Understanding and Characterizing Program Visualization Systems

Technical Report GIT-GVU-91-17

John T. Stasko
Charles Patterson

Graphics, Visualization, and Usability Center
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

E–mail: {stasko,charliep}@cc.gatech.edu

October 16, 1993

Abstract

The general term *program visualization* refers to graphical views or illustrations of the entities and characteristics of computer programs. This term along with many others including *data structure display*, *program animation*, *algorithm animation*, etc., have been used inconsistently in the literature, which has led to confusion in describing systems providing these capabilities. In this paper we present a scaled characterization of program visualization terms along aspect, abstractness, animation, and automation dimensions. Rather than placing existing systems into hard-and-fast categories, we focus on unique and differentiating aspects across all systems.

Keywords: program visualization, algorithm animation, software understanding

1 Introduction

A visualization tool provides graphical views of the entities and characteristics of a computer system or program. The purpose of such a visualization tool is stated nicely by Myers, et. al.: "Human information processing is clearly optimized for pictorial information, and pictures make the data easier to understand for the programmer[MCS88]." The two-dimensional format of a picture can provide greater amounts of relevant information more fluently than

a stream of text. Programming textbooks reflect this fact when they use the familiar boxes for variables, columns of boxes for arrays, and arrows for pointers. One could argue that many programmers conceptualize algorithms as rough pictures which are then translated into text for the coding phase. The ability to debug and demonstrate a program using images, therefore, can make the programming process easier. Visualization techniques have already made a significant impact on programming language environments[AB89]. Appropriate animated images have also been used for teaching the purpose and functionality of algorithms[Bro88a, Sta90].

In this paper, we focus on graphical views of computer programs, such as illustrations of variables, code sections, the run-time stack, and program semantics. One term that has become accepted for describing this general area is "program visualization." Baecker defines the term as, "the use of the techniques of interactive graphics and the crafts of graphic design, typography, animation, and cinematography to enhance the presentation and understanding of computer programs[Bae86]."

Unfortunately, so many different names are given to so many hybrids of program visualization systems that it is difficult to consistently recognize the purpose of each. For instance, the terms data structure display, program animation, process display, and algorithm animation have all been used to label systems of varying utility. Our goal is to characterize the many types of program visualization, both to provide a clearer meaning of the terms in use, and to attempt a structuring of program visualization tasks to guide further work. Lack of clarity in terms is disadvantageous in that when a new system is developed and described by its creators, the capabilities offered by the system are not quite clear. A precise descriptive scheme provides a framework for designers to describe their work and disseminate information.

Another research area with a similar descriptive name to program visualization is visual programming[Cha87, Shu88], but it differs importantly from the subject matter of this paper. Visual programming involves actual programming through the use of pictures, icons, and graphical entities. A programmer manipulates the visual entities in order to create a

Figure 1: View of a linked list of Pascal records from the MacGnome system. (Figure missing)

semantically meaningful computational process. The matter addressed herein, however, involves the use of pictures to convey information about programs written in traditional textual languages. A good summary of the distinctions between the two can be found in [Mye90].

2 Characterizing Program Visualizations

Taxonomies of program visualization and visual programming systems already exist. Myers has developed a program visualization classification scheme using two axes: whether the systems illustrate the code, data, or algorithm of a program, and whether they are dynamic or static[Mye90]. Singh presents a similar scheme[Sin90]. Our classification system follows roughly from these—we utilize four classifying dimensions with two corresponding closely to Myers' two dimensions mentioned above. But we do not seek to place existing systems into labelled categories. Rather, we show how different systems exhibit varying levels of the four dimensions we have identified. We also concentrate solely on program visualization systems, focusing on characteristics and unique capabilities that different systems offer.

The characterization scheme we propose contains four dimensions: aspect, abstractness, animation, and automation. Below we discuss each of the dimensions in more detail.

2.1 Aspect

Program visualization systems usually focus on a different *aspect* of a program to be visualized. This dimension most closely represents the purpose of the visualization—why the visualization is being created and what parts of the program are being emphasized.

The simplest aspect level of program visualization is just an enhanced presentation of program text. The SEE system[BM90] uses human factors knowledge and typography techniques to display C programs. Debuggers often show the text of programs' procedures as they execute, with line by line highlighting.

Moving beyond purely textual views, some systems provide views of the data and data structures in programs. One of the first general purpose data structure display systems, Incense[Mye83], generates a view of user-specified data structures during debugging. A follow-up system, MacGnome[MCS88], focuses on providing simple canonical Pascal data structure views for novice programmers. A linked list data structure view taken from MacGnome is shown in Figure 1.

Some systems provide views of program aspects beyond pure data structures. For instance, a system may include views of flow-of-control such as a rendering of program subroutines as icons, a call-graph view, views of the run-time stack and dynamic links, etc., in addition to data structure views. We call these types of systems program state visualization systems. The Pecan system[Rei85], a view of which is shown in Figure 2, contains an almost exhaustive set of views including symbol table, data type, stack, flowgraph, and expression displays. The PV system[B$^+$85] provides program structure, flow of control, and program data views, but it also includes views of important phases of the software engineering lifecycle such as diagrams of system requirements.

Yet a further level of program display provides views of the underlying algorithm or higher-level strategy of a program. Algorithm visualization systems are systems that provide visual depictions of the purposeful operations, methodologies, and tactics that programmers utilize in their programs. These types of systems are not concerned with the details of a particular implementation in a programming language. Rather, they focus on the funda-

Figure 2: A Pecan system view containing program text, stack, flowgraph, and debugger windows. (Figure missing)

mental methods utilized to solve a problem. Their displays are inherently semantic, thus differing from views of isolated data.

A graphical view of a comparison sort provides a good example of an algorithm visualization. The view may represent array elements as rectangles and highlight or flash the rectangles when array values are compared prior to a possible exchange. This graphical action involves a mapping from the meaning of the program to the display. Program state visualization systems display code and its syntactic structure, such as scope, but they stop short of showing views of the actual task being performed by the code. One of the first examples of algorithm visualization is the film *Sorting Out Sorting*[BS81], generally accepted as a motivating factor for this research area. The Balsa system[Bro88a] is the prototype for algorithm visualization systems with its high-quality imagery, multiple views, and scripting facilities. Balsa inspired subsequent systems such the Smalltalk based system Animus[Dui86], and ANIM[BK91], a system for building simple algorithm visualizations in a UNIX environment.

Note that algorithm visualization systems often encompass or include data structure and program state visualization capabilities. Many views in the above systems illustrate only program data and its characteristics throughout program execution. Algorithm visualization systems, however, must contain the capabilities to illustrate the higher-level program methodologies.

2.2 Abstractness

Even though program visualization systems may display views of the same aspect of a program, the level of *abstractness* at which the view is presented may vary widely. For example, a data structure display system may render three integer variables named `hours`, `minutes`, and `seconds` as three rectangular boxes containing the individual values as text strings. However, another view of this data structure could display the data in the form of a clock face with appropriate hour, minute, and second hands.

One characterization of the abstractness of a program view is whether the display is iso-

morphic to the program components it represents[Bro88b]. That is, could a data structure be rebuilt from its graphical representation as easily as the representation is created from the data structure?

Algorithm visualizations, as discussed in the previous subsection, typically go beyond isomorphic mappings of program data or code to graphical representations of program semantics. Consequently, algorithm visualizations inherently provide a high level of abstractness, and they have even been defined accordingly[Sta90]. For instance, a visualization of a program performing an exhaustive search might contain a bar representing the number of unsuccessful search attempts it has made. As more unsuccessful attempts accumulate, the bar grows larger. This idea of "incorrect attempts" may not be represented anywhere in the program, but it has semantic meaning with respect to the program's purpose.

To help understand the use of abstractness by program visualization systems, we introduce the concept of *intention content*, the semantics or meaning behind otherwise context-free data and code. Given a task or entity to be visualized, the intention content of the visualization is the level of knowledge about the task's purpose required to map the task to the visualization. Greater amounts of intention content support displays that are more informative and that are more abstract. Also, a greater level of intention content requires a programmer (of the algorithm being viewed) to provide the visualization system with more information on the details of what to display.

For example, consider a view of a sort in an algorithm visualization system. Without any intention content, the sorting display could present an array of values as an indexed list. After each execution cycle or at programmer specified times, the values on the screen in the array would update to match the current state of the process. Actually, this will work with any array, whether or not sorting is involved. More useful didactically are the values of the array displayed as bars of varying height, taller bars for larger values, which seem to trade places as the sort algorithm swaps them. However, this display cannot be standard for all arrays. For instance, an array of indices into another array is meaningless as a row of bars. If a high level of intention content is to be used, the programmer's purpose

is necessary and must be supplied to the system.

Intention content is important in data structure display as well. The lowest intention content level for data structures is represented by the classic box-and-arrow diagram, commonly shown in data structures and programming textbooks. Integers, reals, booleans, etc., each have a distinct box representation complete with name and value. Composite data structures, such as records and structures, are built by using an encompassing box representation around their elements' boxes. Pointers or addresses are represented by arrows to the objects they reference. (Conceptually, the very lowest intention content level would be a string of binary bits of length equal to the computer space used by a data structure, but this is almost always below our needs.)

Figure 3 shows a record data structure, consisting of an integer and an array, interpreted under varying levels of intention content. Each view is a possible interpretation of what the data structure signifies. The view in Figure 3a shows the classic interpretation mentioned above, at the lowest intention content level.

Figure 3b shows a data structure view with more intention content than the one in Figure 3a. Here, the array is shown as a bar graph with the bars scaled according to the accompanying array element's value. An even more specific view, Figure 3c shows a pie graph of the array made possible since the data structure visualization system contains the knowledge that the values represent percentages of components in a whole, such as the percent of elements in a chemical compound. Finally, Figure 3d shows the array data structure and integer value interpreted as a stack. The single integer, which to this point has been an unrelated part of the record containing the array, is the top pointer for the stack. This type of view exhibits a high level of intention content and a high level of abstractness.

Some systems such as the data structure display system Incense[Mye83], support the creation of both low-level concrete views and highly abstract views such as the clock face described above. The system can generate the low-level, canonical views automatically, but it requires programmer assistance (writing display procedures) to generate abstract views.

Figure 3: Illustrating various intention content levels by altering the representation of a data structure.

2.3 Animation

Our third classification dimension describes the dynamics or *animation* shown in program visualization systems. Unfortunately, the term "animation" also has been loosely applied in the past, and a large variety of systems claim to provide animation capabilities. For example, actions such as simply highlighting lines of code as they are executed, altering the boundary style of a graphical object, or changing color intermittently have been called animation.

Fundamentally, animation consists of the rapid sequential display of pictures or images, with the pictures changing gradually over time. These pictures are the frames of the animation. If the imagery's changes from frame to frame are small enough and the speed of displaying the frames is fast enough, the illusion of continuous motion is achieved.

We consider data structure animation systems and program state animation systems to be systems which support repeated display of data structure and program state visualizations, respectively, with changes in view sufficient in both content and time to provide a viewer with the essence of how the data and program transform continuously throughout execution.

As an example of what we mean, consider views of a linked list data structure being built. If, when a new node is added, it is shown immediately at its correct position in the list (say a horizontal row of nodes) this would be considered data structure visualization. But if a view displays a new node being allocated in a special heap memory area, then the node slides over to its correct position in the list, this would be considered a data structure animation.

We consider an even more rigid specification to be met for creating animations of algorithms. To motivate the criteria, we introduce the notion of a *valid configuration*. A valid configuration of a program is a state (data values, context, point of control) of the program that involves semantic meaning and that is reachable during execution. As a program executes, it transforms from valid configuration to valid configuration. These configurations can be at a fine-grain level such as after each line of execution, or at a higher level such as

in a sorting program, after each exchange operation. Valid configurations identify program contexts that make sense in terms of the program's purpose and functionality.

Because an algorithm connotes meaning beyond simple data objects, the transitions between valid configurations, in addition to the configurations themselves, take on added importance. One of the primary goals of algorithm animation is to illustrate not just the set of states a program reaches during execution, but how the transformations between states occur. In order to incorporate this fact, we consider an algorithm animation system to be a system that illustrates a program's behavior by both repeatedly displaying graphical images corresponding to valid configurations and displaying sequences of graphical images that correspond to states "in-between" those configurations. Scene display should occur at a sufficiently brisk pace to provide the illusion of continuous motion.

Essentially, the views that are shown between valid configurations denote configurations that are never realized and carry no semantic meaning. They are produced strictly for aesthetic visual reasons and for illustrating how transitions between valid configurations occur. Perhaps an example best illustrates this concept.

Consider the graphical depiction of a sorting program, discussed earlier, that represents the program's data elements as a row of rectangular images. The two scenes in Figure 4 illustrate two consecutive valid configurations that we might reach during execution of the program. In the second scene, elements 2 and 3 have exchanged their positions from the first scene. The repeated display of such configurations reached during execution does not constitute an algorithm animation according to our characterization. Rather, it would be an algorithm visualization because no intermediate scenes between valid configurations were presented. (Note, this is an algorithm visualization with extremely low intention content, which in fact, could be considered a sophisticated data structure visualization. We use it here to illustrate a point.) On the other hand, Figure 5 shows a superimposed sequence of frames from what we would consider an algorithm animation. In it, the rectangles assume a set of slightly altered positions between the two in Figure 4. If the intermediate scenes were displayed quickly enough, they would present a definite illusion of motion. The important

concept here is that all of the intermediate scenes represent program states that never really exist and have no meaning in terms of the program context. They are purely artificial states, created for the viewing aesthetics of the animation.

An animation of the Towers of Hanoi problem (often used to teach recursion) with disks moving between the separate pegs is another example of a visualization in which showing a series of intermediate frames of the disks' movements is absolutely critical for understanding. Simply presenting a rapid sequence of frames with the disks at their new end-positions (without intermediate movement presented) would be extremely difficult to follow and comprehend.

Brown has characterized specific imagery in algorithm animations along three dimensions: transformation, persistence, and content[Bro88b]. The animation dimension of our scheme coincides similarly with his transformation dimension. Our notion of characterizing animation by the artificial program states presented between valid program configurations, however, helps to clarify the distinction between program visualizations and animations.

One of the earliest systems to recognize the importance of smooth transitions generates algorithm animations of Smalltalk programs by monitoring message passing[LD85]. Later program visualization systems such as Tango[Sta90] provide explicit mechanisms to help produce the in-between configuration views that typify animations. In Tango, view designers develop animations using high-level primitives that hide low-level graphics details. It is still possible, however, to generate more traditional visualization views without the in-between frames in these systems. In fact, some views are more informative when explicit animation (as we characterize it) is not utilized. For example, views involving large data sets and data structures often do not require explicit animation. In these views, the extra frames from animation may slow down the presentation and hinder understanding.

2.4 Automation

We consider the level of *automation* provided for developing program visualizations to be another characterizing factor in these systems. Automation levels can range from totally

Figure 4: Visualizations of consecutive valid configurations from a bubblesort algorithm.

Figure 5: Bubblesort algorithm animation with "in-between" configurations shown.

automatic views generated as a program executes to views requiring explicit programmer design and implementation effort, as well as specification of the appropriate trigger points in the program.

Data structure display system views are usually generated automatically, without explicit programmer support. That is, an execution monitor or debugger examines a program at a specific moment and provides information to generate graphical views of the data structures. This capability, with no turnaround time for view design, is necessary for time-intensive tasks such as debugging. Often, systems support automatic generation of displays, but they also permit viewers to modify the display as desired. The GDBX system[Bas85] provides canonical box-and-arrow displays of Pascal and C programs. It allows viewers to reposition or eliminate data structure views as desired during a debugging session. GELO[RMD89] also includes predefined data views, but it allows users to graphically specify specialized data type displays using topological constraints.

It is possible to think of data structure displays that would be very difficult to generate automatically too. The clock face view discussed earlier would be impossible to automatically generate without some designer assistance and direction. This fact illustrates that our abstraction and automation dimensions usually exist in an inverse relationship. Cre-

ating program visualization views with high levels of abstractness involves a great deal of intention content and simply requires a priori design support.

Program state views such as those of the call graph, run-time stack, or text code can be generated automatically by a program visualization system. Again, these views are extremely useful for debugging which requires little of no view set-up time. For instance, VIPS[ISO87] generates multiple run-time views of Ada programs, including data, block structure, and debugger interaction windows. Often, the most difficult part of building a system to display program state is not the generation of the graphics, but the acquisition of the run-time execution data and information driving the graphics. This may require low-level coding that examines compiler information, symbol table access, or debugger internals.

As discussed earlier, algorithm visualization systems provide visual depictions of the semantic notions and abstractions used in computer programs and processes. Algorithm visualizations can display program information that is not immediately evident or that cannot be automatically deduced by examining the program state during execution. That is, algorithm visualizations require high levels of intention content from a programmer. They are usually hand-crafted, user-conceptualized views of what is "important" about a program, so they require a designer to specify and implement the graphics that accompany a program.

Consequently, algorithm visualizations (and particularly animations), virtually by definition, exhibit a low level of automation. Recently, systems providing algorithm views without explicit end-designer support have appeared[HWF90], but they are restricted to specific algorithm domains and they require considerable compiler crafting. Brown makes strong arguments why, under current constraints, explicit programmer design for algorithm animations is typically required and desirable[Bro88b].

Nevertheless, recent algorithm animation work has focused on reducing the burden of view design and implementation. These efforts still provide a designer with artistic freedom, but they strive to provide tools which make view development easier and more fun. The Aladdin system[HHR89] uses a declarative mechanism to specify view layout. Programmers

	Aspect	Abstractness	Animation	Automation
Data Structure Display	low	low	low	high
Program State Visualization	medium	low	low	high
Program Animation	medium	medium	medium	high
Algorithm Visualization	high	high	low	low
Algorithm Animation	high	high	high	low

Table 1: Tabular breakdown of how accepted program visualization terms fit within our characterization.

interleave graphical specifications throughout a program, which is then executed to generate the visualization. The Gestural system[Dui87] and the Dance animation editor[Sta91] both allow designers to "visually program" their desired program visualization via direct manipulation.

3 Taxonomy

We believe that our method of characterizing program visualization systems provides a framework for understanding and discussing these systems in more detail than has been previously available. One possibility for a subsequent taxonomy is to scale the dimensions from 0 to 10 and then summarize each existing program visualization system via a 4-tuple of values. This is not our intent. In fact, we believe such a classification would be extremely difficult because many systems exhibit varying levels of each of the four dimensions.

We prefer to characterize existing accepted terms for subareas of program visualization along our dimensions as a way of more clearly formalizing what each term means. In Table 1, we characterize a group of program visualization areas according to our perception of how they fit within our scheme.

Each area is categorized as low, medium, or high in the four dimensions as follows: Aspect ranges from data structure (low) to program state (medium) to algorithm (high); Abstractness ranges from concrete (low) to abstract (high); Animation ranges from visualization (low) to animation (high); Automation ranges from little or no system assistance (low) to automatic generation (high).

We characterize the terms data structure display, program state visualization, program

animation, algorithm visualization, and algorithm animation. For example, if a system in now described as providing data structure display support, this means that the system can minimally produce automatic, concrete data structure views with little dynamics or animation. As new types of systems are developed, the need for further descriptive names will emerge. In fact, horizontal designations not residing in our chart such as all high characterizations or low-Aspect, high-Abstractness, medium-Animation, and high-Automation signify open areas of research.

4 Summary

We have presented, under the general area of program visualization, a characterization along the four dimensions of aspect, abstractness, animation, and automation. We clarified the meaning of the dimensions by illustrating how specific system aspects fit within them. The notion of intention content was introduced to help explain the abstractness dimension. We also clarified the differences between visualization and animation by making the distinction that animation presents views of a program between its valid configurations. Finally, we characterized common program visualization subareas according to how they fit within our framework.

References

[AB89] Allen L. Ambler and Margaret M. Burnett. Influence of visual technology on the evolution of language environments. *Computer*, 22(10):9–22, October 1989.

[B+85] Gretchen P. Brown et al. Program visualization: Graphical support for software development. *Computer*, 18(8):27–35, August 1985.

[Bae86] Ronald M. Baecker. An application overview of program visualization. *Computer Graphics: SIGGRAPH '86*, 20(4):325, July 1986.

[Bas85] David B. Baskerville. Graphic presentation of data structures in the DBX debugger. Technical Report UCB/CSD 86/260, University of California at Berkeley, Berkeley, CA, October 1985.

[BK91] Jon L. Bentley and Brian W. Kernighan. A system for algorithm animation. *Computing Systems*, 4(1), Winter 1991.

[BM90] Ronald M. Baecker and Aaron Marcus. *Human Factors and Typography for More Readable Programs*. Addison-Wesley, Reading, MA, 1990.

[Bro88a] Marc H. Brown. Exploring algorithms using Balsa-II. *Computer*, 21(5):14–36, May 1988.

[Bro88b] Marc H. Brown. Perspectives on algorithm animation. In *Proceedings of the ACM SIGCHI '88 Conference on Human Factors in Computing Systems*, pages 33–38, Washington D.C., May 1988.

[BS81] Ronald M. Baecker and David Sherman. Sorting Out Sorting. 16mm color sound film, 1981. Shown at SIGGRAPH '81, Dallas TX.

[Cha87] Shi-Kuo Chang. Visual languages: A tutorial and survey. *IEEE Software*, 4(1):29–39, January 1987.

[Dui86] Robert A. Duisberg. Animated graphical interfaces using temporal constraints. In *Proceedings of the ACM SIGCHI '86 Conference on Human Factors in Computing Systems*, pages 131–136, Boston, MA, April 1986.

[Dui87] Robert A. Duisberg. Visual programming of program visualizations. A gestural interface for animating algorithms. In *Proceedings of the 1987 IEEE Computer Society Workshop on Visual Languages*, pages 55–66, Linkoping, Sweden, August 1987.

[HHR89] Esa Helttula, Aulikki Hyrskykari, and Kari-Jouko Räihä. Graphical specification of algorithm animations with Aladdin. In *Proceedings of the 22nd Hawaii International Conference on System Sciences*, pages 892–901, Kailua-Kona, HI, January 1989.

[HWF90] Robert R. Henry, Kenneth M. Whaley, and Bruce Forstall. The University of Washington illustrating compiler. *Sigplan Notices: SIGPLAN '90*, 25(6):223–233, June 1990.

[ISO87] Sadahiro Isoda, Takao Shimomura, and Yuji Ono. VIPS: A visual debugger. *IEEE Software*, 4(3):8–19, May 1987.

[LD85] Ralph L. London and Robert A. Duisberg. Animating programs using Smalltalk. *Computer*, 18(8):61–71, August 1985.

[MCS88] Brad A. Myers, Ravinder Chandhok, and Atul Sareen. Automatic data visualization for novice Pascal programmers. In *Proceedings of the 1988 IEEE Computer Society Workshop on Visual Languages*, pages 192–198, Pittsburgh, PA, October 1988.

[Mye83] Brad A. Myers. A system for displaying data structures. *Computer Graphics: SIGGRAPH '83*, 17(3):115–125, July 1983.

[Mye90] Brad A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1(1):97–123, March 1990.

[Rei85] Steve P. Reiss. Pecan: Program development systems that support multiple views. *IEEE Transactions on Software Engineering*, SE-11(3):276–285, March 1985.

[RMD89] Steven P. Reiss, Scott Meyers, and Carolyn Duby. Using GELO to visualize software systems. In *Proceedings of the ACM '89 SIGGRAPH Symposium on User Interface Software and Technology*, pages 149–157, Williamsburg, VA, November 1989.

[Shu88] Nancy C. Shu. *Visual Programming.* Van Nostrand Reinhold, New York, NY, 1988.

[Sin90] Gurminder Singh. Graphical support for programming: A survey and taxonomy. In T. S. Chua and T. L. Kunii, editors, *Proceedings of CG International '90*, pages 331–359. Springer-Verlag, 1990.

[Sta90] John T. Stasko. TANGO: A framework and system for algorithm animation. *Computer*, 23(9):27–39, September 1990.

[Sta91] John T. Stasko. Using direct manipulation to build algorithm animations by demonstration. In *Proceedings of the ACM SIGCHI '91 Conference on Human Factors in Computing Systems*, pages 307–314, New Orleans, LA, May 1991.