented earlier. Typical interactions with the underlying computation include selecting input data or stepping through events; Balsa provides a particularly large collection of these. Display interactions include controlling the speed with which the images are displayed and defining the point from which the graphical world is viewed. Pavane, with its three-dimensional graphical world, has a number of controls for the latter function.

Interaction through the image. Another type of interaction is that where the viewer can affect either the underlying computation or the mapping by direct manipulation of the images. For example, the user might use a mouse to select one of the graphical objects in the image and move it to a new position. This interaction might simply be a re-positioning to improve the appearance of the image, in which case it should affect the mapping—the new position should be stored and used to generate subsequent images. However, it could also indicate that the user wants to modify the data represented by the object, in which case the underlying computation should be affected. The ambiguity of interactions with the image—do they affect the mapping or the underlying computation?—is

resolved by the animator, who defines the interpretation to be used. Interaction through the image is closely linked with iconic interfaces to operating systems, in which the storage and execution of files and programs is managed through small images. It also plays a major role in specification by manipulation.

7. Presentation

The term *presentation* refers to the semantics of a visualization, in other words the manner in which visualizations convey information. This criterion is largely concerned with the methods that the animator uses in developing a visualization or collection of visualizations with the goal of fostering understanding on the part of the viewer. We list some of the more important issues below. Some of these are general heuristics that can be adapted to any visualization system, while others may require some form of support from the system.

Interpretation of graphics. The elements of the graphical vocabulary have no pre-defined semantic content and can be arbitrarily arranged. As a result, most visualizations require some explanation in addition to the images. This additional information may be simply a contextual cue ("This is a sorting algorithm") or it may be much longer ("This area shows the search tree, and over here the elements of the array are shown, and notice how the array elements are highlighted as the branches of the search tree are extended, and..."). Visualizations that require little or no explanation are clearly preferable, but what factors make certain interpretations of the graphics "obvious"?

The answer would seem to be that people do have some standard interpretations of graphical representations. The chief source of these interpretations appears to be prior experiences, particularly with statistics and cartography which have developed a wide vocabulary of pictorial representations of data. In addition to these general expectations of how information should be presented, program visualization incorporates a number of representations of control and data structures that are (quite literally) drawn from the pictures used in programming texts. Thus, to effectively communicate information the animator should be aware of these conventional styles of presentation. A complete survey of these techniques is beyond the scope of this paper, but we can list a few important aspects.

Graphical objects generally have a flexible interpretation, which means that in many visualizations the objects can be arbitrarily changed (circles replaced by squares, for example) without interfering with the viewer's ability to understand the images. The attributes of graphical objects—type, color, size, and so forth—can be used to represent information in a variety of ways. Information can also be conveyed by the spatial relationships among objects. Structural properties of programs or data can be captured by analogous geometric structures, e.g., a two-dimensional array may be represented by a tiled rectangle, or processes that execute in a round-robin fashion may be arranged in a ring. More abstract properties may be visualized by enforcing particular geometric alignments or coloring rules.

Analytical presentation. One heuristic for the design of visualizations is to de-emphasize the operational mechanics of program execution and focus on issues that are important in analytical thinking about the program, such as its formal correctness properties. Visualizations can be constructed to show these properties, thereby conveying them to the user. Such visualizations are also useful during program development and debugging, since violations of the properties can be visually detected and the program fault thereby isolated. Analytical presentation is the theme of much of our work with Pavane. This is because we attempted to provide a framework for visualization of concurrent computations—a domain in which formal reasoning about program correctness is far more crucial to understanding an algorithm than in the sequential arena.

Explanatory presentation. We use this term to refer to the use of visual events to convey information. The animator typically attempts to improve the presentation by incorporating visual events that have no direct counterparts in the computation being depicted. The goal may be to improve the aesthetic quality of the presentation, to communicate the implications of a particular computational event, or to focus the viewer's attention on key elements. Tango provides particularly sophisticated facilities for the construction of such animations. In our work with Pavane, we have identified several means whereby the analytical and explanatory approaches can be combined, thus using visual events to convey formal properties of programs.

Orchestration. A carefully-selected collection of visualizations is often much more effective at engendering understanding than a single visualization. Balsa has recognized the importance of such orchestrated presentations (called *scripts* in the Balsa system) and provided mechanisms for composing, editing, and replaying them. Balsa has an extensive library of such scripts for problems such as sorting and bin-packing.

Orchestrations can be composed at several levels. A single problem instance may be illustrated by presenting a variety of algorithms that solve the problem, allowing a comparison of the operation of the programs. This may provide insight into both the individual algorithms and the overall problem. This technique is particularly effective if all the algorithms operate on the same data and are presented simultaneously, since the relative performance characteristics can then be perceived.

A single algorithm can also be orchestrated by presenting multiple visualizations, e.g., several simultaneous Balsa views or Pavane rule sets. This increases the likelihood that the viewer will understand at least one of the visualizations, and can then focus on that view of the program. Again, simultaneous presentation facilitates comparison and can result in additional insight. A related means of orchestration is the presentation of multiple instances of the underlying problem. Often, it is advisable to launch a presentation with a relatively small problem instance and gradually introduce larger ones as the viewer begins to understand the meaning associated with the visual patterns on the screen. Biasing the selection of data is also commonly used—for example, guaranteeing

that all elements of an array are distinct. Pathological problem instances, such as providing already-sorted or inversely-sorted data to a sorting algorithm, are often particularly useful.

8. Summary

In this paper we introduce a new taxonomy for program visualization systems. We have illustrated the taxonomy using examples from three specific systems (Balsa, Tango, and Pavane) with occasional references to several others. The classification principles have been derived from a complete model of program visualization. For formally well-understood areas, the classification is simple and precise. Differentiations by scope, level of abstraction, specification method, and interface fall in this category, and program visualization systems can be readily categorized (as shown in Table 1), because these areas have well-defined formal frameworks. The scope criterion, for instance, has a very clean characterization because it can exploit the conceptualizations of programs that have been developed by programming language theorists.

The remaining classification axis, presentation, is concerned with visual communication. This area currently has very little formal foundation, and this is reflected in the rather ad-hoc categorization of the area in our taxonomy. This reinforces our conviction that any useful taxonomy must rely on a simple objective formal framework, and identifies the development of a more formal treatment of the visual presentation of information as a plausible area for research. The informal categorization presented here represents a small step in this direction.

Acknowledgments: This work was supported in part by the National Science Foundation under the Grant CCR-9015677. The government has certain rights in this material.

References

[1] Brown, M. H., "Exploring Algorithms using Balsa-II," *IEEE Computer*, vol. 21, no. 5, pp. 14-36, 1988.

[2] Brown, M. H., "Zeus: A System for Algorithm Animation and Multi-View Editing," *1991 IEEE Workshop on Visual Languages*, IEEE Computer Society Press, Kobe, Japan, pp. 4-9, 1991.

[3] Stasko, J., "TANGO: A Framework and System for Algorithm Animation," *Computer* 23, Nr. 9, pp. 27-39, 1990.

[4] Roman, G.-C., Cox, K. C., Wilcox, C. D., and Plun, J. Y., "Pavane: a System for Declarative Visualization of Concurrent Computations," *Journal of Visual Languages and Computing*, vol. 3, 1992.

[5] Glinert, E. P. (editor), *Visual Programming Environments*, IEEE Computer Society Press Tutorial, 1990.

[6] Shu, N. C., *Visual Programming*, Van Nostrand Reinhold Company, New York, NY, 1988.

[7] Myers, B. A., "Taxonomies of visual programming and program visualization," *Journal of Visual Languages and Computing*, vol. 1, no. 1, pp. 97-123, 1990.

[8] Brown, M., "Perspectives on Algorithm Animation," *CHI'88 Human Factors in Computing Systems*, Washington, DC, USA, pp. 33-38, 1988.

[9] Stasko, T. J., and Patterson, C., "Understanding and Characterizing Software Visualization Systems," *Proceedings of the IEEE Workshop on Visual Languages*, September 1992, pp. 3-10.

[10] Baecker, R. M., *Sorting Out Sorting* (film), Dynamic Graphics Project, University of Toronto, Toronto, Canada, 1981.

[11] Helttula, E., Hyrskykari, A., and Raiha, K.-J., "Graphical Specification of Algorithm Animations with ALADDIN," in *Proceedings of the 22nd Annual Conference on Systems Sciences*, pp. 892-901, 1988.

[12] Duisberg, R. A., "Animation Using Temporal Constraints," *Human Computer Interaction*, vol. 3, pp. 257-3071, 1987.

[13] Duisberg, R. A., "Visual Programming of Program Visualizations: A Gestural Interface for Animating Algorithms," in *Proceedings 1987 Workshop on Visual Languages*, IEEE Computer Society, Linkoping, Sweden, pp. 55-65, 1987.

		Balsa	Tango	Pavane
Scope	Code	✓		
	Data state	✓	✓	✓
	Control state	✓	✓	✓
	Behavior	✓	✓	✓
Abstraction	Direct representation	✓	✓	✓
	Structural representation	✓	✓	✓
	Synthesized representation	✓	✓	✓
Specification method	Predefinition			
	Annotation	✓	✓	
	Declaration			✓
	Manipulation		(a)	
Interface	Simple objects	✓	✓	✓
	Composite objects		✓	
	Visual events	✓	✓	✓
	World (dimensionality)	2	2	3
	Multiple worlds	✓	(b)	✓
	Interaction through controls	✓	✓	✓
	Interaction through image	(c)		
Presentation (d)	Analytical presentation			✓
	Explanatory presentation		✓	✓
	Orchestration facitilities	✓		

Notes:

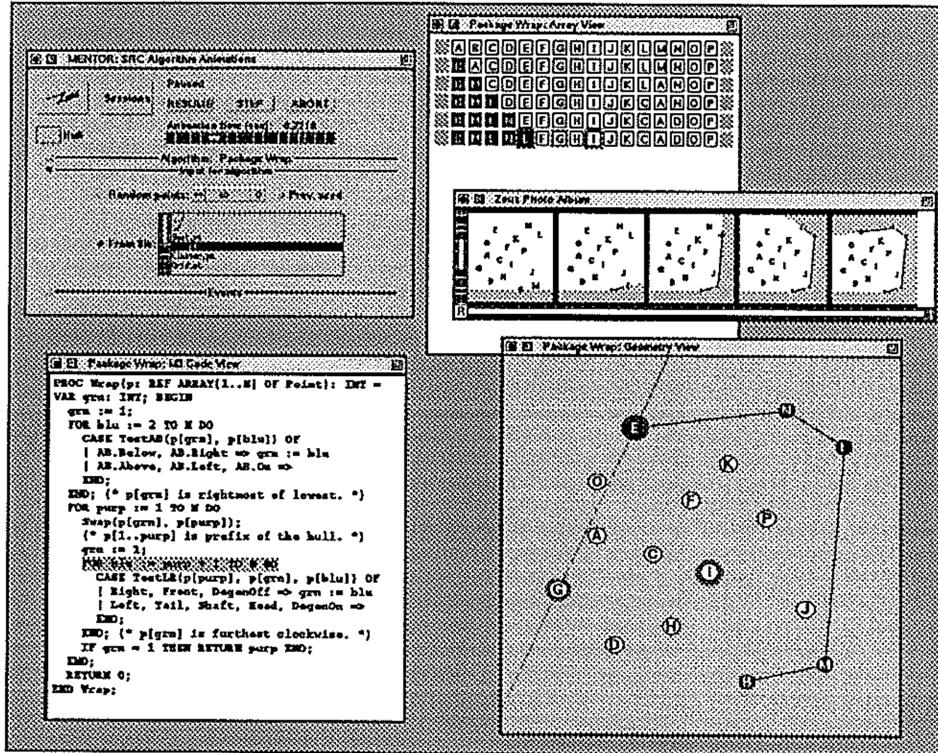
(a) The *Dance* program permits graphical specification of Tango algorithm animations. See J. Stasko, "Using Direct Manipulation to Build Algorithm Animations by Demonstration", *Proceedings CHI'91 Human Factors in Computing Systems* (Addison-Wesley, 1991), pp. 307-314.

(b) Although Tango has the ability to display several different visualizations of a single underlying computation simultaneously, the current distribution version of XTango does not.

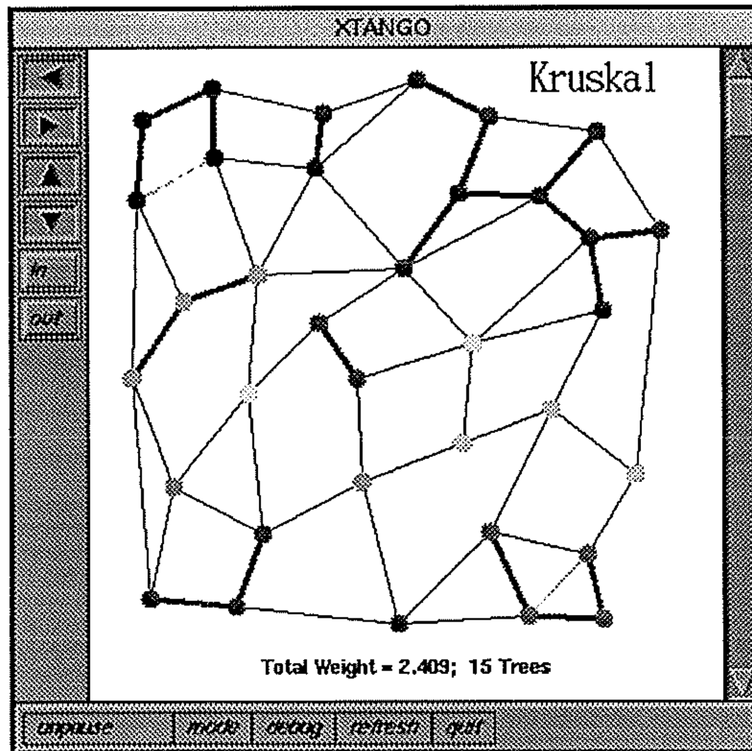
(c) This capability is available to a limited degree in Zeus.

(d) Most aspects of the presentation criterion can be used with all visualization systems (analytical presentation, for example, can be used to construct visualizations in any system). This portion of the table thus indicates areas of particular emphasis or special system support. The "interpretation of graphics" aspect of presentation is a human interaction which is independent of the system, and is therefore omitted.

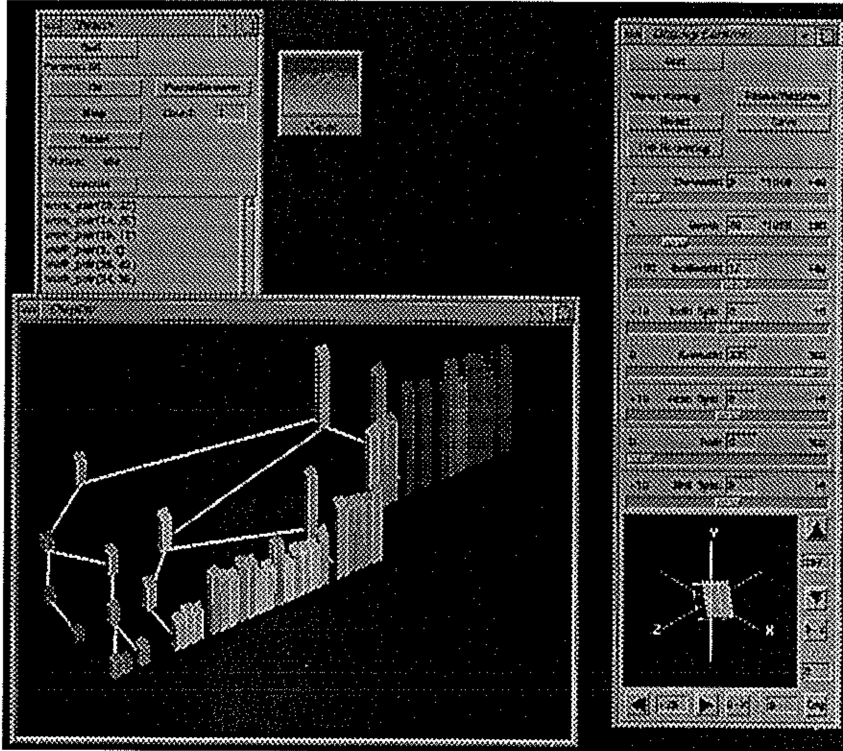
Table 1. Summary of the capabilities of Balsa/Zeus, Tango, and Pavane with respect to the taxonomic criteria of this paper.



(a)



(b)



(c)

Three visualization systems. The development of Balsa marked the emergence of the current generation of program visualization techniques and the first serious attempt to organize the task of constructing complex custom visualizations. Towards this aim, Balsa introduces a highly-modular system structure. Meaningful state changes (significant events) in the program are captured by adding procedure calls to the program code in a process called program annotation. Each event may be distributed to one or more visualization processes or *views* which are responsible for creating and updating the graphical displays associated with individual representations of the program and for managing the windows in which the images are painted on the screen. The graphical representations are two-dimensional and the programs being visualized are mostly sequential. Balsa was an important research prototype against which other efforts in the field are being compared, and was the first program visualization system to make its way into classroom use.

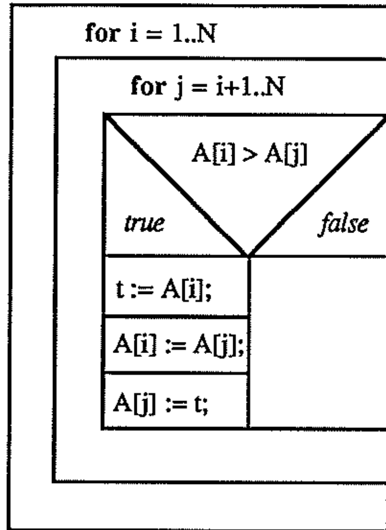
Image (a) (courtesy of M. Brown) shows a visualization generated using the Zeus system for algorithm animation by Marc H. Brown and John Hershberger. Zeus is a direct descendent of Balsa which includes many of Balsa's basic capabilities as well as a number of innovative features. This image shows a convex hull animation implemented by Lyle Ramshaw and James B. Saxe. Three views of the algorithm are depicted, a code view in the lower left and two data views in the upper and lower right. The control panel at the upper left and the window in the center right are provided by Zeus; the latter depicts a miniature image of the contents of several views.

Subsequent attempts to simplify program visualization focused on minimizing the effort required to construct the display code. Tango exploits the encapsulation features of object-oriented programming. The system supplies a number of animation data types (locations, images, paths, and transitions) which a programmer may use to construct graphical objects and to define display events. Simple images are composed from graphical elements such as lines and rectangles. Paths and transitions are used to define series of discrete changes in the values of image attributes such as smooth movement or color changes. In addition, Tango provides facilities for identifying program data as deriving from image attributes and for mapping program events (defined through program annotation) to display events. The latter is accomplished with the help of a finite-state transducer. Thus, the same program event may be mapped to different display events at different times depending upon the state of the transducer.

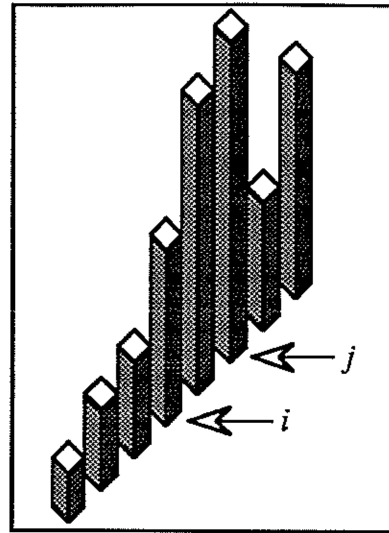
Image (b) (courtesy of J. Stasko) shows a visualization produced with XTANGO, a version of Tango which uses the X Window System®. The animation is of Kruskal's minimum-spanning-tree algorithm. The buttons to the left control the image viewpoint through scroll and zoom operations, while the scroll bar on the right controls the animation speed.

The techniques employed in Balsa and Tango cannot be easily applied to the domain of concurrent programming. In this domain, code annotation becomes impractical because a particular event may not necessarily be associated with a specific location in the program code. In addition, the large number of events and possible co-occurrences of events makes it difficult to reason about a program in terms of sequences of events. It is this change in emphasis from sequential to concurrent programming that forced the authors to consider a radical paradigm shift in the design of Pavane. In Pavane, visualizations are constructed by mapping program states to images; the transition between two states can be mapped into a corresponding transition between two images. An image is described as a collection of simple objects whose existence and visual attributes are determined by the current state of the program. The mapping is specified by sets of rules which are applied after each state change to generate a new image. Animation is achieved by allowing the objects in an image to have time-dependent attributes. The rules are similar in form to those employed by production systems. This combination of features allows the construction of abstract representations of concurrent programs using a relatively small number of rules, without having to write display code or annotate the program.

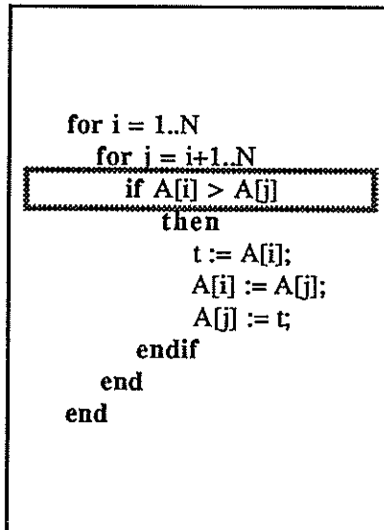
Image (c) reproduces a Pavane visualization of a quicksort algorithm. The window at the lower left displays the generated image. The window at upper left controls the underlying computation (the quicksort), while the window to the right handles the image display. The small icon at the upper left is a window, currently closed, which allows inspection of the mapping computation which applies the Pavane rules to generate the image.



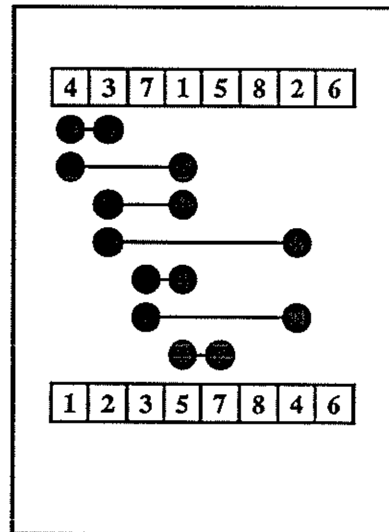
Code Visualization



Data State Visualization



Control State Visualization



Behavior Visualization

Scope. The scope component of the taxonomy is illustrated using a simple selection-sort computation. The visualization at the upper left depicts the program *code* in a modified Nassi-Shneiderman diagram. This static representation is useful for showing the structure of the code, but does not convey the computational activities.

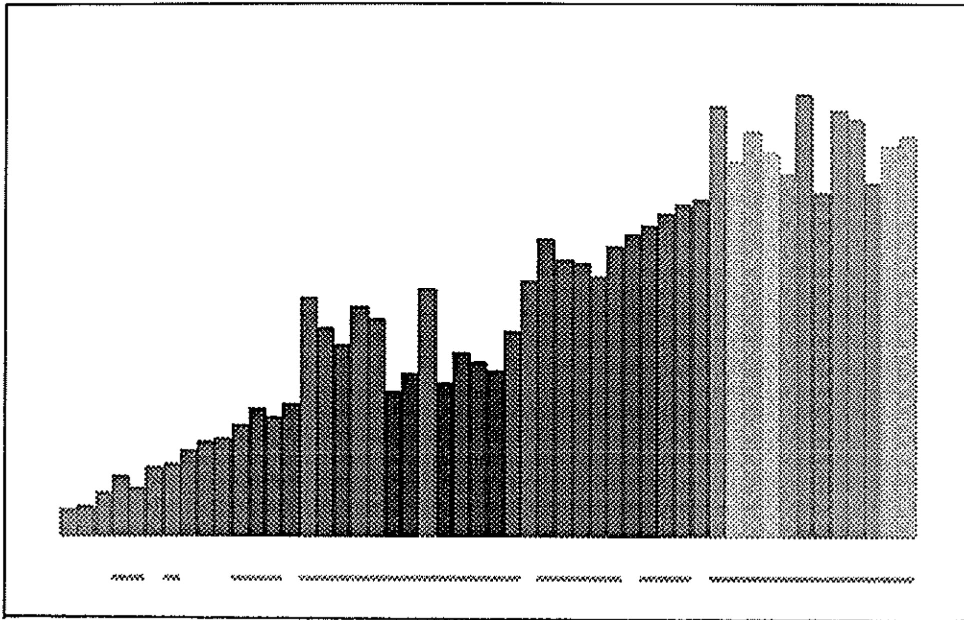
The visualization at the lower left shows the *control state* by presenting the code and highlighting the current statement—the conditional "**if** A[i] > A[j]". The viewer of this visualization gets a strong feeling for the

control activities of the computation, particularly the operation of the nested loops; however, the sorting of the array is not apparent.

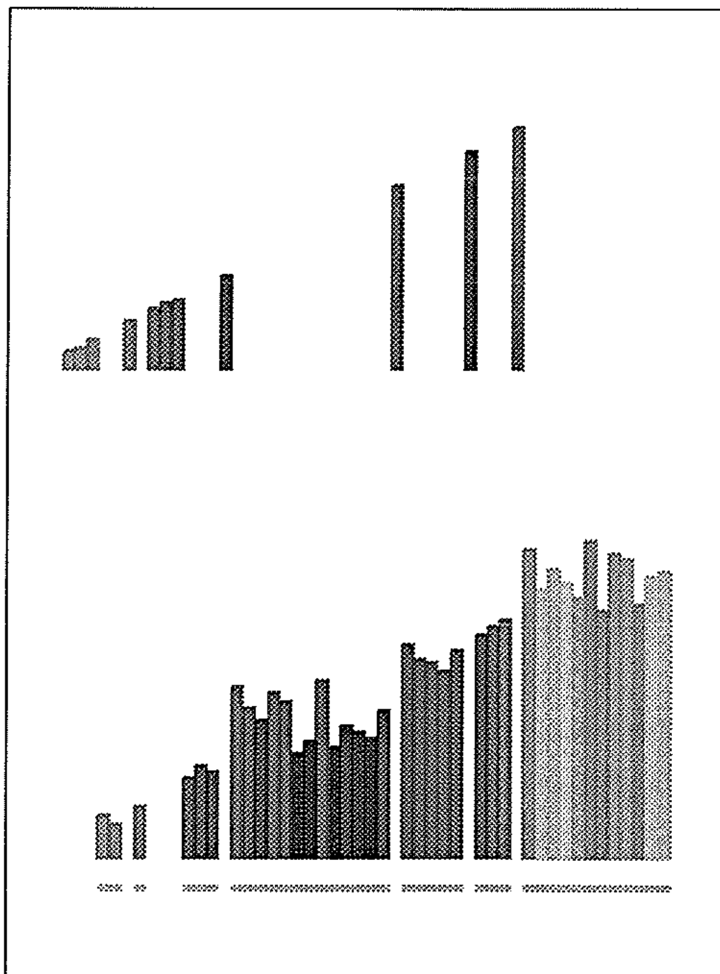
A third possible scope is the *data state* of the computation, as shown in the upper right. The array *A* is depicted as a row of boxes, with the height of the boxes proportional to the value stored in the array element. The variables *i* and *j*, which are used as indices into the array, are shown with arrows pointing at the corresponding array elements. This visualization shows that the algorithm sorts the array.

The final aspect of the scope distinguished by our taxonomy is the *behavior* of the computation, defined as the sequence of events or actions that occur. The primary event in this sorting algorithm is the exchange of two elements. The behavior visualization in the lower right image shows the sequence of changes that have occurred, as well as the initial and current contents of the array. Viewing the entire sequence of exchanges in this manner may provide insight into the algorithm that is not obtained by viewing the exchanges one at a time.

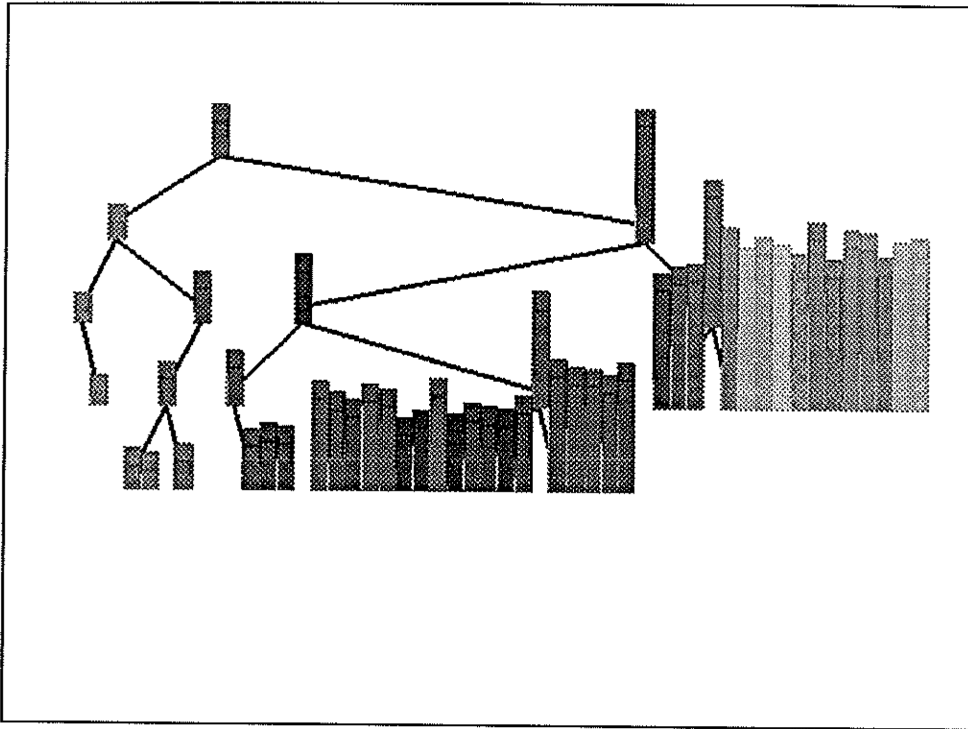
These images also illustrate the fact that most visualizations depict several different aspects of the computation. For example, this control state visualization also shows the program code, while the data state visualization captures some control state information through its depiction of the loop variables *i* and *j*.



(a)



(b)



(c)

Abstraction. The quicksort algorithm can be used to illustrate the abstraction of data. The state of a quicksort computation consists of the array being sorted and the regions within the array which remain to be sorted. At each step, a region is selected and a "pivot element" within the region is chosen. The array elements in the region are then reordered into the elements less than the pivot, the pivot itself, and those greater than the pivot. This results in one correctly-placed element (the pivot) and two new regions which remain to be sorted (the elements less than the pivot, and those greater than the pivot).

A *direct* representation can be produced by mapping array elements to a row of rectangles, with the size and color of the rectangle determined by the value stored in the array, and mapping regions which remain to be sorted to horizontal lines which span the corresponding array elements. Such a visualization is shown in image (a). This image captures the entire state of the computation, but does not especially aid the viewer's understanding of the algorithm.

A *structural* representation is suggested by the program correctness property, "Elements not within a region to be sorted are correctly placed." These elements can be visually separated from the others, as shown in image (b). In this case, the completed work is positioned above the work remaining to be done. Array indices still determine the X-coordinates of the rectangles, and element values are represented by the length and color of the rectangles. This visualization conveys a better understanding of the operation of the algorithm, allowing observation of the fact that each "pivoting" correctly places an element.

Sidebar: Abstraction

An alternate, *synthesized* representation is suggested by the correctness property, "Each region to be sorted is eventually converted into a placed element and zero, one, or two new regions." This behavior can be viewed as defining a tree structure. Each node of the tree corresponds to a placed element or a region; when a region is processed, it produces a sub-tree whose root is the newly-placed pivot and whose children are the new regions (if any). This tree structure is not contained in the state of the computation, and so must be synthesized by the visualization. Such a synthesized visualization is shown in image (c).

```
I, J : integer;
   ...
I := F1(...);
   ...
J := G1(...);
   ...
I := F2(...);
   ...
```

(a)

Predefined Data Display

I	137
J	123

(b)

Display code

```
procedure  I_Change(I)
 x := I;
 If  x>y
   then display  Black_Indicator
   else display  White_Indicator
end

procedure  J_Change(J)
 y := J;
 If  x>y
   then display  Black_Indicator
   else display  White_Indicator
end
```

Annotated Program

```
I, J : integer;
    ...
I := F1(...);
I_Change(I);
    ...
J := G1(...);
J_Change(J);
    ...
I := F2(...);
I_Change(I);
```

(c)

Declarative Mapping

I > J $\Longrightarrow$ *Black_Indicator*

J $\geq$ I $\Longrightarrow$ *White_Indicator*

(d)

Specification. The specification of visualizations is illustrated using the code fragment shown in part (a). The fragment involves two variables, I and J. The visualization is to indicate whenever $J \geq I$—for example, by drawing an indicator light which is white when $J \geq I$ and black when $I > J$.

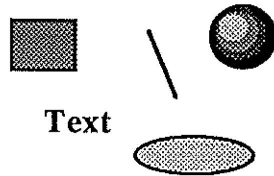
In a system using *predefinition*, it is quite likely that such a visualization could not be constructed. Such systems define the manner in which data must be presented and the animator cannot move beyond these confines. For example, the system might display integers as strings, permitting only the unsatisfactory visualization shown in part (b). However, the construction of the visualization would be essentially automatic.

Part (c) shows an *annotative* approach to the same problem. The animator first writes procedures that, when triggered by events in the computation, update the image. The animator must then identify the locations in the code where these events occur and annotate them with calls to the procedures. As this example indicates, the annotative approach can produce the desired images, but may involve significant effort on the animator's part, particularly in the modification of the program code.

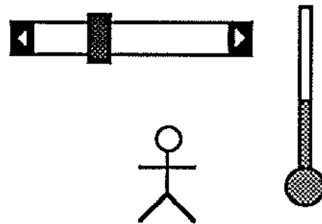
The *declarative* approach is shown in part (d), using a rule-based notation similar to that of Pavane. The animator lists the conditions which are to be detected and indicates what is to be generated for each condition. Explicit modification of the program code is not necessary, although some mechanism (operating at either compile-time or run-time) must be provided by the system to extract the needed information.

Manipulation is unlikely to be used to generate such a simple visualization. A system that provides specification by manipulation typically uses some other mechanism (e.g., annotation) to capture state information. Manipulation is then used to indicate how the data is mapped to the image, and particularly in the specification of complex animations.

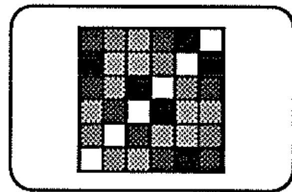
Sidebar: Interface/vocabulary



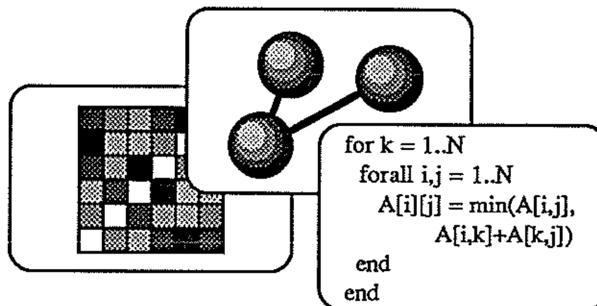
Simple objects



Composite objects



Worlds



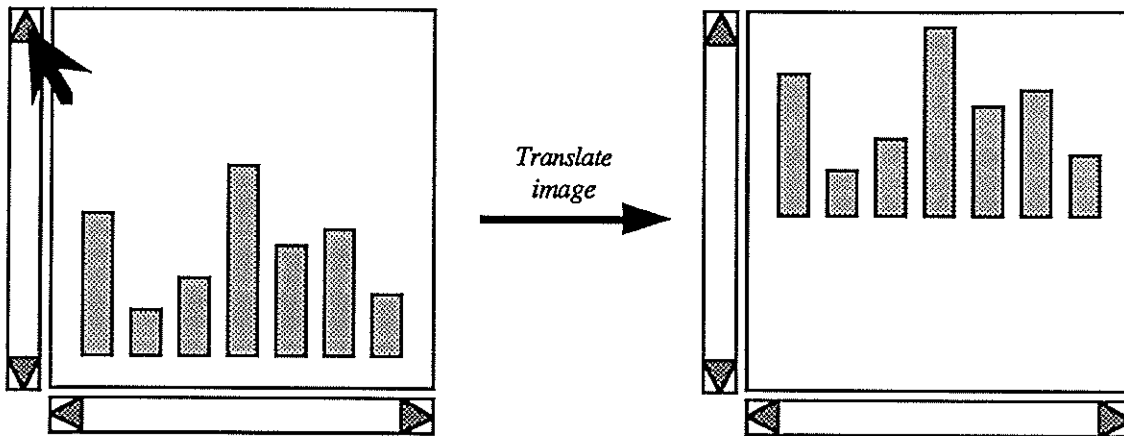
Multiple worlds

Graphical vocabulary. This figure illustrates some common components of the graphical vocabulary, the syntax used by the animator in constructing visualizations. The most basic components of the vocabulary are *simple objects* such as rectangles or lines, text, and icons (small, fixed images, generally with a predefined semantics). These graphical objects are often implemented in an object-oriented fashion, with each graphical object corresponding to an instance of a software object class. The software object's methods include routines to draw the graphical object as well as to change its attributes.

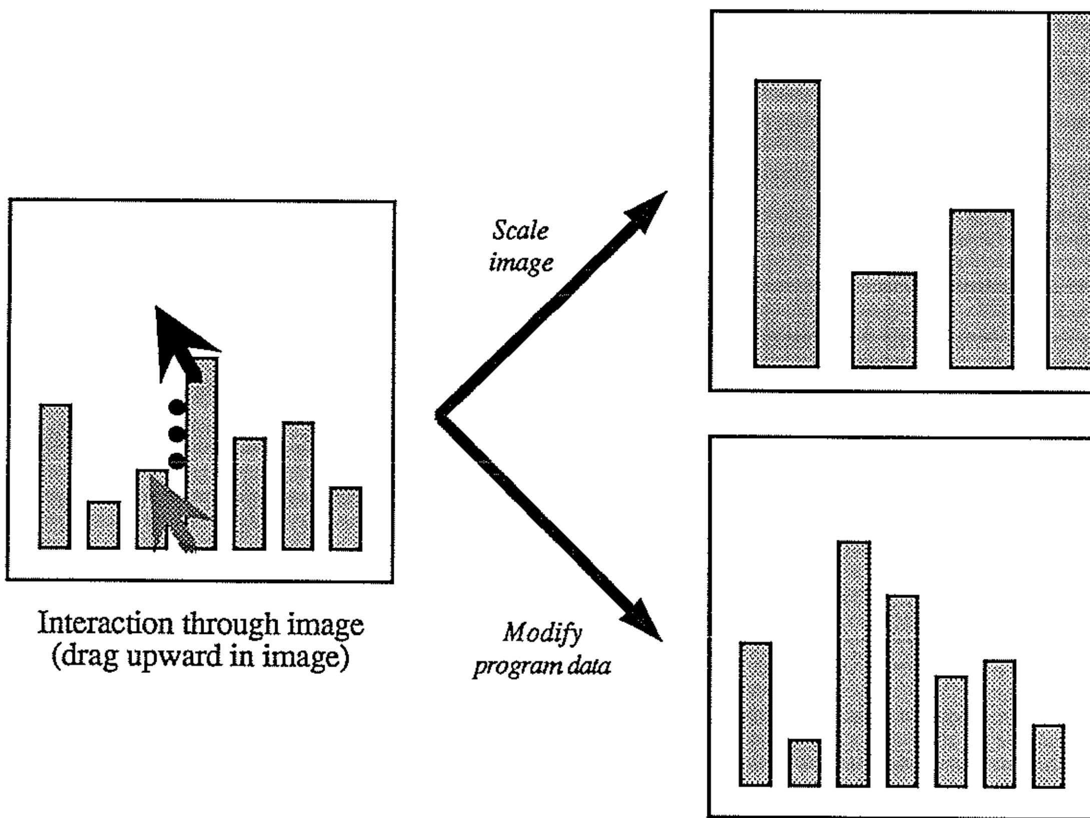
Composite objects are collections of these simple objects, with relationships between the objects. For example, the "thermometer" shown here is a combination of a filled circle, a filled rectangle, and an unfilled

rectangle, arranged in the indicated manner. This can be treated as a single graphical object with position and size attributes. When the object is drawn, the thermometer attributes are used to determine the attributes of the circle and rectangles.

In most systems, the animator generates images by arranging objects (whether simple, complex, or a mixture) in a coordinate space or *world*. The image is generated by viewing the objects from some point—a typical approach, used by Balsa, is to arrange the objects in the X-Y plane and view them from a point "above" this plain. A system may also permit the animator to generate *multiple worlds* (and images) simultaneously, thereby producing several different representations of the same computation.



Interaction through controls
(click a button)



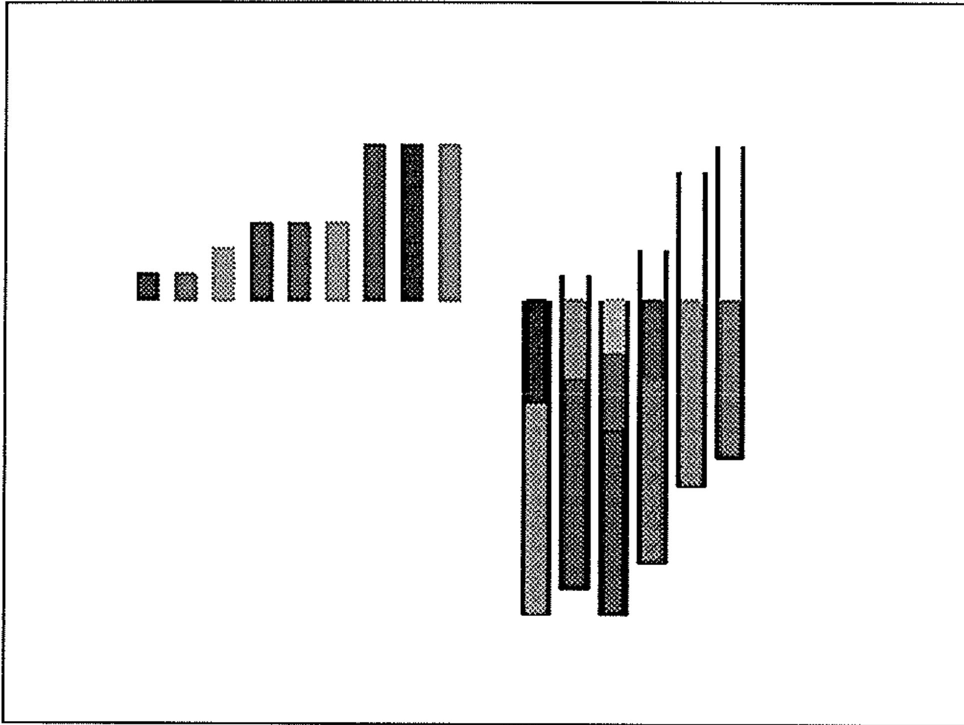
Interaction through image
(drag upward in image)

Interaction. Viewer interactions with a visualization use two primary mechanisms, as illustrated here. The system may provide a number of *controls* which the viewer can use to modify the information that is displayed. This is shown in the top portion of the figure, where clicking on a button (as indicated by the "arrow cursor")

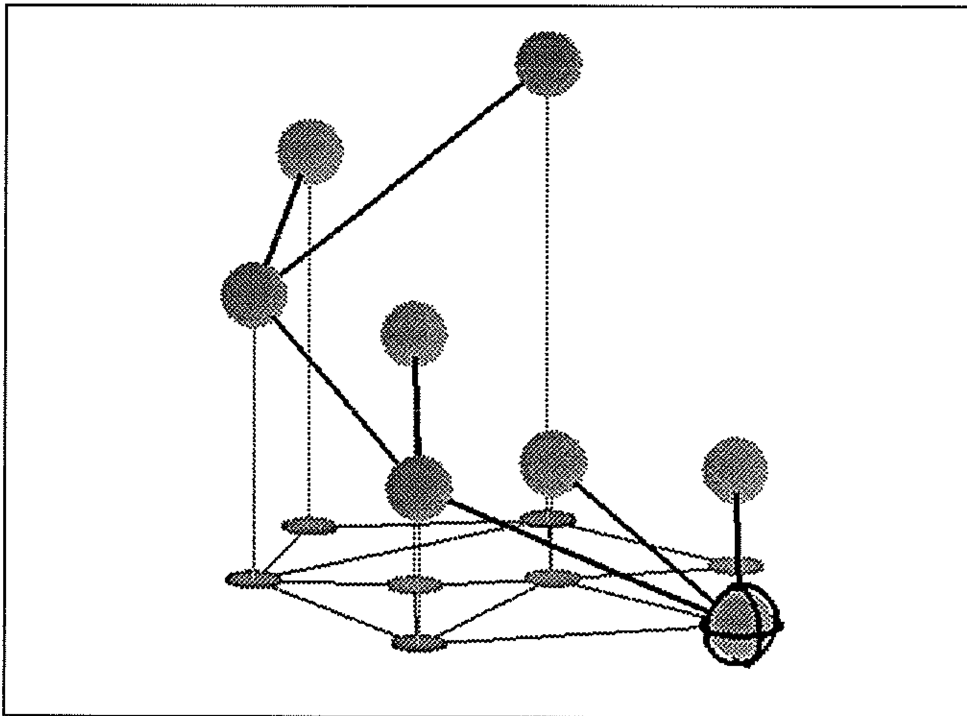
translates the image viewpoint. The function of each control is defined by the system, so there is no ambiguity in the interaction.

This is not true of *interaction through the image*, in which the viewer directly manipulates the display and thereby affects the visualization in some manner. Consider the gesture indicated in the center left, in which the viewer clicks one of the displayed rectangles and drags upward. This simple gesture has several possible interpretations, two of which are shown here. The gesture could indicate a scaling transformation, in which the displayed sizes of all other rectangles are to be increased by the indicated factor. In this interpretation, the manipulation affects the mapping—that is, the transformation from the program data to the image.

Another possible interpretation of the gesture is as a modification of the length of the rectangle, and a corresponding change of the data in the underlying computation. Assume that the rectangles in this image represent an array of numbers; then modifying the length of the third rectangle should induce a corresponding change in the third element of the array. The underlying computation should then proceed using the new value. Such interactions have been illustrated in Zeus using sorting algorithms.



(a)



(b)

Presentation. These two images illustrate some aspects of the presentation of data. Image (a) is from a bin-packing algorithm which places items, each having some weight, into bins of fixed capacity. The total weight of the items in a bin cannot exceed the bin capacity. The rectangles in the left portion of the image represent unpacked items. The open-topped rectangles to the right represent bins, with packed items placed in the bins.

The graphical objects and their attributes represent information at several different levels. Such simple data as the weight of the items and the capacities of the bins are conveyed by the sizes of the graphical objects, while the status of each item—unpacked, or packed into a particular bag—is shown by their positions. The positioning of the graphical objects also encodes more subtle analytical properties, for example that the total weight of the items in a bag is less than the capacity, or that some of the unbagged items do not fit into certain of the bags.

The visualization can also make use of explanatory visual events to guide the viewer. For example, when an item is packed, instead of the rectangle simply disappearing from the "unpacked" region and reappearing in one of the bins, it can move smoothly from its old to its new position. This simultaneously focuses the viewer's attention on the object and allows the viewer to see that the act of packing the item does not change its weight.

An orchestration involving this visualization could be constructed in several ways. For example, several different problem instances—different choices of item weights or bag capacities—could be shown. An alternative approach would be to use this visualization with various bin-backing computations, such as first-fit, best-fit, and worst-fit, to allow the viewer to contrast the performance of these algorithms.

Image (b) is a visualization of a shortest-distance algorithm which shows how analytical presentations can aid in algorithm understanding. The algorithm computes the shortest distance between each pair of nodes in a graph using a two-dimensional array $d[i,j]$. Initially $d[i,j]$ is the length of the edge connecting i and j, if such an edge exists, or infinity otherwise. The algorithm modifies the array in a series of steps, in each of which a node is selected and "scanned". When the algorithm finishes, each $d[i,j]$ contains the distance between nodes i and j. An ad-hoc visualization of this algorithm would simply display the contents of the array, for example as a grid of squares whose colors encode the values stored in the array. Such a visualization would not particularly aid the viewer's understanding.

The analytical approach attempts to display the important properties of the algorithm. In this case, one of the properties says that at all times the value stored in $d[i,j]$ is the length of the shortest path from node i to node j all of whose internal nodes have been scanned. This suggests showing the status, scanned or unscanned, of each node and the shortest paths (these paths are not maintained by the algorithm, but must be synthesized from the contents of the array). Such a visualization is shown in the image. The graph is shown as a planar network of circles and lines. Paths from a particular node (marked with circles) are represented in three dimensions with spheres and lines; the height of the sphere above the planar graph is proportional to the distance of the node from the selected

starting point. Scanned nodes are red and unscanned nodes are green. Viewing this visualization, particularly the way in which the scanning of a node modifies the paths, provides an understanding of the way in which the algorithm computes the distances.